Notice

Creation of derivative works unless agreed to in writing by the copyright owner is forbidden. No portion of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission from the copyright holder.

Texas Instruments reserves the right to update this Guide to reflect the most current product information for the spectrum of users. If there are any differences between this Guide and a technical reference manual, references should always be made to the most current reference manual. Information contained in this publication is believed to be accurate and reliable. However, responsibility is assumed neither for its use nor any infringement of patents or rights of others that may result from its use. No license is granted by implication or otherwise under any patent or patent right of Texas Instruments or others.

Copyright © 1997 - 2011 by Texas Instruments Incorporated. All rights reserved.

Training Technical Organization
Texas Instruments, Incorporated
6500 Chase Oaks Blvd, MS 8437
Plano, TX  75023
(214) 567-0973

Revision History
October 2002, Version 1.0
October 2002, Version 1.0a  (Errata only)
April 2003, Version 1.1
  –  Lab solutions for C64x (as well as C67x)
  –  Labs now use CCS v2.2
January 2005, Version 1.2
  –  Add support for CCS 3.0
  –  Add tutorials for code tuning tools
August 2005, Version 1.3
  –  Add C64x+ CPU information
December 2007, Version 1.4
  –  Add support for CCS 3.3
March 2011, Version 1.51
  –  Partial support for CCSv4
  –  Updated for new devices (C674x, C66x)
What Will You Accomplish This Week?

When you leave the workshop at the end of the week, you should be able to perform certain tasks and make critical assessments and decisions about the C6000s’ capabilities. We developed this list based on customer feedback over the past 6 years and our own workshop design experience spanning the past 25 years. All of the modules, exercises, and labs support these accomplishments (as you’ll see when we discuss the workshop’s agenda).

The first two accomplishments are really the overall objectives of the entire workshop. Many students attend the workshop to meet these two needs. The rest of the list supports these two objectives and provides more insight into the expected outcomes. We hope this list meets or exceeds most of your expectations. If you think about it, we’re going through the equivalent of a college semester course in 4 days! We obviously can’t discuss everything given the time limitations, but we have provided the fastest path toward understanding, using and becoming confident in these activities.

<table>
<thead>
<tr>
<th>What Will You Accomish?</th>
</tr>
</thead>
<tbody>
<tr>
<td>When you leave the workshop, you should be able to…</td>
</tr>
<tr>
<td>❧ <strong>Evaluate</strong> C6000’s ability to meet your system requirements</td>
</tr>
<tr>
<td>❧ <strong>Compare/contrast</strong> C6000 to other processors you have used or evaluated</td>
</tr>
<tr>
<td>❧ Write optimized <strong>C</strong> and <strong>assembly code</strong></td>
</tr>
<tr>
<td>❧ Decide when to use <strong>C vs. ASM</strong> and how to mix them</td>
</tr>
<tr>
<td>❧ Write highly optimized, <strong>interruptible</strong> code</td>
</tr>
<tr>
<td>❧ Understand how <strong>cache</strong> works; optimize for it; examine its side affects</td>
</tr>
<tr>
<td>❧ Analyze implications of a <strong>fixed-point processor</strong></td>
</tr>
<tr>
<td>❧ Use <strong>development tools</strong> to compile, optimize, assemble, link, debug and benchmark code</td>
</tr>
</tbody>
</table>

So, if your need falls “inside the box”, be prepared to ask questions when the topic comes up. If your need falls “outside the box”…
What We Won’t Cover…

It’s very important to us to set the correct expectations right up front. This includes describing what we intend to discuss (accomplishments) as well as what we won’t have time to cover. We have chosen, based on time constraints and our experience over the years, to explicitly not cover certain topics. Not only do we expect a certain level of knowledge coming into the workshop (pre-requisites such as some C programming, basic assembly, understanding basic engineering terms and system concepts, etc), we also want to specifically state what won’t be covered during the week. This list includes DSP Theory, algorithms, and specific applications.

Regarding DSP Theory, we will not cover topics such as IIR/FIR filters, convolution, FFTs, and the rest of the topics addressed by the numerous DSP theory books and college courses. We assume that you know this theory if need to apply it. Our job is to show you how to use the device to accomplish these tasks (i.e. the CPU and instruction set) – instead of spending time showing the actual theory (the WHY’S). Algorithms are defined as PID, servo, VSELP, GSM, Viterbi, etc – all of the software pieces that make up a specific software application. We do not have time to dive into any one specific algorithm (and if we did, it’d probably not be the one you wanted). Again, once you’ve completed the workshop, you should have the ability to write C and/or assembly code to tackle each of these algorithms. Specific hardware applications include PCI, E1, T1, AC’97, MVIP and major software applications. We do provide details about on-chip hardware peripherals, which you can apply to the various hardware/software applications, required by your system – we just don’t intend to show the details of any specific application.

What We Won’t Cover and Why...

Workshop Scope and Depth

- In 4 days, it is impossible to cover everything. However, we do cover an equivalent of a college semester course on the C6000.
- We’ve chosen the “Accomplishments” list based on customer feedback and years of workshop experience.
- Many app notes have been written to address specific topics not covered in the workshop (check out the TI website).
- If you have a need that falls “outside the box”, please inform your instructor. Often, they can offer answers/ideas before or after class.

We’ve had to make some decisions about the material in the workshop based on time and what makes sense for all users. Many app notes have been written (and are available on the TI web site at http://www.ti.com) which cover, in detail, many of the topics we cannot here. So, if your need falls “outside the box” (i.e. in addition to the accomplishment list discussed previously), then you have two options: (1) ask the instructor if a manual or app note is available which addresses your specific issue; or, (2) let the instructor know before or after class time – we might be able to shed some light or direct you to other resources. Please communicate your need and we will do our best to fulfill it.
Workshop Outline

The first morning of the workshop covers the basic architecture and pipeline of the ‘C6x DSP microprocessor family from Texas Instruments. In the afternoon you’ll begin writing and debugging code. The next day and a half focuses on optimizing C and assembly code for speed. The final day and a half discusses system, peripheral, and hardware issues.

<table>
<thead>
<tr>
<th>C6000 Optimization Workshop</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture</strong></td>
</tr>
<tr>
<td>0. Welcome</td>
</tr>
<tr>
<td>1. C6000 Architectural Overview</td>
</tr>
<tr>
<td>2. Using CCS with C Programs</td>
</tr>
<tr>
<td><strong>Getting Code to Run</strong></td>
</tr>
<tr>
<td>3. Introduction to the Pipeline</td>
</tr>
<tr>
<td>4. Calling Functions From C (C Environment)</td>
</tr>
<tr>
<td>5. Using the Assembly Optimizer</td>
</tr>
<tr>
<td>6. Architecture Details</td>
</tr>
<tr>
<td><strong>Optimizing Code</strong></td>
</tr>
<tr>
<td>7. Optimization Methods</td>
</tr>
<tr>
<td>8. Software Pipelining</td>
</tr>
<tr>
<td>9. Advanced C Performance Optimizations</td>
</tr>
<tr>
<td>10. Tuning Code Size</td>
</tr>
<tr>
<td><strong>System Issues</strong></td>
</tr>
<tr>
<td>11. Basic Memory Management (Linking)</td>
</tr>
<tr>
<td>12. Advanced Memory Management</td>
</tr>
<tr>
<td>13. Internal Memory &amp; Cache</td>
</tr>
<tr>
<td>14. Packaging an Algorithm</td>
</tr>
<tr>
<td>15. Optimizing Interruptible Code</td>
</tr>
<tr>
<td>16. Numerical Issues (Fixed-Pt)</td>
</tr>
</tbody>
</table>
Introductions

Learning more about you, your application, and your experience will help your instructor tailor the materials to the class needs. This is important since there is more information than can be taught during a single week.

Introduce Yourself

**Briefly, a little about you:**
- Name & Company
- Application
- Which C6000 DSP do you plan to use?

**And, a little about your experience:**
- Do you have experience with DSP?
  - TI DSP's (TMS320)
  - Another DSP
  - Other microprocessors
- Programmed in C, Assembly, or both
- Have you used an OS or RTOS?
## TI Embedded Processors Portfolio

<table>
<thead>
<tr>
<th>Microcontrollers</th>
<th>ARM-Based</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>16-bit</strong></td>
<td><strong>32-bit</strong></td>
<td><strong>DSP</strong></td>
</tr>
<tr>
<td>MSP430</td>
<td>C2000™</td>
<td>C6000 (C62x/67x/64x/64x+)</td>
</tr>
<tr>
<td>Ultra-Low Power</td>
<td>Fixed &amp; Floating Point</td>
<td>Leadership DSP</td>
</tr>
<tr>
<td>Up to 25 MHz</td>
<td>Up to 300 MHz</td>
<td>Performance</td>
</tr>
<tr>
<td>Flash 1 KB to 256 KB</td>
<td>Flash 64 KB to 1 MB</td>
<td>Up to 3 MB</td>
</tr>
<tr>
<td>Analog I/O, ADC, LCD, USB, RF</td>
<td>USB, ADC, CAN, SPI, I²C</td>
<td>L2 Cache</td>
</tr>
<tr>
<td>Measurement, Sensing, General Purpose</td>
<td>Motor Control, Digital Power, Lighting, Sensing</td>
<td>10 EMAC, SDIO, DD2, PCI-66</td>
</tr>
<tr>
<td>$0.49 to $9.00</td>
<td>$1.50 to $20.00</td>
<td>$4.00 to $99.00+</td>
</tr>
</tbody>
</table>

| **32-bit**       | **32-bit ARM** | **ARM + DSP** |
| ARM              | Industry Std Low Power | ARM9 Cortex A-8 |
| Up to 300 MHz    | <100 MHz | Industry-Std Core, High-Performance |
| Flash 32 KB to 512 KB | Flash 64 KB to 1 MB | 4800 MMACs/1.07 DMIPS/MHz |
| Analog I/O, ADC, LCD, USB, RF | USB, SPI, CAN, SPI, I²C | USB, EMAC, MMAC |
| Measurement, Sensing, General Purpose | Host Control | Linux/WinCE User Apps |
| $2.00 to $8.00   | $8.00 to $35.00 | $12.00 to $65.00 |

| **DSP**          | **ARM + DSP** |
| ARM9 Cortex A-8  | ARM9 Cortex A-8 |
| Industry-Std Core, High-Performance | Industry-Std Core, High-Performance |
| 4800 MMACs/1.07 DMIPS/MHz | 4800 MMACs/1.07 DMIPS/MHz |

### Different Needs? Multiple Families!

#### Lowest Cost
- Control Systems
  - Segway
  - Motor Control
  - Storage
  - Digital Ctrl Systems

#### Efficiency
- Best MIPS per Watt / Dollar / Size
  - Wireless phones
  - Internet audio players
  - Digital still cameras
  - Modems
  - Telephony
  - VoIP

#### Max Performance with Best Ease-of-Use
- Multi Channel and Multi Function App’s
- Wireless Base-stations
- DSL
- Imaging & Video
- Home Theater
- Performance Audio
- Multi-Media Servers
- Digital Radio
Fixed Point

- 16-bit fixed-point is most widely used format in history (range: 64K)
- This format provides a balance between low cost and enough precision to accomplish most tasks
- Drawback is that it's easy to have an overflow error. Similar to when you keep multiplying on a calculator until you get an ERROR, algorithms could overflow during runtime.
- For ease of use (not having to worry about overflow and such) most algorithms are created using floating-point.
- Bottom line, if you take the extra time and expense to convert your algorithms to fixed-point, you can usually achieve a lower cost system.

Floating Point

- Most commonly found as 32-bit floating-point. With 24-bits of integer precision (called mantissa) along with an 8-bit exponent. (very large range: $3.40282346 	imes 10^{38}$)
- Writing floating-point numbers looks like: $0.31416 	imes 10^{34}$
- Traditionally, floating-point has been more expensive since it requires more bits.
- While more 16-bit fixed point devices are sold, more applications are developed with 32-bit floating-point. Except for high volume, high-speed systems, floating-point is easier to use – hence, time to market is shorter
### Summary of DSP Devices by Generation

<table>
<thead>
<tr>
<th>Fixed-Point Cores</th>
<th>Float-Point Cores</th>
<th>DSP</th>
<th>DSP+DSP (Multi-core)</th>
<th>ARM+DSP (Integra, DaVinci)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C62x</td>
<td>C67x</td>
<td>C620x, C670x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C621x</td>
<td>C67x</td>
<td>C6211, C671x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C64x</td>
<td></td>
<td>C641x DM642</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C67x+</td>
<td>C672x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C64x+</td>
<td>DM643x C645x</td>
<td>C647x</td>
<td>DM64xx, OMAP35x, DM37x</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C6748</td>
<td></td>
<td>OMAP-L138* C6A8168</td>
</tr>
<tr>
<td>C66x</td>
<td>Future</td>
<td>C6670 C667x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ARM + DSP

What Types of Processing Do You Need?
For example, in an Audio/Video application, what needs to be done?

- User Controls, GUI, OSD
- Peripheral Drivers
- Ethernet (other system comm)
- Video processing decoding, encoding, analytics, etc.
- Audio processing decoding, encoding, filtering, etc.
- Other Signal Processing...

Key System Blocks
An integrated solution that reduces System complexity, Power consumption, and Support costs

- **Low Power**
  - No heat sink or fan required. Ideal for end equipment that require air-tight, sealed enclosures

- **ARM Core**
  - High performance processors (375mhz - 1.5ghz) drive complex applications running on Linux, WinCE or Android systems

- **Graphics Accelerator**
  - Provides rich image quality, faster graphics performance and flexible image display options for advanced user interfaces

- **Prog. Real-time Unit (PRU)**
  - Use this configurable processor block to extend peripheral count or I/F's
  - Tailor for a proprietary interface or build a customized system control unit

- **Display Subsystem**
  - Off-loads tasks from the ARM, allowing development of rich "iPhone-like" user interfaces including graphic overlays and resizing without the need for an extra graphics card

- **C6x DSP Core**
  - Off-load algorithmic tasks from the ARM, freeing it to perform your applications more quickly
  - Allows real-time multi-media processing expected by users of today's end-products
  - Think of the DSP as the ultimate, programmable hardware accelerator
  - Video Accelerator – either stand-alone or combined with the DSP provide today’s meet today’s video demands with the least power req’d

- **Peripherals**
  - Multiplicity of integrated peripheral options tailored for various wired or wireless applications – simplify your design and reduce overall costs

**NOTE** Features not available on all devices
ARM Processors: ARM+DSP

- ARM9 and Cortex-A8 provide the horsepower required to run high-level operating systems like: Linux, WinCE, and Android
- ARM926 processor (375 – 450MHz) is the most popular and widely used processor in the market
- ARM Cortex™-A8 processor (600 MHz – 1.5 GHz) is ideal for high compute and graphic intense applications
Looking at the ARM options ...
Integra (ARM + DSP) ‘C6A8168

Where To Go For More Information

For support we suggest you try TI’s web site first – especially the wiki and e2e forums. Then call your local support – either your local TI representative or Authorized Distributor Sales/FAE. Of course, we provide other workshops which may help to round-out your skills.

Where can I get additional skills?

- **Building Linux based Systems**
  (ARM or ARM+DSP processors)
- **Building BIOS based Systems**
  (DSP processors)
- **Developing Algos for C6x DSP’s**
  (Are you writing/optimizing algorithms for latest C64x+ or C674x DSP’s CPU’s)

DaVinci / OMAP / Sitara System Integration Workshop using Linux (4-days)
www.ti.com/training

System Integration Workshop using DSP/BIOS (4-days)
www.ti.com/training

C6000 Optimization Workshop (4-days)
www.ti.com/training

Online Resources:
- OMAP / Sitara / DaVinci Wiki
  http://processors.wiki.ti.com
- TI E2E Community (videos, forums, blogs)
  http://e2e.ti.com
- This workshop presentation & exercises

Finally, we invite you to check-out, and participate, in our open source projects (Gforge) and documentation (wiki).
C6000 Workshop Comparison

The following briefly summarizes the differences between the two C6000 Workshops. Here’s a quick visual comparison:

<table>
<thead>
<tr>
<th>Audience</th>
<th>BIOS IW</th>
<th>OP6000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm Development and Optimization</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>System Integration (data I/O, peripherals, scheduling, etc.)</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C6000 Hardware</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Architecture &amp; Pipeline Details</td>
<td>✓</td>
</tr>
<tr>
<td>Using Peripherals (EDMA3, McASP, SIO drivers, etc)</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tools</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler, Optimizer, Assembly Optimizer, Profiler, Simulator, etc</td>
<td>✓</td>
</tr>
<tr>
<td>CCSv4, BIOS instrumentation, EVM H/W, HexAIS (boot image)</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coding &amp; System Topics</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C Performance Techniques, Adv. C Runtime Environment</td>
<td>✓</td>
</tr>
<tr>
<td>Programming in Linear Assembly</td>
<td>✓</td>
</tr>
<tr>
<td>Software Pipelining Loops</td>
<td>✓</td>
</tr>
<tr>
<td>DSP/BIOS, Real-Time Analysis</td>
<td>✓</td>
</tr>
<tr>
<td>Creating a Standalone System (Boot)</td>
<td>✓</td>
</tr>
</tbody>
</table>

In general, the C6000 BIOS/Integration Workshop covers overall system design:
- Developing an real-time embedded system
- Using BIOS to easily implement multiple system threads and priorities
- Getting data into/out-of your device via I/O peripherals and drivers
- Preparing your program to boot and initialize the C6000

The OP6000 Workshop is focused on algorithm-level development:
- Writing your code in C or Linear Assembly languages
- Using intrinsics – and other means – to optimize your C code
- Profiling your code to discover CPU cycle used

Both workshops cover the following topics:
- Internal memory and cache
- Building projects with Code Composer Studio (BIOS/IW uses CCSv4; OP6000 uses mix of CCSv3.3 and CCSv4)
- Hardware interrupts (BIOS/IW includes a lab exercise)
- Packaging an algorithm (and calling with the Codec Engine framework)
- Numerical issues
Where To Go For More Information

Suggested Literature

It is easiest to search for the manuals by their “SPR” code. In most cases, this code number is the name of the Acrobat (PDF) file. For example, the CPU and Instruction Set Reference Guide is documented in the PDF file: SPRU189g.pdf (where “g” is the revision code).

### Key C6000 Manuals

<table>
<thead>
<tr>
<th>Manual Type</th>
<th>C64x/C64x+</th>
<th>C674</th>
<th>C66x</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Instruction Set Ref Guide</td>
<td>SPRU732</td>
<td>SPRUE8</td>
<td>SPRUGH7</td>
</tr>
<tr>
<td>Megamodule/Corepac Ref Guide</td>
<td>SPRU871</td>
<td>SPRUFK5</td>
<td>SPRUGW0</td>
</tr>
<tr>
<td>Peripherals Overview Ref Guide</td>
<td>SPRUE52</td>
<td>SPRUFK9</td>
<td>N/A</td>
</tr>
<tr>
<td>Cache User's Guide</td>
<td>SPRU62</td>
<td>SPRUG82</td>
<td>SPRUGY8</td>
</tr>
<tr>
<td>Programmers Guide</td>
<td>SPRU198</td>
<td>SPRU198</td>
<td>SPRAB27</td>
</tr>
</tbody>
</table>

**DSP/BIOS Real-Time Operating System**
- SPRU423 - DSP/BIOS (v5) User's Guide
- SPRU403 - DSP/BIOS (v5) C6000 API Guide
- SPRUEX3 - SYS/BIOS (v6) User's Guide

**Code Generation Tools**
- SPRU186 - Assembly Language Tools User's Guide

To find a manual, at www.ti.com and enter the document number in the Keyword field:

**Note:** For the latest version of the manuals, please check on the TI website (www.ti.com). The manuals and applications notes found on the CCS Installation Disc are current at the time CCS is released, but may become outdated over time.
## For More Generic DSP Information

### Looking for Literature on DSP?

- **“A Simple Approach to Digital Signal Processing”**  
  by Craig Marven and Gillian Ewers;  
  ISBN 0-4711-5243-9

- **“DSP Primer (Primer Series)”**  
  by C. Britton Rorabaugh;  
  ISBN 0-0705-4004-7

- **“Understanding Digital Signal Processing”**  
  by Richard G. Lyons;  
  Prentice Hall; 2nd edition (March 15, 2004)  
  ISBN 0-1310-8989-7

- **“DSP First: A Multimedia Approach”**  
  James H. McClellan, Ronald W. Schafer, and Mark A. Yoder;  
  ISBN 0-1324-3171-8
DSP Books which include the C6000

Looking for Books on ‘C6000 DSP?

- “Digital Signal Processing Implementation using the TMS320C6000TM DSP Platform”
  by Naim Dahnoun; ISBN 0201-61916-4

- “C6x-Based Digital Signal Processing”

- “Real-Time Digital Signal Processing: Based on the TMS320C6000” by Nasser Kehtarnavaz; Newnes; Book & CD-Rom (July 14, 2004)

- “Digital Signal Processing and Applications with the C6713 and C6416 DSK (Topics in Digital Signal Processing)”
  Wiley-Interscience; Book & CD-Rom (December 3, 2004)
  by Rulph Chassaing;
  ISBN 0-471-69007-4

Looking for Books on ‘C6000 DSP?

- “Real-Time Digital Signal Processing from Matlab to C with the TMS320C6x DSK” by Thad B. Welch;
  Cameron Wright; Michael Morrow; Book & CD-Rom (2006) ISBN 0-8493-7382-4
Administrative Details

Let’s get some of the administrative details out of the way …

<table>
<thead>
<tr>
<th>Administrative Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name Tags</td>
</tr>
<tr>
<td>Start &amp; End Times</td>
</tr>
<tr>
<td>Bathrooms</td>
</tr>
<tr>
<td>Phone calls</td>
</tr>
<tr>
<td>Lunch !!!</td>
</tr>
<tr>
<td>Let us know if you’ll miss part of the workshop</td>
</tr>
</tbody>
</table>
Introduction

This overview introduces the basic architecture of the TMS320C6000 in a tutorial fashion. Each concept and piece of the architecture builds upon the previous. Beginning with a simple *sum-of-products* algorithm we develop the architecture step-by-step while writing the code solution. Obviously, the next few pages cannot cover every detail of the ‘C6000 (why else would it be called an *overview*?). However, this first module will get the wheels turning describing the architecture from the ground up.

Learning Objectives

After reading and studying this module, you should have a basic understanding of the core CPU architecture, memory configuration, peripherals, tools, and instruction set of the ‘C6000 family.

- Describe the basic C6000 CPU architecture and instruction set
- Draw the C6000’s basic memory block diagram
- List the various peripherals for the C6000 devices; specifically, the EDMA3 and buses
Chapter Topics

Architectural Overview ............................................................................................................................ 1-1

Building the 'C6000 Architecture from Start to Finish ................................................................. 1-3
'C6000 System Diagram ...................................................................................................................... 1-3
What Problem Are We Trying To Solve? ............................................................................................. 1-4
Sum of Products (Example) .................................................................................................................. 1-7
Multiply (.M Unit) .................................................................................................................................. 1-8
Add (.L Unit) .......................................................................................................................................... 1-9
Register File ............................................................................................................................................ 1-10
Specifying Register Names .................................................................................................................. 1-11
Creating a Loop ..................................................................................................................................... 1-12
Branching (.S Unit) ............................................................................................................................... 1-13
Creating a Loop Counter (MVK) ............................................................................................................ 1-14
Decrementing the Loop Counter ........................................................................................................ 1-14
Conditional Instructions ...................................................................................................................... 1-15
Using Conditional Instructions .......................................................................................................... 1-16
Loading Values Into Registers .......................................................................................................... 1-17
Load/Store Options ............................................................................................................................. 1-18
Load/Store (.D Unit) ............................................................................................................................ 1-19
Creating a Pointer ............................................................................................................................... 1-20
Incrementing Pointers ......................................................................................................................... 1-21
Another Set of Functional Units - Adding Side B .............................................................................. 1-22
Example Code Review (using side A only) ......................................................................................... 1-23
Conclusion ............................................................................................................................................. 1-23

Instruction Set Overview ..................................................................................................................... 1-24
'C62x Devices ....................................................................................................................................... 1-24
'C671x Devices ..................................................................................................................................... 1-25
'C672x Devices ..................................................................................................................................... 1-25
'C64x Devices ....................................................................................................................................... 1-26
'C64x+ Devices ..................................................................................................................................... 1-27

'C6000 Internal Data Buses .................................................................................................................. 1-29

'C6000 Peripherals ............................................................................................................................... 1-29
Clocking the 'C6000 .............................................................................................................................. 1-29

Two New Devices (C6455, C672x)........................................................................................................... 1-29
Building the ‘C6000 Architecture from Start to Finish

‘C6000 System Diagram

The ‘C6000 processor can be divided into three main parts – CPU (or the processor “core”), memory, and peripherals. This chapter primarily focuses on the development of the CPU, although it also provides an overview of the peripherals.

Let’s look into the CPU...
What Problem Are We Trying To Solve?

It is important to understand the fundamental problem or algorithm for which this architecture was optimized. Armed with this information, it’s easier to understand WHY certain tradeoffs were made in creating this architecture. Before looking at the algorithm, let’s step back and examine a few digital signal processing (DSP) basics.

What is DSP? DSP is the process of sampling analog signals, converting them to their digital approximations and performing a numerical operation to change, modify, or enhance the samples prior to converting them back to the analog world. Digital samples (just a sequence of bits that represent an analog value) are easier to manipulate with software than using analog resistors, capacitors and amplifiers to modify the analog signal. Digital processing is more flexible, deterministic, and robust. Since it isn’t affected by temperature or age it is also more reliable. All these attributes contribute to its increasing popularity.

The figure below demonstrates an incoming analog signal sampled at a specific frequency \( f_s \) – sampling frequency. The signal is converted to digital via an analog-to-digital (A/D) converter which outputs a series of digital (numerical) samples. After these samples are converted, they are sent to the DSP for processing. A typical DSP equation (the actual operation used to manipulate the incoming signal) is also shown:

Most DSP algorithms such as filters, convolution, FFTs, etc., can all be boiled down to a simple sum-of-products equation as shown above. The fundamental components of this algorithm include multiplying, adding, looping, and acquiring new data values. While the ‘C6000 was designed to handle these in real-time — as you’ll witness throughout this course — it is primarily a RISC CPU capable of handling multiple applications and algorithms as it processes at a very high rate of speed.
The fundamental algorithm contains two basic operations – multiply and add. These operations are easy to perform (calculators have done them for years.), but become more difficult when they must execute in real-time. Prior to the 1970s, real-time multiply and accumulate functions required expensive parallel hardware (such as array processors and supercomputers with vector pipelined instruction sets) to keep up with an incoming analog signal. Today, a single DSP can easily handle these performance requirements.

The definition of real-time depends on the application. For example, if the highest frequency of the incoming analog signal is 4KHz (such as speech), the Nyquist theorem says you must sample at greater than 8KHz (2x) to avoid aliasing (corrupting the input sample values). Based on a 10KHz sampling frequency, the time between samples is \( \frac{1}{10K} = 100\mu\text{sec} \). In our example above, \( f_s = 10K\text{Hz} \), therefore

\[
T = 100\mu\text{sec}
\]

If the sample period is 100\(\mu\text{sec} \) and the selected algorithm requires 100 instructions, each instruction must execute in 1\(\mu\text{sec} \) (100\(\mu\text{sec} / 100 = 1\mu\text{sec} = 1000\text{ns} \)) to meet the definition of real-time for this system. By today’s standards, 1000\(\text{ns} \) is very slow – typical DSP cycle times range from 20-50\(\text{ns} \) or even faster. This level of performance increase allows users to:

- Sample higher frequency input signals
- Increase bandwidth to support more features with a single processor
- Combine multiple applications or multiple instances of an application onto one processor (e.g. Instead of one processor being dedicated to a single modem, multiple modems could share one CPU, thus decreasing size and cost.)

The ‘C6000 processor family is capable of performing a maximum of 8 instructions in 1\(\text{ns} \) – delivering up to 8000 MIPs to your application. This raw performance opens new doors and enables you to significantly enhance your current applications and products.

*So, get ready to step inside the world of the ‘C6000 and experience the world’s most advanced DSP architecture … and, don’t forget to fasten your seat belts…*
Page left intentionally blank.
Sum of Products (Example)

Let’s dissect the basic DSP algorithm and look at the pieces one at a time. The key operations are multiply and add. As we begin solving the sum-of-products equation, we’ll develop the architecture step-by-step. Along the way we’ll even introduce a few assembly instructions.

Sum of Products (SOP) - Example

Let's write the code for this algorithm...
And develop the architecture along the way...

What are the two basic instructions required by this algorithm?

- Multiply
- Add

Note: Although the code examples in this module resemble the correct ‘C6000 syntax, they will not work as written. Specific syntax details are omitted in order to focus on architectural concepts. Proper ‘C6000 syntax is described in modules 3 and 12.
Multiply (.M Unit)

Let’s begin by writing the assembly code to solve for Y:

```assembly
MPY a, x, prod ; prod = a * x
```

This instruction reads “multiply a and x together and store the result in prod.” The semicolon “;” denotes comments (actually the correct assembler syntax). Which hardware unit typically performs a multiply?

The multiplier, of course – and on the ’C6000, we call it the .M unit:

![Multiply (.M unit)](image)

All hardware units on the ‘C6000 have one-letter names preceded by a period “.”. We call the multiplier the “M unit” or “dot M unit”. The .M unit accepts the variables “a” and “x” as inputs and outputs the product of the multiply to the variable “prod”. The ’C6000 uses a 16-bit multiplier which provides a 32-bit result.

Is it necessary to specify the unit name in the code syntax? Shouldn’t the tools be smart enough to figure out that the .M unit performs the multiply? Great questions – we’re glad you asked. Actually, the tools do not require the programmer to specify ANY unit names. The tools will choose the proper unit based on the specified instruction and operands. However, we have chosen to specify unit names throughout this workshop for the following reasons: (1) it’s legal; (2) we want to familiarize you with which instructions operate on specific units; (3) when optimizing your code (using parallel instructions), it’s important to know which units are used; (4) it helps document what you’re doing (actually, comments would help, but who has time to write comments?! <grin>). So, we want to develop a good habit now of specifying the unit names which will help us as we introduce new units and instructions.

What’s the next operation?
Add (.L Unit)

The next step is to accumulate the product “prod” into a summing register, which will be our final result “Y”:

\[
\text{ADD } Y, \text{ prod, } Y ; Y = Y + \text{prod}
\]

This instruction reads “add \( Y \) to \( \text{prod} \) and store the result in \( Y \)” In addition to the multiplier, we need another functional unit - an adder. Many processors call this unit the ALU (arithmetic logic unit) and it not only performs adds and subtracts, but also logical operations such as AND, OR and XOR.

Because we like one letter names for the functional units, we can pick from A, L or U. U doesn’t make any sense – A makes the most sense (but as you’ll see later, we use A for something else) – so we called it the .L unit (because it’s logical to do so!). Actually, L can be viewed as denoting logical operations, although this unit supports other instructions as well.

Now that we have performed the multiply and add, we need to determine where the variables are stored and how to access them.
Register File

We need a place to store the variables a, x, prod and Y, such as memory or internal registers. Because we want easy and quick access, we’ll store the variables in a register file connected to the functional units:

Register file A contains sixteen (16) 32-bit registers (named A0-A15). These store the contents of our variables and provide quick access into and out of the functional units. We’ll store “a” in A0; “x” in A1, “prod” in A3; and, “Y” in A4. Based on this, how should the code syntax change?
Specifying Register Names

Our code must change to reflect the use of registers instead of the symbolic variable names. We simply substitute the proper register names in the code (a → A0, x → A1, prod → A3, Y → A4):

\[
Y = \sum_{n=1}^{40} a_n x_n
\]

Later on, we'll see how to keep using symbols, rather than register names (ASM Optimizer allows this):

MPY a, x, prod
ADD sum, prod, sum

For now, let's use registers to get a better feel for the architecture:

MPY .M1 A0, A1, A3
ADD .L1 A4, A3, A4

Most instructions require you to specify the operands as register names. So, how do the variables get loaded into the registers? Another good question – just hold onto it for a few more minutes and we’ll answer it.

Later in the Workshop: Using Symbols with the Assembly Optimizer

Later on, we'll show you how to use symbols (as we have done thus far):

MPY a, x, prod
ADD sum, prod, sum

rather than register names:

MPY .M1 A0, A1, A3
ADD .L1 A4, A3, A4

For now, though, let's use the actual hardware register names as this should help us get a better feel for the C6000’s architecture.

Now that we’ve finished the multiply and add, what’s next? If you look at the equation, you’ll notice that we need to perform the MPY/ADD pair forty times. How do you repeat these instructions forty times? A loop, of course…
Creating a Loop

There are four basic steps in creating a loop. First, decide where to loop (start and end) and use an instruction (branch) to perform the loop. Also needed: a loop counter, a way to decrement the counter, and to stop branching (looping) when the counter reaches zero.

Creating a Loop

1. Add branch instruction (B) and a label
2. Create a loop counter (= 40)
3. Add an instruction to decrement the loop counter
4. Make the branch conditional based on the value in the loop counter

Let’s see how each of these operations is performed on the ‘C6000…
Branching (.S Unit)

First, add a branch instruction to the end of the code. But, where do we branch to? Let’s add a label to the top of our code called “loop” and branch to there.

Which functional unit performs the branch? We have a 3rd unit called .S which performs branches. It might have been called it .B, which appears to make more sense, but B is used elsewhere (you’ll see later). Since the .S unit also performs shift operations, “.S” does make sense.
Creating a Loop Counter (MVK)

The next step creates a loop counter with a value of 40. This is where the CPU will keep track of how many more times to loop. Because the starting point, forty (40) is a constant, use the MVK instruction to MoVe a Konstant into a register (we chose A2):

\[
\text{MVK .S 40, A2 ; A2 } = 40
\]

MVK loads a 16-bit value into the lower half of a 32-bit register. The upper 16 bits can be loaded with another instruction, MVKH, if necessary. Let’s add this new instruction to our code:

Decrementing the Loop Counter

Another instruction decrements the counter register inside the loop (subtracting 1 from A2). We should probably add this instruction before the branch:

The subtract instruction (SUB) also uses the .L unit, similar to the ADD. Actually, add and subtract can be performed on six different units. This provides flexibility when optimizing code.
Conditional Instructions

The branch instruction must execute conditionally based on the value in A2 (if A2 ≠ 0, branch). We can accomplish this by adding \([a2]\) before the branch instruction. In fact, all C6000 instructions are conditional!

<table>
<thead>
<tr>
<th>Code Syntax</th>
<th>Execute instruction if:</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{cond}])</td>
<td>true: (\text{cond} \neq 0)</td>
</tr>
<tr>
<td>([!\text{cond}])</td>
<td>false: (\text{cond} = 0)</td>
</tr>
</tbody>
</table>

Where condition is: \(A0^*, A1, A2, B0, B1, B2\)

By using a register name inside brackets \([\ ]\) on a line of code, the CPU will execute the instruction if evaluated as [TRUE] or [NONZERO].

For example, if we want an instruction to execute if \(A2\neq0\), we add \([A2]\) to the instruction.

Conversely, if we use \([!A2]\), the instruction will execute only if \(A2=0\). If a condition evaluates to false, that specific instruction is not executed and the associated functional unit sits idle for that cycle. You could think of it like the instruction is replaced by a NOP if the condition is false.

The conditional registers are limited to the following:

<table>
<thead>
<tr>
<th>C62x, C67x</th>
<th>A1, A2 B0, B1, B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C64x, C64x+</td>
<td>A0, A1, A2 B0, B1, B2</td>
</tr>
</tbody>
</table>

That is, only the preceding registers may be used as conditional \([\ ]\) operators. As you can see, the ‘C64x allows one additional register \([A0]\). Also, the ‘C64x has an additional decrement and branch instruction that may use any register for a conditional loop counter. This instruction will be discussed later in the workshop.

**Bottom Line Advantage**

Allowing all instructions to be conditional significantly reduces the number of branches (or breaks in program flow) required throughout your code.
Using Conditional Instructions

In our example, branch should only execute when \( A2 \neq 0 \); therefore, we used \([A2]\):

\[
Y = \sum_{n=1}^{40} a_n \cdot x_n
\]

When \( A2 \) reaches zero, the branch doesn’t execute and the code falls out of the loop.

We’ve finished creating the loop by adding the branch instruction, label, counter decrement, and condition.
Loading Values Into Registers

The final question is, how do the variables get loaded into the register file? To answer this question, we must first determine where the variables are located initially, i.e. somewhere in memory (internal or external). If the variables are located in memory, how do we load them into the proper registers?

The first step is to create a pointer to the variable. The pointer contains the address of our variable. Where do we store the pointer’s contents (i.e. the address)? A register, of course.

After creating a pointer, the next step is to use a load instruction (LD) to load the variables “a” and “x” into the proper registers and a store instruction (ST) to store the result to Y.

How do a and x get loaded?

- a, x, Y located in memory
- Create a pointer to values
  - A5 = &a
  - A6 = &x
  - A7 = &Y
- Use pointer with load/store
  - LD *A5, A0
  - LD *A6, A1
  - ST A4, *A7

冷链物流 A

<table>
<thead>
<tr>
<th>Register File A</th>
<th>How do a and x get loaded?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>a</td>
</tr>
<tr>
<td>A1</td>
<td>x</td>
</tr>
<tr>
<td>A2</td>
<td>loop count</td>
</tr>
<tr>
<td>A3</td>
<td>prod</td>
</tr>
<tr>
<td>A4</td>
<td>Y</td>
</tr>
<tr>
<td>A5</td>
<td>&amp;a[n]</td>
</tr>
<tr>
<td>A6</td>
<td>&amp;x[n]</td>
</tr>
<tr>
<td>A7</td>
<td>&amp;Y</td>
</tr>
<tr>
<td>A31</td>
<td>32-bits</td>
</tr>
</tbody>
</table>

Memory

| a[40]          | ← *A5                      |
| x[40]          | ← *A6                      |
| Y              | ← *A7                      |
### Load/Store Options

The 'C6000 actually supports three or four different types of load and store instructions.

You might have already realized that the 'C6000 allows byte addressability; this means you can load bytes (8-bits), halfwords (16-bits) and words (32-bits) depending upon the data type you desire. Therefore, the instruction set includes LDB (load byte), LDH (load half-word), LDW (load word) and the corresponding store instructions:

<table>
<thead>
<tr>
<th>Load instructions</th>
<th>C Data Type</th>
<th>Not Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDB</td>
<td>char</td>
<td></td>
</tr>
<tr>
<td>LDH</td>
<td>short</td>
<td></td>
</tr>
<tr>
<td>LDW</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>LDDW</td>
<td>double, long long</td>
<td>C62x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Store instructions</th>
<th>C Data Type</th>
<th>Not Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>STB</td>
<td>char</td>
<td></td>
</tr>
<tr>
<td>STH</td>
<td>short</td>
<td></td>
</tr>
<tr>
<td>STW</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>STDW</td>
<td>double, long long</td>
<td>C62x, C67x</td>
</tr>
</tbody>
</table>

If we’re multiplying 16-bit numbers, hence loading 16-bit numbers, we should use the LDH instruction.

### Double-Word Access

The ‘C67x includes the LDDW instruction which allows 64-bit loads from on-chip memory. The ‘C67x devices only have a 32-bit wide external memory interface.

The ‘C64x includes both LDDW and STDW instructions. These allow 64-bit load and store instructions.
Load/Store (.D Unit)

Let’s begin by adding the load/store instructions to our code. To complete our code example, we need to load the register file with the variables from memory and store the result “Y” back to memory. Memory could be internal or external.

To facilitate the load/store instructions, we use the 4th and “final” unit – the .D unit – which can be viewed as the data unit, i.e. the unit that loads and stores data. The syntax for the load and store instructions is shown below denoting the .D functional unit.

The register name (such as A5) is combined with an “*” to read “*A5” to tell the CPU to access the data at the address pointed to by A5 (instead of the data contained in A5). This is sometimes called indirect addressing. In our example, using:

```
LDH .D *A5, A0
```

loads A0 with the 16-bit (halfword) value contained at the address pointed to by A5.

The .D unit is the ONLY method of loading/storing data from/to memory. This implies that the ‘C6000 is a load/store RISC architecture. If you want to operate on variables, you must first load them into the register file, perform an operation, and then store them back to memory. Basically, the only “addressing mode” available on this device is via “pointers” which is sometimes called “indirect”. Direct addressing (using symbols or addresses as operands) is not used. Simplifying the methods of acquiring data values from memory makes assembly programming easier and helps the compiler create less-complicated, more efficient code.

Now that we’ve added the LD/ST instructions to our code, how do the pointer values (addresses) get loaded initially?
Creating a Pointer

The first step in using a pointer is to load the address of the specific data you want to access. In our example, we need to load three pointers: one for the “a” array, “x” array and the result, “Y”. What memory element do we use to hold these addresses? A data register. How is the address loaded into a register? An address is constant, so we can use MVK. But if the address of the data value is 32 bits wide but MVK only loads the lower 16 bits – is this a problem? No. We can also use MVKL/MVKH. MVKL performs the same operation as MVK (loads the lower 16-bits with sign-extension). MVKH loads the upper (high) 16 bits of the 32-bit register (and doesn’t affect the lower 16-bits).

<table>
<thead>
<tr>
<th>Loading a Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>How do you load a pointer with an address?</strong></td>
</tr>
<tr>
<td>An address is a constant, so use MVK:</td>
</tr>
<tr>
<td>MVK .S a, A5</td>
</tr>
<tr>
<td>How many bits can MVK move? 16</td>
</tr>
<tr>
<td>How many bits represent a full address? 32</td>
</tr>
<tr>
<td><strong>Solution?</strong></td>
</tr>
<tr>
<td>MVKL .S a, A5</td>
</tr>
<tr>
<td>MVKH .S a, A5</td>
</tr>
</tbody>
</table>

So, to load A5 with the address of “a”, we use:

```
MVKL .S a, A5
MVKH .S a, A5
```

Similar instruction sequences can be used to load the addresses of “x” and “Y” as well. Please note that the order of these instructions is important. If MVKH is used first, MVKL will sign-extend to the upper half of the register and destroy the upper 16 bits. Using **MVKL first**, then MVKH is the proper order.

**Note:** Always use MVKL and MVKH in combination to load values greater than 16-bits. To load constant values less-than or equal to 16-bits, use MVK.
Incrementing Pointers

If we simply leave the code as is, the algorithm will load the same value each time through the loop. How do we write the code so that we access the next value in the array each pass? We must use the notation “++” in the pointer syntax to “bump” or “increment” the pointer each time through the loop. If you’re a C programmer, this method of incrementing values should be quite familiar. (Note, A4 is assumed to have been previously cleared.)

Let’s add this new syntax to our previous code:
Another Set of Functional Units - Adding Side B

We are already familiar with the A side (16 registers with 4 functional units). To effectively double the parallelism and available bandwidth, we have added an identical set of functional units and registers (called the B side.) Because we need to distinguish between the named execution units on each side, the A side unit names change to .S1, .M1, .L1, .D1 and the B side units are named .S2, .M2, .L2 and .D2.

The B side functional units operate identically to the A side units and limited communication between the sides is available:
Example Code Review (using side A only)

Because our code example uses registers from the A side (Side 1) only, we need to modify the unit names. Instead of using “.M” for example, we add a “1” to the unit name to specify that we are using “.M1” instead of “.M2”. Once again, the tools would simply pick the proper functional unit based on the instruction and the operands. However, because of the reasons we noted earlier, we will specify the exact unit name in our code:

\[
Y = \sum_{n=1}^{40} a_n \cdot x_n
\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Example</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVK .S1</td>
<td>40, A2</td>
<td>; A2 = 40, loop count</td>
<td></td>
</tr>
<tr>
<td>loop: LDH .D1</td>
<td>*A5++, A0</td>
<td>; A0 = a(n)</td>
<td></td>
</tr>
<tr>
<td>LDH .D1</td>
<td>*A6++, A1</td>
<td>; A1 = x(n)</td>
<td></td>
</tr>
<tr>
<td>MPY .M1</td>
<td>A0, A1, A3</td>
<td>; A3 = a(n) * x(n)</td>
<td></td>
</tr>
<tr>
<td>ADD .L1</td>
<td>A3, A4, A4</td>
<td>; Y = Y + A3</td>
<td></td>
</tr>
<tr>
<td>SUB .L1</td>
<td>A2, 1, A2</td>
<td>; decrement loop count</td>
<td></td>
</tr>
<tr>
<td>[A2] B .S1</td>
<td>loop</td>
<td>; if A2 ≠ 0, branch</td>
<td></td>
</tr>
<tr>
<td>STH .D1</td>
<td>A4, *A7</td>
<td>; *A7 = Y</td>
<td></td>
</tr>
</tbody>
</table>

Note: Assume A4 previously cleared.

Conclusion

We now have 8 functional units which you can use in parallel to execute up to 8 instructions in a given cycle. If the cycle time for one instruction is 5ns (200 MHz clock), this results in 1600 MIPs of performance! Different devices in the family support various sizes of memory, types of peripherals, etc. The final issues we need to address:

- Instruction set
- Memory map
- Peripherals
- Tools
Instruction Set Overview

‘C62x Devices

We’ve summarized the ‘C6000 instruction set in two ways. First, the instructions are grouped according to their categories, i.e. the types of operations these instructions perform:

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Logical</th>
<th>Data Mgmt</th>
<th>Bit Mgmt</th>
<th>Program Ctrl</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>AND</td>
<td>LDB/H/W</td>
<td>CLR</td>
<td>B</td>
</tr>
<tr>
<td>ADD</td>
<td>CMPEQ</td>
<td>MV</td>
<td>EXT</td>
<td>IDLE</td>
</tr>
<tr>
<td>ADDA</td>
<td>CMPG</td>
<td>MVC</td>
<td>LMBD</td>
<td>NOP</td>
</tr>
<tr>
<td>ADDK</td>
<td>CMPLT</td>
<td>MVK</td>
<td>NORM</td>
<td></td>
</tr>
<tr>
<td>ADD2</td>
<td>NOT</td>
<td>MVKL</td>
<td>SET</td>
<td></td>
</tr>
<tr>
<td>MPY</td>
<td>OR</td>
<td>STB/H/W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPYH</td>
<td>SHL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEG</td>
<td>SHR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMPY</td>
<td>SSHL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMPYH</td>
<td>XOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SADD</td>
<td>SAT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAT</td>
<td>SSUB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>SUB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBB</td>
<td>SUB2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZERO</td>
<td>ZERO</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Refer to the ‘C6000 CPU Reference Guide for more details

The second diagram groups instructions by functional unit. Note that instructions such as ADD and SUB can be performed on six different functional units, providing significant resource flexibility in the architecture.
‘C671x Devices

In addition to the instructions listed above, the ‘C67x CPU includes 31 instructions. These perform single and double precision floating-point math, reciprocals, and extended precision fixed-point math. (These instructions also apply to the ‘C6701.)

‘C67x – Superset of Fixed-Point

.C67x Devices

‘C672x Devices

C67x+ CPU Core Enhancements

CPU Enhancements

- Number of registers doubled to 64
- Cross-path operand sourcing ability doubled to 2
- Execution Packets can now Span Fetch Packets (for better code size!)
- All changes are backwards compatible to 67x CPU

New Instructions

- .S Units enhanced with FP Adder
  - ADDSP
  - ADDDP
  - SUBSP
  - SUBDP
- Along with .L unit, you can have 4 float adds/subtracts in parallel
- .M Units enhanced with mixed precision multiply instructions
  - MPYSPDP – SP x DP into DP
  - MPYSP2DP – SP x SP into DP
- Many apps may benefit from these mixed precision floating point mpy’s
- These provide faster alternatives to the full double precision MPYDP
‘C64x Devices

Similar to the ‘C67x devices, the ‘C64x devices are a superset of the ‘C62x. TI made the ‘C64x architecture announcement in 1Q00. At 4400-8000 MIPS, these will be the fastest members of the ‘C6000 family, yet.

The following block diagram demonstrates the new architectural features. The lighter gray boxes represent the new additions (Advanced Instruction Packing, Advanced Emulation, Registers A16-A31 and B16-B31, plus additional arithmetic/logical operations in each of the functional units.)

As mentioned earlier, the ‘C64x is a superset of the ‘C62x. Therefore, it contains all the instructions of the ‘C62x and adds these additional instructions:
A few of the C64x+ Features

<table>
<thead>
<tr>
<th>New Feature</th>
<th>Benefit</th>
<th>Discussed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compatibility</td>
<td>100% Object Code compatible with C62x/C64x</td>
<td>Throughout</td>
</tr>
<tr>
<td>New Instructions</td>
<td>◆ 8000 16x16 MMAC's</td>
<td>Chapter 7</td>
</tr>
<tr>
<td></td>
<td>◆ 32-bit Integer Multiplies</td>
<td></td>
</tr>
<tr>
<td></td>
<td>◆ Complex Multiples</td>
<td></td>
</tr>
<tr>
<td>SPLOOP Buffer</td>
<td>Decreases code size and lowers power dissipation</td>
<td>Chapters 8/10/15</td>
</tr>
<tr>
<td>Compact Instruct's</td>
<td></td>
<td>Chapter 10</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Support for 124 interrupt events</td>
<td>Chapter 15</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Support for internal and external exceptions</td>
<td>Chapter 15</td>
</tr>
<tr>
<td>DMA Support</td>
<td>◆ IDMA for transfers between internal memories</td>
<td>Not discussed</td>
</tr>
<tr>
<td></td>
<td>◆ EDMA3 improves upon stellar C64x's EDMA2</td>
<td></td>
</tr>
<tr>
<td>64-bit TimeStamp Ctr</td>
<td>Time-Stamp Counter runs at CPU clock rate</td>
<td>Not discussed</td>
</tr>
<tr>
<td>Privilege</td>
<td>Supervisor and User modes</td>
<td>Not discussed</td>
</tr>
<tr>
<td>Internal Memory</td>
<td>◆ Larger sizes supported</td>
<td>Chapter 14</td>
</tr>
<tr>
<td></td>
<td>◆ L1 can be SRAM or Cache</td>
<td></td>
</tr>
<tr>
<td>Memory Protection</td>
<td>Provides support for paged memory protection</td>
<td>Not discussed</td>
</tr>
</tbody>
</table>

Listing the new C64x+ instructions...

New C64x+ Instructions

<table>
<thead>
<tr>
<th>No Unit</th>
<th>.L</th>
<th>.M</th>
<th>.S</th>
</tr>
</thead>
<tbody>
<tr>
<td>DINT</td>
<td>ADDSUB</td>
<td>CMPY</td>
<td>CALLP</td>
</tr>
<tr>
<td>RINT</td>
<td>ADDSUB2</td>
<td>CMPYR</td>
<td>DMV</td>
</tr>
<tr>
<td>SPKERNEL</td>
<td>DPACK2</td>
<td>CMPYR1</td>
<td>RPACK2</td>
</tr>
<tr>
<td>SPKERNELR</td>
<td>DPACKX2</td>
<td>DDOTP4</td>
<td></td>
</tr>
<tr>
<td>SPLOOP</td>
<td>SADDSUB</td>
<td>DDOTPH2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SADDSUB2</td>
<td>DDOTPH2R</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SHFL3</td>
<td>DDOTP2L2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SSUB2</td>
<td>DDOTP2L2R</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GMPY4</td>
<td>SMPY32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPY2IR</td>
<td>XORMPY</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MPY32 (32-bit result)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MPY32 (64-bit result)</td>
</tr>
</tbody>
</table>
‘C674x Devices

This CPU core supports all instructions from all previous generations of fixed and float.

‘C66x Devices

**C66x CPU Core Additions**

<table>
<thead>
<tr>
<th>New Feature</th>
<th>Benefit</th>
</tr>
</thead>
</table>
| Compatibility | ◆ 100% Binary Code compatible with all previous gen’s  
◆ Original float instr’s remain for compatibility (e.g. ADDSP)  
◆ New instr’s created for optimized floating-pt (e.g. FADDSP) |

<table>
<thead>
<tr>
<th>4x the Multiplies</th>
<th># Per Cycle</th>
<th>Precision</th>
<th>MMACS (@1.25GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>32-bit IEEE (float)</td>
<td>10,000</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>32-bit fixed  (int)</td>
<td>10,000</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>16-bit fixed  (short)</td>
<td>40,000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Improved Floating-Point</th>
<th>Benefit</th>
</tr>
</thead>
</table>
|                         | ◆ Up to 16 floating-point operations (up from 6)  
◆ Added floating-point support for complex math |

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C674x</td>
<td>32b + 8b</td>
<td>64b</td>
<td>32b</td>
<td>128b</td>
</tr>
<tr>
<td>C66x</td>
<td>32b + 8b</td>
<td>64b</td>
<td>32b</td>
<td>128b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Crosspaths</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>C674x</td>
<td>One 32-bit operand in each direction; use for 1-2 units</td>
</tr>
<tr>
<td>C66x</td>
<td>One 32- or 64-bit bus in each dir; can use for all 4 units</td>
</tr>
</tbody>
</table>

**New C66x Instructions**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD (SIMD)</td>
<td>DADD</td>
<td>DADD2</td>
</tr>
<tr>
<td>DADDSP</td>
<td>DSADD</td>
<td></td>
</tr>
<tr>
<td>DSADD2</td>
<td>DSSUB</td>
<td></td>
</tr>
<tr>
<td>DSSUB2</td>
<td>DSUB</td>
<td></td>
</tr>
<tr>
<td>DSSUB2</td>
<td>DSSUB2</td>
<td></td>
</tr>
<tr>
<td>DSSUB2</td>
<td>DSUB2</td>
<td></td>
</tr>
<tr>
<td>DSUB2</td>
<td>DSSUB2</td>
<td></td>
</tr>
<tr>
<td>Compare (SIMD)</td>
<td>DCMPEQ2</td>
<td>DCMPEQ4</td>
</tr>
<tr>
<td>DCMPEQ2</td>
<td>DCMPEQ4</td>
<td></td>
</tr>
<tr>
<td>DCMPEQ4</td>
<td>DCMPEQ4</td>
<td></td>
</tr>
<tr>
<td>DCMPEQ4</td>
<td>DCMPEQ4</td>
<td></td>
</tr>
<tr>
<td>Shifts</td>
<td>DSHL</td>
<td></td>
</tr>
<tr>
<td>DSHL</td>
<td>DSHL2</td>
<td></td>
</tr>
<tr>
<td>DSHL2</td>
<td>DSHR</td>
<td></td>
</tr>
<tr>
<td>DSHR</td>
<td>DSHR2</td>
<td></td>
</tr>
<tr>
<td>DSHR2</td>
<td>DSHR2</td>
<td></td>
</tr>
<tr>
<td>DSHR2</td>
<td>DSHR2</td>
<td></td>
</tr>
<tr>
<td>DSHR2</td>
<td>DSHR2</td>
<td></td>
</tr>
<tr>
<td>DSHR2</td>
<td>DSHR2</td>
<td></td>
</tr>
<tr>
<td>DSHR2</td>
<td>DSHR2</td>
<td></td>
</tr>
<tr>
<td>Complex Rotate</td>
<td>CROT270.L</td>
<td>CROT90.L</td>
</tr>
<tr>
<td>CROT90.L</td>
<td>DCR0T270.L</td>
<td></td>
</tr>
<tr>
<td>DCR0T90.L</td>
<td>Max/Min</td>
<td></td>
</tr>
<tr>
<td>DMAX2.L</td>
<td>DMAXU4.L</td>
<td></td>
</tr>
<tr>
<td>DMAXU4.L</td>
<td>DMIN2.L</td>
<td></td>
</tr>
<tr>
<td>DMIN2.L</td>
<td>DMINU4.L</td>
<td></td>
</tr>
<tr>
<td>DMINU4.L</td>
<td>Pack/Unpack</td>
<td></td>
</tr>
<tr>
<td>Pack/Unpack</td>
<td>DPACKH2</td>
<td></td>
</tr>
<tr>
<td>DPACKH2</td>
<td>DPACKH4</td>
<td></td>
</tr>
<tr>
<td>DPACKH4</td>
<td>DPACKHL2</td>
<td></td>
</tr>
<tr>
<td>DPACKHL2</td>
<td>DPACKL4</td>
<td></td>
</tr>
<tr>
<td>DPACKL4</td>
<td>DPACKLR4</td>
<td></td>
</tr>
<tr>
<td>DPACKLR4</td>
<td>DSPACEKU4 .S</td>
<td></td>
</tr>
<tr>
<td>DSPACKE4 .S</td>
<td>UNPKBU4</td>
<td></td>
</tr>
<tr>
<td>UNPKBU4</td>
<td>UNPKH2</td>
<td></td>
</tr>
<tr>
<td>UNPKH2</td>
<td>UNPKH2</td>
<td></td>
</tr>
<tr>
<td>UNPKH2</td>
<td>UNPKH2</td>
<td></td>
</tr>
<tr>
<td>Fast Float</td>
<td>FADDDP</td>
<td>FADDSP</td>
</tr>
<tr>
<td>FADDSP</td>
<td>FMPSADP</td>
<td></td>
</tr>
<tr>
<td>FMPSADP</td>
<td>FSUBDP</td>
<td></td>
</tr>
<tr>
<td>FSUBDP</td>
<td>FSUBSP</td>
<td></td>
</tr>
<tr>
<td>FSUBSP</td>
<td>Misc</td>
<td></td>
</tr>
<tr>
<td>DAPS32.L</td>
<td>DMV: .L/ .S</td>
<td></td>
</tr>
<tr>
<td>DXPND2 .M</td>
<td>DXPND4 .M</td>
<td></td>
</tr>
<tr>
<td>DXPND4 .M</td>
<td>MFENCE</td>
<td></td>
</tr>
<tr>
<td>MFENCE</td>
<td>INT &gt; SP, SP &gt; INT</td>
<td></td>
</tr>
<tr>
<td>INT &gt; SP, SP &gt; INT</td>
<td>DINTHSBC</td>
<td></td>
</tr>
<tr>
<td>DINTHSBC</td>
<td>DINTSPU</td>
<td></td>
</tr>
<tr>
<td>DINTSPU</td>
<td>DSPINT</td>
<td></td>
</tr>
<tr>
<td>DSPINT</td>
<td>DSPINT</td>
<td></td>
</tr>
<tr>
<td>DSPINT</td>
<td>Logical</td>
<td></td>
</tr>
<tr>
<td>LAND .L</td>
<td>LANDN .L</td>
<td></td>
</tr>
<tr>
<td>LANDN .L</td>
<td>LOR .L</td>
<td></td>
</tr>
<tr>
<td>LOR .L</td>
<td>Average</td>
<td></td>
</tr>
<tr>
<td>DAVG2 .M</td>
<td>DAVGNR2 .M</td>
<td></td>
</tr>
<tr>
<td>DAVGNR2 .M</td>
<td>DAVGNR4 .M</td>
<td></td>
</tr>
<tr>
<td>DAVGU4 .M</td>
<td>Matrix Multiply</td>
<td></td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>CCMATMPY</td>
<td>CCMATMPY</td>
</tr>
<tr>
<td>CCMATMPY</td>
<td>CCMATMPY</td>
<td></td>
</tr>
<tr>
<td>CCMATMPY</td>
<td>CCMATMPY</td>
<td></td>
</tr>
<tr>
<td>CCMATMPY</td>
<td>CCMATMPY</td>
<td></td>
</tr>
<tr>
<td>CCMATMPY</td>
<td>CCMATMPY</td>
<td></td>
</tr>
<tr>
<td>CCMATMPY</td>
<td>CCMATMPY</td>
<td></td>
</tr>
<tr>
<td>CCMATMPY</td>
<td>CCMATMPY</td>
<td></td>
</tr>
<tr>
<td>CCMATMPY</td>
<td>CCMATMPY</td>
<td></td>
</tr>
<tr>
<td>CCMATMPY</td>
<td>CCMATMPY</td>
<td></td>
</tr>
<tr>
<td>CCMATMPY</td>
<td>Double MPY</td>
<td></td>
</tr>
<tr>
<td>Double MPY</td>
<td>DMPY2</td>
<td></td>
</tr>
<tr>
<td>DMPY2</td>
<td>DMPY8</td>
<td></td>
</tr>
<tr>
<td>DMPY8</td>
<td>DMPY4U</td>
<td></td>
</tr>
<tr>
<td>DMPY4U</td>
<td>DMPYU2</td>
<td></td>
</tr>
<tr>
<td>DMPYU2</td>
<td>Quad MPY</td>
<td></td>
</tr>
<tr>
<td>Quad MPY</td>
<td>QMPY32</td>
<td></td>
</tr>
<tr>
<td>QMPY32</td>
<td>QMPYSP</td>
<td></td>
</tr>
<tr>
<td>QMPYSP</td>
<td>QMPY32R1</td>
<td></td>
</tr>
<tr>
<td>QMPY32R1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Internal Data Buses & Memory

Internal Buses (to/from the CPU core)

Shown below is part of the internal bus structure of the ‘C6000 family. As you can see, the CPU is able to perform three simultaneous bus operations: program-read and two data-read/writes. This bus architecture provides flexibility required to achieve high throughput.

Notice that the later C6000 family members have been modified to allow 64-bit data loads. This allows the ‘C67x CPU to perform a single-cycle load of a double-precision floating point operand. The wider buses allow the C64x devices to access up to 16 packed-bytes per load/store, thus enhancing its ability to perform packed data processing.

Note: Another term often used along with packed data processing is SIMD – single-instruction, multiple-data. With instructions like LDDW, STDW, and a number of others to be explored throughout this workshop, you will easily the C64x performing these types of parallel operations.
**Internal Memories**

'C6000 System Block Diagram

- **Level 1 Program Memory (L1P)**
  - Single-Cycle
  - Cache / RAM
  - 256

- **Level 1 Data Memory (L1D)**
  - Single-Cycle
  - Cache / RAM
  - 64-bit

- **Level 2 Memory (L2)**
  - Program / Data
  - Cache / RAM

- **On-Chip Peripherals**
  - EMAC
  - SPI
  - I2C
  - PCIe
  - EMAC
  - Async EMIF

- **C6x MegaModule or C66x CorePac**

- **Reg A**
- **Reg B**

- **MegaModule/CorePac includes:**
  - CPU
  - L1 RAMs
  - L2 RAM
  - and a few more things

- **Some devices have add'l slower L3 RAM**

- **External memories**
  - DDR2/3, SDRAM, Async
Switched Central Resource (TeraNet)

'C6000 System Block Diagram

Switched Central Resource (SCR)
- Central crossbar switch
  - From: CPU and Master Peripherals (e.g. EMAC, USB, PCIe, etc.)
  - To: Slave peripherals (e.g. SPI, I2C, McBSP, etc.)
- Renamed “TeraNet” for new C66x devices
- Devices prior to C64x+
  - Have only a single bus rather than a crossbar/SCR
  - All memory transactions are handled by the EDMA controller

Let's look at an example device...

DM644x Architecture Example

Master Devices:
- Can initiate data transfers
Slave Devices:
- Only Sink/Source for data transfers
- Can not initiate data transfers
- Often, they send an interrupt to request data transfer by CPU or EDMA3
C6000 Peripherals

We'll just look at two: EDMA3 and PRU…
EDMA3

Multiple DMA’s: EDMA3 and QDMA

<table>
<thead>
<tr>
<th>VPSS</th>
<th>EDMA3 (System DMA)</th>
<th>C64x+ DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master Periph</td>
<td>DMA (sync)</td>
<td>L1P</td>
</tr>
<tr>
<td></td>
<td>QDMA (async)</td>
<td>L1D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L2</td>
</tr>
</tbody>
</table>

- EDMA3
  - Enhanced DMA (version 3)
  - DMA to/from peripherals
  - Can be sync’d to peripheral events
  - Handles up to 64 events

- QDMA
  - Quick DMA
  - DMA between memory
  - Async – must be started by CPU
  - 4-8 channels available

Both Share (number depends upon specific device)
- 128-256 Parameter RAM sets (PARAMs)
- 64 transfer complete flags
- 2-4 Pending transfer queues

Notes:
- Both ARM and DSP can access the EDMA3
- Only DSP can access hardware IDMA

Multiple DMA’s: Master Periphs & C64x+ IDMA

<table>
<thead>
<tr>
<th>VPSS</th>
<th>EDMA3 (System DMA)</th>
<th>C64x+ DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master Periph’s</td>
<td>DMA (sync)</td>
<td>L1P</td>
</tr>
<tr>
<td></td>
<td>QDMA (async)</td>
<td>L1D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IDMA</td>
</tr>
</tbody>
</table>

- Master Peripherals
  - VPSS (and other master periph’s) include their own DMA functionality
  - USB, ATA, Ethernet, VLYNQ share bus access to SCR

- IDMA
  - Built into all C64x+ DSPs
  - Performs moves between internal memory blocks and/or config bus
  - Don’t confuse with iDMA API (ch 14)
PRU

Programmable Realtime Unit (PRU)

PRU consists of:
- 2 Independent, Realtime RISC Cores
- Access to pins (GPIO)
- Its own interrupt controller
- Access to memory (master via SCR)
- Device power mgmt control
  (ARM/DSP clock gating)

- Use as a soft peripheral to implement add’l on-chip peripherals
- Examples implementations include:
  - Soft UART
  - Soft CAN
- Create custom peripherals or setup non-linear DMA moves.
- Implement smart power controller:
  - Allows switching off both ARM and DSP clocks
  - Maximize power down time by evaluating system events before waking up DSP and/or ARM

PRU consists of:
- 2 Independent, Realtime RISC Cores
- Access to pins (GPIO)
- Its own interrupt controller
- Access to memory (master via SCR)
- Device power mgmt control
  (ARM/DSP clock gating)

PRU SubSystem : IS / IS-NOT

<table>
<thead>
<tr>
<th>Is</th>
<th>Is-Not</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual 32-bit RISC processor specifically designed for manipulation of packed memory mapped data structures and implementing system features that have tight real time constraints.</td>
<td>Is not a HW accelerator used to speed up algorithm computations.</td>
</tr>
</tbody>
</table>
| Simple RISC ISA:  
  - Approximately 40 instructions  
  - Logical, arithmetic, and flow control ops all complete in a single cycle | Is not a general purpose RISC processor:  
  - No multiply hardware/instructions  
  - No cache or pipeline  
  - No C programming |
| Simple tooling:  
  Basic command-line assembler/linker | Is not integrated with CCS. Doesn't include advanced debug options |
| Includes example code to demonstrate various features. Examples can be used as building blocks. | No Operating System or high-level application software stack |
Pin Multiplexing

What is Pin Multiplexing?

- How many pins is on your device?
- How many pins would all your peripheral require?
- Pin Multiplexing is the answer – only so many peripherals can be used at the same time … in other words, to reduce costs, peripherals must share available pins
- Which ones can you use simultaneously?
  - Designers examine app use cases when deciding best muxing layout
  - Read datasheet for final authority on how pins are muxed
  - Graphical utility can assist with figuring out pin-muxing...

Pin Mux Example

Graphical Utility can assist with figuring out pin-muxing...

Pin Muxing Tools

- Graphical Utilities For Determining which Peripherals can be Used Simultaneously
- Provides Pin Mux Register Configurations
Device Family Review

Devices Overview

DSP Generations: DSP and ARM+DSP

<table>
<thead>
<tr>
<th>Fixed-Point Cores</th>
<th>Float-Point Cores</th>
<th>DSP</th>
<th>ARM+DSP</th>
<th>ARM+DSP+Video</th>
</tr>
</thead>
<tbody>
<tr>
<td>C62x</td>
<td>C67x</td>
<td>C620x, C670x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C621x</td>
<td>C67x</td>
<td>C6211, C671x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C64x</td>
<td>C641x</td>
<td>DM642</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C64x+</td>
<td></td>
<td>C672x</td>
<td></td>
<td>DM64xx, OMAP35x, DM37x</td>
</tr>
<tr>
<td>C64x+</td>
<td>DM643x</td>
<td>C647x</td>
<td></td>
<td>OMAP-L138* C6A8168</td>
</tr>
<tr>
<td>C674x</td>
<td>C6748</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C66x</td>
<td>Future</td>
<td>C6670 C667x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Looking at devices from a pin-for-pin perspective...

Processor Portfolio Scalability

<table>
<thead>
<tr>
<th>ARM Only</th>
<th>ARM + DSP</th>
<th>DSP Only</th>
<th>ARM + DSP + Video</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM17x</td>
<td>OMAPL137</td>
<td>C6743/5/7</td>
<td>DM365 / 55 ARM+DSP</td>
</tr>
<tr>
<td>AM18x</td>
<td>OMAPL138</td>
<td>C6472/6/8</td>
<td>DM644x ARM+DSP+Video</td>
</tr>
<tr>
<td>OMAP3503/15</td>
<td>Cortex-A8</td>
<td>C6472/6/8</td>
<td>DM365 / 55 ARM+DSP</td>
</tr>
<tr>
<td>AM3715/03</td>
<td>Cortex-A8</td>
<td>C6472/6/8</td>
<td>DM3730/25 ARM+DSP+Video</td>
</tr>
<tr>
<td>AM389x</td>
<td>Cortex-A8</td>
<td>C6A816x</td>
<td>DM3730/25 ARM+DSP+Video</td>
</tr>
<tr>
<td>AM389x</td>
<td>Cortex-A8</td>
<td>C6A814x</td>
<td>DM3730/25 ARM+DSP+Video</td>
</tr>
</tbody>
</table>

Note: Only showing devices with pin-for-pin compatibility.
Sample Devices

**TMS320C6748**

- **JTAG Interface**
  - PLL/Clk Generator w/OSC
  - General-Purpose Timer (x3)
  - RTC/32.768kHz OSC
  - Power/Sleep Controller
  - GPIO

- **COP Subsystem**
  - C674x™
  - DSP CPU
  - AET
  - 32KB L1 Pgm
  - 32KB L1 RAM
  - 256KB L2 RAM
  - BOOT ROM

- **Switched Central Resource (SCR)**

**OMAP-L138 (ARM9 + C6748)**

- **JTAG Interface**
  - PLL/Clk Generator w/OSC
  - General-Purpose Timer (x3)
  - RTC/32.768kHz OSC
  - Power/Sleep Controller
  - GPIO

- **ARM Subsystem**
  - ARM926EJ-S CPU
  - With MMU
  - 4KB ETB
  - 16KB I-Cache
  - 16KB D-Cache
  - 64KB RAM
  - (Vector Table)
  - 64KB ROM

- **C674x™**
  - DSP CPU
  - AET
  - 32KB L1 Pgm
  - 32KB L1 RAM
  - 256KB L2 RAM
  - BOOT ROM

- **Switched Central Resource (SCR)**

**Peripherals**
- **DMA**
  - EDMA3 (x3)
  - MMABSP (x2)
  - EDMA2 w/FIFO
  - USB 2.0 OTG Ctrl PHY
  - USB 1.1 OHCI Ctrl PHY

- **Audio Ports**
  - CPU (x2)
  - SPI (x3)
  - UART (x3)
  - LCD Ctrl
  - uPP
  - SATA
  - VPIF

- **Serial Interfaces**
  - EMAC 10/100 (MII/RMMI)
  - MMC/SD (x2)

- **Display**
  - LCD Ctrl

- **Video**
  - VPIF

- **Parallel Port**
  - uPP

- **Internal Memory**
  - 128KB RAM

- **Customizable Interface**
  - PRU Subsystem

- **Connectivity**
  - USB 2.0 OTG Ctrl PHY
  - USB 1.1 OHCI Ctrl PHY
  - EMAC 10/100 (MII/RMMI)
  - MMC/SD (x2)
  - SATA
  - VPIF

- **External Memory Interfaces**
  - ENIF/EMIF/168
  - NAND/Flash 16MB SRAM
  - DDR3/DDR2 Controller

**Control Timers**
- **ePWM (x2)
- eCAP (x3)**
Choosing a Device – TI Web Tool

DSP & ARM MPU Selection Tool

Using CCS with C Programs

Introduction

The importance of the C language has grown significantly over the past few years. TI has responded by creating a compiler that produces extremely efficient processor code, which is so speed efficient you may not need to program in assembly. Thus, we begin discussing 'C6000 coding with the C Compiler.

All it takes is a couple minutes to get your C code running on the 'C6000. That's the goal of this module. First you'll compile a C dot-product routine, and then debug and benchmark it.

Outline

- C6000 Programming Methods
- Code Composer Studio (CCS)
- Projects
- Where To Go for More C Information
- Lab Exercise
- Lab Debrief (+ Optional Topics)
Chapter Topics

Using CCS with C Programs

Programming Methods for the ‘C6000

Code Composer Studio

A Closer Look

Code Composer Projects

CCS Project Options

Using the Configuration Tool (.TCF files)

Going further with C

Lab 2 – Using C programs in CCS

CCS Automation (After the Lab)

Command Window

GEL Scripting

CCS Scripting / Debug Server Scripting (DSS)

BIOS Textual Configuration (TCF)
Programming Methods for the ‘C6000

Texas Instruments offers three methods for programming the ‘C6000 series of DSP microprocessors: C, Linear Assembly, and standard assembly language. While C and assembly are common among processors, Linear Assembly is something new.

<table>
<thead>
<tr>
<th>Source</th>
<th>Efficiency*</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>80 - 100%</td>
<td>Low</td>
</tr>
<tr>
<td>C ++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear ASM</td>
<td>95 - 100%</td>
<td>Med</td>
</tr>
<tr>
<td>ASM</td>
<td>100%</td>
<td>High</td>
</tr>
</tbody>
</table>

* Typical C62x efficiency vs. hand optimized assembly

As described in the introduction, C is –by far– the most popular method of programming the ‘C6000 family of devices. The ‘C6000 processor was designed with C code in mind. In fact, its architecture was designed concurrently with its C compiler. This provided a rapid prototyping design environment and afforded effective architectural decisions.

Unlike most real-time embedded DSPs, the efficiency of the C compiler, combined with the raw high-performance of the ‘C6000, makes for an incredible combination. The goal? …to achieve supercomputing performance with maximum ease-of-use.

OK, so this sounds like marketing stuff (and it is), but it’s also true. C is the predominant language for ‘C6000 programming. The efficiency enumerated above has been demonstrated across a series of DSP-centric benchmarks. In fact, in the first couple lab exercises, we’ll see performance in the 100% range.
Here are some additional benchmarks you can view online:

### Sample Compiler Benchmarks

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Used In</th>
<th>Asm Cycles</th>
<th>Assembly Time (µs)</th>
<th>C Cycles (Rel 4.0)</th>
<th>C Time (µs)</th>
<th>% Efficiency vs Hand Coded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Mean Square Error MSE of a 20 column</td>
<td>For motion compensation of</td>
<td>348</td>
<td>1.16</td>
<td>402</td>
<td>1.34</td>
<td>87%</td>
</tr>
<tr>
<td>image matrix</td>
<td>image data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Codebook Search</td>
<td>CELP based voice coders</td>
<td>977</td>
<td>3.26</td>
<td>961</td>
<td>3.20</td>
<td>100%</td>
</tr>
<tr>
<td>Vector Max 40 element input vector</td>
<td>Search Algorithms</td>
<td>61</td>
<td>0.20</td>
<td>59</td>
<td>0.20</td>
<td>100%</td>
</tr>
<tr>
<td>All-zero FIR Filter 40 samples, 10 coefficients</td>
<td>VSELP based voice coders</td>
<td>238</td>
<td>0.79</td>
<td>280</td>
<td>0.93</td>
<td>85%</td>
</tr>
<tr>
<td>Minimum Error Search Table Size = 2304</td>
<td>Search Algorithms</td>
<td>1185</td>
<td>3.95</td>
<td>1318</td>
<td>4.39</td>
<td>90%</td>
</tr>
<tr>
<td>IR Filter 16 coefficients</td>
<td>Filter</td>
<td>43</td>
<td>0.14</td>
<td>38</td>
<td>0.13</td>
<td>100%</td>
</tr>
<tr>
<td>IR – cascaded biquads 10 Cascaded biquads</td>
<td>Filter</td>
<td>70</td>
<td>0.23</td>
<td>75</td>
<td>0.25</td>
<td>93%</td>
</tr>
<tr>
<td>(Direct Form II)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC Two 40 sample vectors</td>
<td>VSELP based voice coders</td>
<td>61</td>
<td>0.20</td>
<td>58</td>
<td>0.19</td>
<td>100%</td>
</tr>
<tr>
<td>Vector Sum Two 44 sample vectors</td>
<td></td>
<td>51</td>
<td>0.17</td>
<td>47</td>
<td>0.16</td>
<td>100%</td>
</tr>
<tr>
<td>MSE MSE between two 256 element vectors</td>
<td>Mean Sq. Error Computation</td>
<td>279</td>
<td>0.93</td>
<td>274</td>
<td>0.91</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>in Vector Quantizer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TI C62x™ Compiler Performance Release 4.0: Execution Time in µs @ 300 MHz
Versus hand-coded assembly based on cycle count

So if C is so good, how come we offer two other programming methodologies?

As with all microprocessors, CPU machine instructions are represented by assembly mnemonic syntax. Compilers must transform high-level syntax to these low-level assembly instructions. In some cases your knowledge of the real intent of your system may help create higher performance code. To this end, TI has enabled many ways to provide additional information to the compiler (discussed later in the workshop) to automate this process.

Occasionally, though, you may want to write a few key functions directly in assembly. While you may program in standard assembly language, it is done rarely. Rather, when you want to write directly at the assembly level, Linear Assembly is a better option. Linear Assembly is a variation of standard assembly. It provides access to the same mnemonic instructions but, since code passes through an Assembly Optimizer, Linear Assembly provides three important benefits:

- Instead of defining and specifying specific register usage, the Assembly Optimizer can provide register assignment and optimization for you.
- The tedious chore of setting up argument passing from subroutine to subroutine, or C function to assembly subroutine is handled automatically for you.
- The ‘C6000, like most RISC processors, provides a simple, fast instruction set. Likewise, it’s also a pipelined processor that requires management of instruction latencies. With the Assembly Optimizer you are freed from worrying about these issues. While the “rules” to follow are simple, we thought, ‘Why can’t we handle these issues for you?’.
The Code Composer Studio (CCS) application provides all the necessary software tools for DSP development. At the heart of CCS you’ll find the original Code Composer IDE (integrated development environment). The IDE provides a single application window in which you can perform all your code development; from entering and editing your program code, to compilation and building an executable file, and finally, to debugging your program code.

When TI developed Code Composer Studio, it added a number of capabilities to the environment. First of all, the code generation tools (compiler, assembler, and linker) were added so that you wouldn’t have to purchase them separately. Secondly, the simulator was included (only in the full version of CCS, though). Third, TI has included DSP/BIOS. DSP/BIOS is a real-time kernel consisting of three main features: a real-time, pre-emptive scheduler; real-time capture and analysis; and finally, real-time I/O.

Finally, CCS has been built around an extensible software architecture which allows third-parties to build new functionality via plug-ins. See the TI website for a listing of 3rd parties already developing for CCS. At some point in the future, this capability may be extended to all users. If you have an interest, please voice your opinion by calling the TI SC Product Information Center (you can find their phone number and email address in last module, “What Next?”).
Here’s a snapshot of the CCSv3 screen:

Since it’s hard to evaluate a tool by looking at a simple screen capture, we’ll provide you with plenty of hands-on-experience throughout the week.
**CCS Licensing**

### CCS Pricing Summary

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platinum Eval Tools</td>
<td>Time limited full tools</td>
<td>FREE</td>
</tr>
<tr>
<td>Platinum Bundle</td>
<td>EVM/DSK, sim, XDS100 use</td>
<td>FREE</td>
</tr>
<tr>
<td>Application Development</td>
<td>Linux SDK &amp; Android Dev kit</td>
<td>FREE</td>
</tr>
<tr>
<td>Platinum Node Locked</td>
<td>Full tools tied to a machine</td>
<td>$1995</td>
</tr>
<tr>
<td>Platinum Floating*</td>
<td>Full tools shared across machines</td>
<td>$2995</td>
</tr>
<tr>
<td>Microcontroller Core</td>
<td>MSP/C2000 code size limited</td>
<td>FREE</td>
</tr>
<tr>
<td>Microcontroller Node Locked</td>
<td>MSP/C2000/Stellaris/Cortex R4</td>
<td>$495</td>
</tr>
<tr>
<td>Microcontroller Floating*</td>
<td>MSP/C2000/Stellaris/Cortex R4</td>
<td>$795</td>
</tr>
</tbody>
</table>

- CCSv3.3 pricing is different: $3495 for Platinum, $495 for C2000, node locked only
- CCSv4 licenses can be used with the early versions of CCSv5
- Subscriptions run approximately 20% of original licenses

---

**Emulators (XDS)**

### C6000 Extended Development System (XDS)

- **Scan-based emulation (JTAG)**
- Works with Code Composer Studio
- Uses 14-pin header connector
- Supports all TI’s ARM and DSP dev’s
- Generations are backward compatible
- Each gen increases transfer rates

1. **XDS100**
   - USB bus powered, no power supply
   - Inexpensive (prices start at $89)

2. **XDS510**
   - Non-intrusive scan-based emulation
   - USB bus powered, no power supply
   - As low as $249

3. **XDS560**
   - USB or Ethernet connection to PC
   - Also supports advanced trace connector
   - Advanced emulation trace features found on high-end & multi-core C6x devices
   - Starting at $1495

A Closer Look

A Short Review of CCS File Extensions

Using Code Composer Studio (CCS) you may not need to know all these file extension names, but we included a basic review of them for your reference:

- C and C++ use the standard .C and .CPP file extensions.
- Linear Assembly is written in a .SA file.
- You can either write standard assembly directly, or it can be created by the compiler and Assembly Optimizer. In all cases, standard assembly uses .ASM.
- Object files (.OBJ), created by the assembler, are linked together to create the DSP’s executable output (.OUT) file. The map (.MAP) file is an output report of the linker.
- The .OUT file can be loaded into your system by the debugger portion of CCS.

If you want to use your own extensions for file names, they can be redefined with code generation tool options. Please refer to the TMS320C6000 Assembly Tools Users Guide for the appropriate options.
Code Composer Projects

Code Composer works within a project paradigm. If you’ve done code development with most any sophisticated IDE (Microsoft, Borland, etc.), you’ve no doubt run across the concept of projects.

Essentially, within CCS you create a project for each executable program you wish to create. Projects store all the information required to build the executable. For example, it lists things like: the source files, the header files, the target system’s memory-map, and program build options.
The project information is stored in a .PJT file which is created and maintained by CCS. To create a new project, you need to select the menu. This is different from Microsoft’s Designers Studio as they provide project new/open commands on the File menu.

Along with the main menu, you can also manage open projects using the right-click popup menu. Either of these menus allows you to Add Files… to a project. Of course, you can also drag-n-drop files onto the project from Windows Explorer.

There are many other project management options. In the preceding graphic we’ve listed a few of the most commonly used actions:

- If your project team builds code outside the CCS environment, you may find Export Makefile (and/or Source Control) useful.
- CCS now allows you to keep multiple projects open simultaneously. Use the Set as Active Project menu option or the project drop-down to choose which one is active.
- If you like digging below the surface, you’ll find that the .PJT file is simply an ASCII text file. Open for Editing opens this file within the CCS text editor.
- Configurations… and Options… are covered in detail, next.
CCS Project Options

Project options direct the code generation tools (i.e. compiler, assembler, linker) to create code according to your system’s needs. Do you need to logically debug your system, improve performance, and minimize code size? Your results can be dramatically affected by options on the C6000 platform.

There are probably about a 100 options available for the compiler alone. Usually, this is a bit intimidating to wade through. To that end, we’ve provided a condensed set of options. These few options cover about 80% of most users needs.

### Compiler Build Options

- Nearly one-hundred compiler options available to tune your code’s performance, size, etc.
- Following table lists most commonly used options:

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-mv6700</td>
<td>Generate 'C67x code ('C62x is default)</td>
</tr>
<tr>
<td>-mv6400</td>
<td>Generate 'C64x code</td>
</tr>
<tr>
<td>-mv6400+</td>
<td>Generate 'C64x+ code</td>
</tr>
<tr>
<td>-mv6740</td>
<td>Generate 'C674x code</td>
</tr>
<tr>
<td>-mv6600</td>
<td>Generate 'C66x code</td>
</tr>
<tr>
<td>-fr &lt;dir&gt;</td>
<td>Directory for object/output files</td>
</tr>
<tr>
<td>-fs &lt;dir&gt;</td>
<td>Directory for assembly files</td>
</tr>
<tr>
<td>-g</td>
<td>Enables src-level symbolic debugging</td>
</tr>
<tr>
<td>-ss</td>
<td>Interlist C statements into assembly listing</td>
</tr>
<tr>
<td>-o3</td>
<td>Invoke optimizer (-o0, -o1, -o2/-o, -o3)</td>
</tr>
<tr>
<td>-k</td>
<td>Keep asm files, but don't interlist</td>
</tr>
</tbody>
</table>

As you probably learned in college programming courses, you should probably follow a two step process when creating code:

1. Write your code and debug its logical correctness (without optimization).
2. Next, optimize your code and verify it still performs as expected.

As demonstrated above, certain options are ideal for debugging, but others work best to create highly optimized code. When you create a new project, CCS creates two sets of build options – called **Configurations**: one called **Debug**, the other **Release** (you might think of as Optimize). Configurations will be explored in the next section.

**Note:** Like any compiler or toolset, learning the various options requires a bit of experimentation, but it pays off in the tremendous performance gains that can be achieved by the compiler. To this end, this workshop will explore these options further in an upcoming chapter.
Build Configurations  (Sets of Build Options)

To help make sense of the many compiler and linker options, you can create sets of build options. These sets of options are called configurations. TI provides two default configurations in each new project you create. For example, if you created a project, it would contain:

- **Debug**
  - `-g -fr"$(Proj_dir)\Debug" -d"_DEBUG" -mv6700`

- **Release**
  - `-o3 -fr"$(Proj_dir)\Release" -mv6700`

The two main differences between the *Debug* and *Release* configurations:
- **Debug** uses the –g option to enable source-level debugging
- **Release** invokes the optimizer with –o3 (and doesn’t use –g)

**Note:** `$(Proj_dir)` indicates the current project directory. This aids in project portability. See SPRA913 (*Portable CCS Projects*) for more information.

The following graphic summarizes the default configurations for a project. Additionally, it shows how to:
- Select the configuration before building your project
- Add or Remove configurations from a project (*Project → Configurations...* menu)
- Steps to edit a configuration

**Note:** The examples shown are for a C67x DSP, hence the –mv6700 option.
CCS Graphical Interface for Build Options

To make it easier to choose build options, CCS provides a graphical user interface (GUI) for the various compiler options. Here’s a sample of the Debug configuration options discussed earlier:

-\texttt{g -fr$(Proj\_dir)/Debug -d\_DEBUG -mv6700}

These are the default build options for a new project.

GUI has 8 pages of options for code generation tools.

Basic page defaults (in this example) are \texttt{g -mv6700}.

There is a one-to-one relationship between the items in the text box and the GUI check and drop-down box selections. Once you have mastered the various options, you’ll probably find yourself just typing in the options.

Here are a few more compiler option pages (the ones that contain options from the Debug configuration):

Feedback page.
Default is \texttt{-g}, which suppresses some tool feedback.
Why Use –fr and –fs?

When changing configurations, using –fr prevents your .obj and .out files from being overwritten. While not required, it allows you to preserve all variations of your project's object and executable files.

Similarly, –fs allows you to place the assembly files generated by the compiler to any folder you specify. Commonly, users place them into the same folder where they store their executable outputs. Keeping all versions of the generated assembly files lets you compare the effects of different build configurations.

Using Separate Output Folders

- When changing configurations, the –fr and –fs options prevent files from being overwritten.
- While not required, it allows you to preserve all variations of your project's output files.

\[-\text{fr} \text{ and } -\text{fs} \text{ options prevent files from being overwritten. While not required, it allows you to preserve all variations of your project's output files.} \]
The \texttt{-d\_DEBUG} explained:

- \texttt{-d} defines a project wide symbol. It’s like adding the following \texttt{#define} to all source files:
  \begin{verbatim}
  \#define _DEBUG 1
  \end{verbatim}

- CCS tools do not use the \_DEBUG symbol, it is provided for your convenience.

- You might use this symbol in your source code for debugging. Here’s a C example:
  \begin{verbatim}
  if DEBUG then printf( ... );
  \end{verbatim}
Linker Build Options

There are many linker options but these four handle all of the basic needs.

- `-o <filename>` specifies the output (executable) filename.
- `-m <filename>` creates a map file. This file reports the linker’s results.
- `-c` tells the compiler to autoinitialize your global and static variables.
- `-x` tells the compiler to exhaustively read the libraries. Without this option libraries are searched only once, and therefore backwards references may not be resolved.

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-o &lt;filename&gt;</code></td>
<td>Output file name</td>
</tr>
<tr>
<td><code>-m &lt;filename&gt;</code></td>
<td>Map file name</td>
</tr>
<tr>
<td><code>-c</code></td>
<td>Auto-initialize global/static C variables</td>
</tr>
<tr>
<td><code>-x</code></td>
<td>Exhaustively read libs (resolve back ref's)</td>
</tr>
</tbody>
</table>

The linker is explored in greater depth in Chapter 11.
Using the Configuration Tool (.TCF files)

The Configuration Tool is often called the DSP/BIOS Configuration Tool since many of its modules pertain to DSP/BIOS items. Sometimes it’s also called the GUI Tool or GUI Configuration Tool. In this workshop, we use all these names interchangeably; though, most often we’ll call it the Config Tool.

The Config Tool creates/edits a Configuration DataBase (.TCF) file. If we talk about using .TCF files, we’re also talking about using the Config Tool. The following figure shows a .TCF file opened within the configuration tool:

As the bullets in the figure state, the Config Tool simplifies embedded design by providing a GUI interface for many system choices, and by automating many requirements. For pure coding (and code optimization) the Config Tool offers only a few features. These few features will be explored in this – and the next few – chapters. Later in the workshop, we begin to explore its many other capabilities. To explore its full capabilities, we suggest you attend the 4-day DSP/BIOS workshop. (We just don’t have time to explore all of it in this workshop.)

When a .TCF file is saved, the Config Tool creates a number of other files:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.tcf</td>
<td>Configuration Database</td>
</tr>
<tr>
<td>.cfg_c.c</td>
<td>C code created by Config Tool</td>
</tr>
<tr>
<td>.cfg.s62</td>
<td>ASM code created by Config Tool</td>
</tr>
<tr>
<td>.cfg.cmd</td>
<td>Linker commands</td>
</tr>
<tr>
<td>.cfg.h</td>
<td>header file for *cfg_c.c</td>
</tr>
<tr>
<td>.cfg.h62</td>
<td>header file for *cfg.s62</td>
</tr>
</tbody>
</table>
When you add a .TCF file to your project, CCS automatically adds the C and assembly (S62) files. To prevent confusion, these files are added to the **Generated Files** folder.

**Going further with C**

- The *TMS320C6000 Compiler Tutorial* is an invaluable reference. You can find this excellent resource built into Code Composer Studio.
- Also recommended is the *TMS320C6000 Programmers Guide*. It contains code optimization details for C, Linear Assembly, and standard assembly programming.
- All the options are detailed in *TMS320C6000 Optimizing C Compiler User's Guide*. It’s highly recommended that you take time to read through the entire manual. OK, we know that reference manuals can be boring (and this one isn’t any different) but the information you gain will be worth it.
Lab 2 – Using C programs in CCS

Please refer to the separate lab exercises manual for complete directions on how to complete the hands-on lab exercise.

CCS Automation (After the Lab)

Command Window

Some frequently used commands:
- help
- dlog <filename>,a
- dlog close
- alias ...
- take <filename.txt>
- load <filename.out>
- reload
- reset
- restart
- ba <label>
- wa <label>
- run <cond>
- run <label>
- go <label>
- step <number>
- cstep <number>
- halt
GEL Scripting

GEL: General Extension Language
- C style syntax
- Large number of debugger commands as GEL functions
- Write your own functions
- Create GEL menu items

CCS Scripting / Debug Server Scripting (DSS)

OLD Version
We recommend using DSS instead of CCS Scripting
DSS supports both CCSv3.3 and CCSv4

- Debug using VB Script or Perl
- Using CCS Scripting, a simple script can:
  - Start CCS
  - Load a file
  - Read/write memory
  - Set/clear breakpoints
  - Run, and perform other basic debug functions
Where DSS fits in …

1. Start with CC_APP.EXE
2. Isolate debugging engine code from GUI code – call it the DebugServer
3. Provide a DebugServer.DLL library independent of CCS GUI application
   - Use independently, and
   - Re-used with new CCS/Eclipse GUI

DSS in CCSv4 Architecture
Scripting

Problem:
- Some tasks such as testing need to run for hours or days without user interaction
- Need to be able to automate common tasks

Solution:
- CCSv4 has a complete scripting environment allowing for the automation of repetitive tasks such as testing and performance benchmarking.
- The CCSv4 Scripting Console allows you to type commands or to execute scripts within the IDE.

DSS : Debug Server Scripting

What is DSS Used for?
- Automation of debug tasks
  - Testing
  - Profiling

What are the advantages over CCS Scripting?
- SPEED!
  - No GUI dependency
  - In process
  - Low overhead API
- Documentation
  - Every API is automatically documented
- Support
  - DSS is actively maintained
  - Our own tests rely on it
DSS: **Sample Script**

```javascript
// Create Scripting Environment
var env = new ScriptingEnvironment();

// Begin logging - log *everything* to a file and INFO messages to the console
env.traceBegin("dsslog.xml");
env.traceSetFileLevel(TraceLevel.ALL);
env.traceSetConsoleLevel(TraceLevel.INFO);

// Open a debug session
var server = env.getServer("DebugServer.1");
var session = server.openSession();

// Reset CPU
session.target.reset();

// Load the .out file
try {
    session.memory.loadProgram("simple.out");
} catch (ex) {
    // Do Failure Routine
}

// Set start & stop profiling BPs
session.breakpoint.add("simple.c", 5);
session.breakpoint.add("simple.c", 20);

// Run to first (start) breakpoint
session.target.run();

// Open a profiling session
profServer = env.getServer("ProfileServer.1");
profSession = profServer.openSession(debugSession);

// Run to second breakpoint and count cycles
profSession.runBenchmark();

// Read the value of pseudo-reg CLK and log, report value
var cycleCount = session.memory.readRegister("CLK");
env.traceWrite("cycle.CPU (Incl. Total): " + cycleCount + ": cycles");

// Close our Session and Server
session.terminate();
server.stop();
```

---

DSS: **Sample Script**

```
// Get start & stop profiling BPs
session.breakpoint.add("simple.c", 5);
session.breakpoint.add("simple.c", 20);

// Run to first (start) breakpoint
session.target.run();

// Open a profiling session
profServer = env.getServer("ProfileServer.1");
profSession = profServer.openSession(debugSession);

// Run to second breakpoint and count cycles
profSession.runBenchmark();

// Read the value of pseudo-reg CLK and log, report value
var cycleCount = session.memory.readRegister("CLK");
env.traceWrite("cycle.CPU (Incl. Total): " + cycleCount + ": cycles");

// Close our Session and Server
session.terminate();
server.stop();
```
BIOS Textual Configuration (TCF)

Tconf Script (.tcf)

```javascript
/* load platform */
utils.loadPlatform("ti.platforms.dsk6416");
config.board("dsk6416").cpu("cpu0").clockOscillator = 600.0;

/* make all prog objects JavaScript global vars */
utils.getProgObjs(prog);

/* Create Memory Object */
var myMem = MEM.create("myMem");
myMem.base = 0x00000000;
myMem.len = 0x00100000;
myMem.space = "data";

/* generate cfg files (and CDB file) */
prog.gen();
```

- Textual way to create and configure TCF files
- Runs on both PC and Unix
- Create #include type files (.tci)
- More flexible than Config Tool
Introduction

Usually, people cringe when they hear the word pipeline in a casual conversation due to its infamous history of causing heartache to programmers. In many of our DSP workshops, the students are relieved to learn that “most of the time, you don’t need to worry about it”. It’s the “most” word that prompts the inevitable question: “so, what are the exceptions and how do I write code to avoid them?” Because the pipeline is an integral part of the ‘C6000 and certainly affects performance and programming, we need a basic understanding of its operation and what happens in each stage. Getting the code to work is usually straightforward and requires only a few guidelines. Optimizing this code to take full advantage of all eight functional units takes a little more time. With the help of optimizing tools, the task becomes easier.

At the end of this section, you should have a firm grasp of topics such as:

- Pipeline
- Fetch Packet
- Execute Packet
- Delay Slots, Latency
- Parallel Operations
- VLIW, VelociTI™

Learning Objectives

- Describe the C6000 hardware pipeline operation
- Define C6000 terminology:
  - Fetch Packet (FP)
  - Execute Packet (EP)
  - Delay Slots / Latency
- Modify standard C6000 assembly language to deal with instruction latency
- Explain how parallel, partially parallel, and non-parallel code differs in the C6000 pipeline
- List the differences between the C62x, C67x, and C64x pipelines
Why a Pipeline?

Chapter Topics

Introduction to Pipeline .......................................................................................................................... 3-1

Why a Pipeline? ........................................................................................................................................ 3-3
Non-Pipelined vs. Pipelined CPU ........................................................................................................... 3-3

‘C6000 Pipeline Stages.............................................................................................................................. 3-4
Program Fetch (PF-stage) ......................................................................................................................... 3-5
Decode (D-Stage) ...................................................................................................................................... 3-6
Execute (E-Stage) ...................................................................................................................................... 3-6
Load Execute Phases (similar to program fetch)................................................................................. 3-7
What does a branch look like? .............................................................................................................. 3-8
Final Pipeline Phases .............................................................................................................................. 3-9
One More Definition .............................................................................................................................. 3-10

Running Code thru the Pipeline............................................................................................................ 3-11
Fetch Packet ........................................................................................................................................... 3-11
‘C6x System Block Diagram (256-bit access)..................................................................................... 3-11

Pipeline Code Example - Sum of Products ............................................................................................. 3-12
Pipeline Code Example (Program Fetch) ............................................................................................... 3-13
Pipeline Code Example (Dispatch and Decode) .................................................................................... 3-14
Pipeline Code Example (Execute) ......................................................................................................... 3-15

Pipeline Code Example - Delay Slots .................................................................................................... 3-16
Pipeline Code Example – Delaying Add (LD is Next…) ...................................................................... 3-18
Pipeline Code Example - NOPs and Delay Slots ................................................................................ 3-19
Pipeline Code Example - NOPs Added ................................................................................................. 3-20

Pipeline Code Example – Using Multi-Cycle NOPs ............................................................................ 3-21

Pipeline Code Example - Benchmark ..................................................................................................... 3-22

Parallel Instructions, Execute Packets .................................................................................................. 3-23
Pipeline - Serial Execution ..................................................................................................................... 3-23
Pipeline - Partially Parallel Execution ................................................................................................. 3-24

Pipeline - Fully Parallel Execution ....................................................................................................... 3-28

C64x Pipeline Variations ....................................................................................................................... 3-30

Optional: ‘C67x Pipeline Variations ....................................................................................................... 3-31
‘C67x Delay Slots / Latencies ............................................................................................................... 3-31
Functional Unit Latency ....................................................................................................................... 3-32

Optional Topics ..................................................................................................................................... 3-34
VelociTI vs. Standard VLIW .................................................................................................................. 3-34
Execute Packet Alignment .................................................................................................................... 3-37
Advanced Instruction Packing on the ‘C64x .......................................................................................... 3-38
Why a Pipeline?

What types of operations does the CPU need to perform in order to execute an instruction?

- (F) Fetch the instruction from memory
- (D) Decode the instruction (what type of instruction it is – e.g. ADD, MPY)
- (E) Execute it (actually do the ADD, MPY, etc)

Many general purpose CPUs actually perform these steps in a serial fashion. With this scheme, one instruction takes multiple cycles to complete. However, because the ‘C6x provides multiple buses, separate functional units and parallel hardware, it can overlap these operations which significantly increases performance.

Non-Pipelined vs. Pipelined CPU

Let’s look at a comparison of a non-pipelined vs. a pipelined CPU:

<table>
<thead>
<tr>
<th>CPU Type</th>
<th>Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Pipelined</td>
<td>F₁ D₁ E₁</td>
</tr>
<tr>
<td></td>
<td>F₂ D₂ E₂</td>
</tr>
<tr>
<td></td>
<td>F₃ D₃ E₃</td>
</tr>
<tr>
<td>Pipelined</td>
<td>F₁ D₁ E₁</td>
</tr>
<tr>
<td></td>
<td>F₂ D₂ E₂</td>
</tr>
<tr>
<td></td>
<td>F₃ D₃ E₃</td>
</tr>
<tr>
<td></td>
<td>Pipeline full</td>
</tr>
</tbody>
</table>

This diagram shows the essential workings of a pipeline – several instructions in different “stages” during the same clock cycle. As you can see, once the pipeline is “full” (all stages operating simultaneously), an instruction executes on every cycle. The major advantage of a pipeline is performance. If you compare the previous two scenarios, you’ll notice that a pipelined architecture executes E₃ much sooner than the serial version. What is the disadvantage of a pipeline? Discontinuities. If the CPU executes a branch (or similar instruction), the pipeline must be “flushed” – i.e. it must start over with a new fetch and discard any instructions which have already been fetched. For example, if F₁ is a branch instruction, E₁ would execute the branch and cause the program to jump to a new location, thereby making F₂ and F₃ steps unnecessary. Minimizing this “flush” overhead is critical to any pipelined architecture.
‘C6000 Pipeline Stages

Let’s take a look at one instruction’s migration through the ‘C6x pipeline. We’ll start by reviewing the three basic stages:

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF</td>
<td>Generate program fetch address</td>
</tr>
<tr>
<td></td>
<td>Read opcode</td>
</tr>
<tr>
<td>D</td>
<td>Route opcode to functional unit</td>
</tr>
<tr>
<td></td>
<td>Decode instruction</td>
</tr>
<tr>
<td>E</td>
<td>Execute instruction</td>
</tr>
</tbody>
</table>

In the Program Fetch stage (PF), the CPU generates an address, reads the instruction’s opcode from memory, and sends it to the decoder. The decode logic (D) intelligently routes the opcode to the functional unit which determines the type of instruction (ADD, SUB, MPY, etc). Then, the CPU executes (E) the instruction.
Program Fetch (PF-stage)

What is the cycle time of the ‘C6x? 4ns or 5ns. So, how can the CPU generate an address and read a memory location in one cycle? This would require a VERY fast memory. In order to talk with existing memories – as well as meet its own internal timing requirements – the ‘C6x actually breaks up the Program Fetch stage into 4 phases. These P-phases correspond to hardware:

### Program Fetch Phases

<table>
<thead>
<tr>
<th>Program Fetch Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PG</td>
<td>generate fetch address</td>
</tr>
<tr>
<td>PS</td>
<td>send address to memory</td>
</tr>
<tr>
<td>PW</td>
<td>wait for data ready</td>
</tr>
<tr>
<td>PR</td>
<td>read opcode</td>
</tr>
</tbody>
</table>

During the PG and PS phases, the CPU *generates* the address of the instruction and *sends* it to the memory. The CPU then *waits* for “data ready” at the memory (PW) before *reading* the 32-bit opcode (PR). So, at the end of 4 cycles, the instruction has reached the CPU.

**Note:** The “memory” shown above represents all memory from the CPU’s perspective. While the ‘C6201 accesses internal memory with zero additional cycles, the external memory interface may require additional cycles. As with most microprocessors, this is implemented by halting the processor while the pipeline is in the PW phase. Once the CPU receives a memory ready signal, processing continues.
Decode (D-Stage)

The CPU is now ready to decode (D) the instruction. The CPU breaks decode into two phases. During the DP phase, the CPU intelligently routes (or dispatches) the instruction to one of the eight functional units where the instruction is decoded:

Execute (E-Stage)

When the instruction reaches the execute (E) stage, the CPU executes each instruction within its respective functional unit. How many cycles would you like each instruction to require? One or less, of course! Well ... all 'C62x instructions execute in a single cycle. However, some results (e.g. a load) are delayed:

Instruction Delays

<table>
<thead>
<tr>
<th>Description</th>
<th># Instr.</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Cycle</td>
<td>All, instr’s except ...</td>
<td>0</td>
</tr>
<tr>
<td>Multiply</td>
<td>MPY, SMPY</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>LDB, LDH, LDW</td>
<td>4</td>
</tr>
<tr>
<td>Branch</td>
<td>B</td>
<td>5</td>
</tr>
</tbody>
</table>

Most ‘C6x instructions fall in the ISC (single cycle) category, such as ADD, SUB, AND, OR, XOR, etc. MPY (which has several varieties) requires one cycle of delay. A one cycle delay means that the multiplies results will not be available until one cycle later (i.e. not available for the next instruction
to use.) The results of a load (examined below) are delayed for 4 cycles. Branches reach their target
destination five cycles later. Store instructions are viewed as single cycle from the CPU’s perspective
because there are no execution phases associated with a store beyond E1 (unlike a load). Because the
maximum delay is five cycles (total of 6 execution cycles), the CPU breaks the execute (E) stage into
six parts:

### Execute Phases

<table>
<thead>
<tr>
<th>Execute Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>ISC instructions completed</td>
</tr>
<tr>
<td>E2</td>
<td>IMPY instructions completed</td>
</tr>
<tr>
<td>E3</td>
<td></td>
</tr>
<tr>
<td>E4</td>
<td></td>
</tr>
<tr>
<td>E5</td>
<td>Load value into register</td>
</tr>
<tr>
<td>E6</td>
<td>Branch to destination complete</td>
</tr>
</tbody>
</table>

Note: No instructions complete in E3 or E4; therefore they
are left blank, as placeholders, in the table above.

### Load Execute Phases (similar to program fetch)

Load execute phases are very similar to a program fetch in terms of cycles, pipeline phases, and
operation. Consider the following diagram:

As a review, we’ve placed the PG → DC phases on the diagram. The next phase in the pipeline is
E1, E2, E3, etc. If you think about a load, it requires a memory access to “fetch” a value. The results
of a load are delayed four cycles, a total latency of five executions: four (E1-E4) to interface to
memory (like the PG → PR phases) and one (E5) to write the value to the register.
What does a branch look like?

Let’s take a look at how a branch instruction executes in the pipeline. As described in the previous table, a branch contains 5 delay slots, which could each contain 1 to 8 instructions (that means you could execute up to 40 instructions in the delay slots of a branch!)

The branch instruction is initiated in E1. The E2-E6 phases of the branch are simply place-holders for delay slots (no actions occur in these phases of the branch instruction). As shown below, the branch-to address is generated for PG (as indicated by “Branch Target” on the bottom line) during the same cycle when the branch executes (E1). This is the smallest latency possible. Assuming the delay slots are filled with useful work (up to 40 instructions), the branch overhead is absolutely minimal. You could state that the branch instruction really only requires one cycle (E1) to initiate and the rest of the cycles can execute useful code.
Final Pipeline Phases

Our pipeline phase diagram is now complete with 12 phases:

Here’s another way to look at the pipeline. Like the conceptual pipeline diagram we examined at the beginning of the module, here’s the final summary of the ‘C6x pipeline. We’ve even highlighted the cycle where we show a full pipeline.

Next, we’ll discover how the ‘C6x handles multiple instructions in the pipeline.
One More Definition

The CPU Reference Guide also uses one other term when describing the ‘C6000 pipeline: latency. Latency describes how many cycles it takes before the result is available. In other words:

\[ \text{Latency} = 1 + \# \text{ of delay slots} \]

Adding this to our previous table, the ‘C62x result latencies would be:

<table>
<thead>
<tr>
<th>Description</th>
<th>Instructions</th>
<th>Delay</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Cycle</td>
<td>All, except ...</td>
<td>0</td>
<td>0 + 1 = 1</td>
</tr>
<tr>
<td>Multiply</td>
<td>MPY / SMPY</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Load</td>
<td>LDB/H/W</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Branch</td>
<td>B</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

One More Definition: Latency

What is Latency? The total cycles an instruction requires.
Running Code thru the Pipeline

Fetch Packet

If all functional units were operating simultaneously, how many instructions would execute in parallel? The answer is eight – the number of functional units. For this reason, the ‘C6000 fetches eight 32-bit instructions (256 bits) every clock cycle.

These eight 32-bit instructions form what is called a fetch packet. Each clock cycle, the program counter (PC) is incremented by eight 32-bit locations to access the next fetch packet. The fetch packet moves through each phase of the pipeline (as if it were a single instruction) and then, if necessary, is split into individual instructions during the decode stage.

`'C6x System Block Diagram (256-bit access)`

How does the ‘C6x support fetching eight 32-bit instructions every 4 or 5ns??! Internally, the program memory is structured as fetch packets – 256-bit instructions. Supported by a 256-bit bus, it is feasible to fetch a new VLIW word every cycle. Externally, however, you’ll notice that the data bus is only 32-bits wide, which will require 8 independent fetches if x32 memory is used.
Pipeline Code Example - Sum of Products

Let’s see how this works by reviewing our last code example and watching it move through the pipeline.

In order to understand how the pipeline works, it is necessary to watch a fetch packet move through the entire pipeline, paying close attention to the interaction of the instructions. We developed this code during the first module, so it should be familiar. [Note: it’s assumed that A4 was previously cleared.]

```
MVK .S1 40, A2
loop:LDH .D1 *A5++, A0
       LDH .D1 *A6++, A1
       MPY .M1 A0, A1, A3
       ADD .L1 A3, A4, A4
       SUB .L1 A2, 1, A2
       [A2] B .S1 loop
       STH .D1 A4, *A7
```
### Pipeline Code Example (Program Fetch)

This code contains eight instructions which we will assume are aligned on a fetch packet boundary (a boundary of eight 32-bit instructions). During the P-stage, the CPU will read all eight 32-bit instructions (denoted by their respective mnemonics below) in one fetch packet.

This first fetch packet moves from the PG-phase through the PR-phase in the first 4 clock cycles:

[Diagram of Pipeline Example (Fetch Packet)]

[Note: we will assume that the fetch packet resides in internal memory with zero wait states.]

As the first fetch packet moves through each phase of the pipeline, the CPU fetches the next few packets in line and manages them through each stage as well. This creates pending work for the CPU to execute - which it does very quickly!
Pipeline Code Example (Dispatch and Decode)

On the next clock cycle, the fetch packet will enter the DP phase where each 32-bit instruction is analyzed and routed to its respective functional unit for decode:

On the next clock cycle, the first instruction (MVK) moves into the decode (DC) phase while the other instructions wait in line:

In the DC phase, the .S unit will decode the MVK instruction and prepare it for execution.
Pipeline Code Example (Execute)

On the next clock cycle, MVK moves into the E1-phase while the first LDH instruction is being decoded (DC). The other six instructions continue to wait at DP:

The “Done” column on the right hand side shows WHEN the instruction actually completes execution. This will help us determine when a specific instruction finishes executing relative to the other instructions in the pipeline.

On the next clock cycle, MVK finishes LDH moves into E1 while the second LDH is decoded:
Pipeline Code Example - Delay Slots

MVK is an ISC (single cycle) instruction and therefore finishes execution in the E1-phase. LDH, however, is a five-cycle instruction – the plus signs (+) indicate the number of additional cycles (greater than one) required by LDH to complete execution – and will therefore finish execution at the end of E5. These “additional” cycles are actually called *delay slots* and are an important element in understanding the pipeline. Let’s review this information in light of our new definition of delays slots:

### Instruction Delay Slots

<table>
<thead>
<tr>
<th>Description</th>
<th># Instr.</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Cycle</td>
<td>All, instr’s except ...</td>
<td>0</td>
</tr>
<tr>
<td>Multiply</td>
<td>MPY, SMPY</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>LDB, LDH, LDW</td>
<td>4</td>
</tr>
<tr>
<td>Branch</td>
<td>B</td>
<td>5</td>
</tr>
</tbody>
</table>

On the next clock cycle, the first LDH moves to E2 (indicated by the mnemonic moving into E2), the second LDH reaches E1 and the MPY enters DC. Again the plus signs (+) indicate the delay slots for the second LDH:
One clock cycle later, the instructions bump to the next phase – MPY reaches E1:

**Pipeline Example (MPY in E1)**

<table>
<thead>
<tr>
<th>Prog. Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>DP</td>
<td>DC</td>
<td>E1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MPY has one delay slot indicated by only one plus sign (+). Looking at the diagram, do you see an error? We intended to load (LDH) values from memory and then multiply them, but this requires the LDHs to finish BEFORE the MPY begins. Here, though, the MPY starts too early and, therefore, uses incorrect values. Hmmm… somehow, MPY must be delayed. Well, keep this in mind while we continue moving through the pipeline to see if any other issues appear. We’ll solve this issue of the loads not finishing in time, shortly. There are other issues to deal with…

Another clock cycle later, ADD moves into E1:

**Pipeline Example (ADD in E1)**

<table>
<thead>
<tr>
<th>Prog. Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>DP</td>
<td>DC</td>
<td>E1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Once again, we seem to have found a problem: the ADD starts before the MPY is finished.
Let’s proceed to the next clock cycle…

The ADD and MPY instructions finish executing on the same cycle and the loads aren’t finished yet! It is obvious that our code, while written in sequential order, is not executed in the same order and will therefore not work.

**Pipeline Code Example – Delaying Add (LD is Next…)**

So, how do we fix this problem? Let’s focus on the MPY/ADD instructions first, forcing them to execute in the proper order. Because MPY takes two cycles to execute (one delay slot), we must delay MPY one cycle so it completes before the ADD begins. How do you tell a CPU to “do nothing” for one cycle? One way, use a NOP. Let’s see if this works in the pipeline. While NOPs are obviously not an optimal solution (NOP = Not Optimized Properly), they’ll do for now. Later, we’ll show how to fill the delay slots of instructions with useful work.

Adding a NOP after MPY causes MPY to reach E2 while ADD is still at DC.
Pipeline Code Example - NOPs and Delay Slots

On the next clock cycle, MPY and NOP will finish execution while the ADD reaches E1:

It works. Adding one NOP after the MPY instruction delays MPY by one cycle, which avoids the conflict with ADD. So, how many NOPs should we add to delay the load instructions? If you review one of the previous diagrams, you’ll notice that adding four NOPs after the second load instruction enables the loads to finish before the MPY begins.

How do you know how many NOPs to add for each instruction? A surprising similarity exists between the number of delay slots associated with each instruction and the number of additional NOPs required to make the code execute sequentially – in fact, they are equivalent:

<table>
<thead>
<tr>
<th>Instruction Types - #NOP’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Single Cycle</td>
</tr>
<tr>
<td>Multiply</td>
</tr>
<tr>
<td>Load</td>
</tr>
<tr>
<td>Branch</td>
</tr>
</tbody>
</table>
Pipeline Code Example - NOPs Added

Therefore, in order to complete the code example, add NOPs to each instruction that contains delay slots:

```
loop:
  MVK .S1 40, A2
  LDH .D1 *A5++, A0
  LDH .D1 *A6++, A1
  NOP
  NOP
  NOP
  NOP
  MPY .M1 A0, A1, A3
  NOP
  ADD .L1 A3, A4, A4
  SUB .L1 A2, 1, A2
  [A2] B .S1 loop
  NOP
  NOP
  NOP
  NOP
  STH .D1 A4, *A7
```
Pipeline Code Example – Using Multi-Cycle NOPs

Typing in so many NOPs can be cumbersome (not to mention the program memory increase), so the assembler allows the programmer to use a constant value with the NOP instruction to indicate the number of NOPs desired:

\[
\text{NOP} \\
\text{NOP} \\
\text{NOP} \\
\text{NOP} \\
\text{NOP} \\
\text{NOP} \text{ } 4
\]

This saves program memory area as well as reduces carpal-tunnel-syndrome effects induced by typing NOP after NOP. So, in our code example, we write a “NOP x” after each instruction which requires delay slots where “x” corresponds to the number of delay slots:

```
Pipeline Example - Multi-cycle NOP’s

MVK .S1 40,A2
loop: LDH .D1 *A5++, A0
       LDH .D1 *A6++, A1
       NOP 4
       MPY .M1 A0,A1,A3
       NOP
       ADD .L1 A3,A4,A4
       SUB .L1 A2,1,A2
       NOP 5
       [A2] B .S1 loop
       STH .D1 A4,*A7
```
Pipeline Code Example - Benchmark

What is the benchmark for this code in terms of cycles?

<table>
<thead>
<tr>
<th>Inner Loop:</th>
<th>1st Load</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd Load</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>MPY</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ADD/SUB</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Total: \(16 \times 40 = 640 + 2 = 642\) cycles

642 cycles (@4-5ns/cycle) is still faster than many of today’s DSPs and general-purpose CPUs. However, this benchmark assumes that the delay slots are filled with inefficient NOPs (remember, NOP means “not optimized properly”). Is it possible to fill the delay slots with something useful? The answer is absolutely yes. You can fill delay slots with other instructions. In fact, when all optimizations provided by this architecture are put into action, the benchmark reduces to 28 cycles – even less than the number of loops!!! The “how” discussion is outside the scope of this module, but is covered in detail in later modules as well as the Programmer’s Reference Guide.

The preceding cycle times were for the instructions common to all C6000 processors. If you use the advantages of the C64x, you’ll see a significant gain in performance.

<table>
<thead>
<tr>
<th></th>
<th>C6203</th>
<th>40 terms → 642 cycles</th>
<th>2.1 µs @ 300 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6203</td>
<td>28 cycles</td>
<td></td>
<td>93 ns @ 300 MHz</td>
</tr>
<tr>
<td>C64x</td>
<td>19 cycles</td>
<td></td>
<td>19 ns @ 1 GHz</td>
</tr>
</tbody>
</table>

But what about the 256 term dot-product from the lab exercise:

<table>
<thead>
<tr>
<th></th>
<th>C6203</th>
<th>256 terms → 136 cycles</th>
<th>453 ns @ 300 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6203</td>
<td>79 cycles</td>
<td></td>
<td>79 ns @ 1 GHz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dot-Product Terms</th>
<th>C62x</th>
<th>C64x</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 term</td>
<td>28</td>
<td>19</td>
</tr>
<tr>
<td>256 term</td>
<td>136</td>
<td>79</td>
</tr>
</tbody>
</table>

- While 28 cycles is great, the C64x is even better
- C64x runs more than twice as fast per cycle as C62x
- Profiling results in a few more cycles than shown here, due to overhead such as function call/return
- Later, you’ll explore how these low cycle counts are achieved
- Theoretically, if we can get 20 cycles for the C62x and 10 for the C64x, how many cycles will it take on the C64x+?
Parallel Instructions, Execute Packets

To complete our journey through the pipeline, we need to examine how the CPU executes parallel instructions. We intend to study three examples using arbitrary code to demonstrate the pipeline’s operation given different input code:

### Pipeline Code Example - Serial

<table>
<thead>
<tr>
<th>Serial</th>
<th>Partially Parallel</th>
<th>Fully Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>B .S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVK .S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD .L1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD .L1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPY .M1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPY .M1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDW .D1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDB .D1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Pipeline - Serial Execution

Our first code example and pipeline discussion displayed how a fetch packet with “serial” instructions (i.e. not parallel) moved through the pipeline – at the DP-phase, each individual 32-bit instruction was routed (dispatched) to the proper functional unit ONE INSTRUCTION AT A TIME. This is what is meant by “serial” execution:

### Pipeline Code Example - Serial Execution

<table>
<thead>
<tr>
<th>Prog. Fetch</th>
<th>Decode</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>DP</td>
<td>E1 E2 E3 E4 E5 E6</td>
</tr>
<tr>
<td>FP5-2</td>
<td>MVK</td>
<td>B + + + + + +</td>
</tr>
</tbody>
</table>

We've already examined this scenario …
Pipeline - Partially Parallel Execution

What happens when parallel instructions are used? How do we write parallel instructions? To make an instruction parallel, we simply add the “double pipe symbol” – “||” – to the instruction we want in parallel with the previous one. We can use up to 8 instructions in parallel as long as each one uses a different functional unit – this is a key point regarding the proper use of CPU resources. Look carefully at the two code segments in the diagram below – what changed from the first to the second? Instead of simply using the A-side units, some B-side units were chosen to avoid conflicts.

### Pipeline Code Example - Partially Parallel

<table>
<thead>
<tr>
<th>Serial</th>
<th>Partially Parallel</th>
<th>Fully Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>B .S1</td>
<td>B .S1</td>
<td></td>
</tr>
<tr>
<td>MVK .S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD .L1</td>
<td>ADD .L1</td>
<td></td>
</tr>
<tr>
<td>ADD .L1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPY .M1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPY .M1</td>
<td>MPY .M1</td>
<td></td>
</tr>
<tr>
<td>LDW .D1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDB .D1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Shown below is the partially parallel code example in the pipeline at the DP phase:

When the CPU fetched 8 instructions simultaneously, we called this grouping a fetch packet. What do we call a grouping of instructions which the CPU executes simultaneously? An execute packet. A fetch packet can therefore contain multiple execute packets depending on whether the code uses parallel instructions. In this example, three execute packets are shown.

Use of parallel instructions does not affect the Program Fetch phases. However, at the DP-phase, the CPU breaks the fetch packet into smaller execute packets – each containing a group of parallel instructions. Then, the CPU routes the instructions to their respective functional units ONE EXECUTE PACKET AT A TIME. The traditional VLIW architecture required that the fetch packets stay “glued together” through execution. The ‘C6x’s Advanced VLIW architecture provides more flexibility by allowing smaller “chunks” to execute.

On the next clock cycle, the first execute packet enters the decode (DC) phase where the individual 32-bit instructions are decoded:
Page left intentionally blank.
After the next clock cycle, the first execute packet (and all of the instructions it contains) enters the E1-phase and begins execution while the next execute packet reaches DC:

**Partially Parallel Execution**

<table>
<thead>
<tr>
<th>Decode</th>
<th>Execute</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DC</td>
<td>E1</td>
</tr>
<tr>
<td>B</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

All instructions contained in the first execute packet are executed simultaneously. Again, the plus signs (+) indicate the delay slots for each instruction.

In the next cycle, the MVK instruction completes execution while the branch continues. Also, the next Execute Packet (EP) begins execution.

**Partially Parallel Execution**

<table>
<thead>
<tr>
<th>Decode</th>
<th>Execute</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DC</td>
<td>E1</td>
</tr>
<tr>
<td>B</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

This process continues as more fetch packets and execute packets arrive. As you can see, using parallel instructions along with multiple functional units can dramatically increase performance.
Pipeline - Fully Parallel Execution

So, we’ve seen a fetch packet containing eight execute packets (serial) and one containing three execute packets using parallel instructions. It is also possible to have all 8 instructions in parallel:

![Fully Parallel Code Example](image)

Notice in the 3rd code example that no resources are duplicated and both sides are being fully utilized. In this case, the fetch packet contains only one execute packet and all 8 instructions reach E1 and execute simultaneously.

Let’s view this fully parallel code in the pipeline:

![Fully Parallel Execution](image)
On the next cycle, the single-cycle instructions complete and the others continue.

If another execute packet containing 8 instructions was in line, it would enter E1 at this time - WOW - 8 instructions executing every 5ns - this is tremendous performance. In this design workshop as well as the Programmer’s Reference Guide, we spend significant time explaining how to properly fill delay slots and use parallel instructions. Of course, TI supplies several tools (such as the Assembly Optimizer and C Compiler) which significantly simplify the process.

Remember:
- Branches have 5 delay-slots before completion.
- Loads have 4 delay-slots before completion.
- Multiplies have 1 delay-slot before completion.
C64x Pipeline Variations

With the ‘C64x being a superset of the ‘C62x it’s not surprising the ‘C64x pipeline is nearly identical to its predecessor. Only a few new instructions that execute on the .M units require additional delay slots … as summarized below:

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Delay Slots</th>
<th>Result Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>C62x and C64x Instructions:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single-Cycle Instructions</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>16 x 16 Multiplies (MPYs)</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Loads (all sizes)</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Branches</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>C64x Exceptions:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Special Instructions</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>BITC, BITR, AVGx, ROTL, SHFL, DEAL, XPNDx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply Extensions</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>DOTPx, MPYHI, MPYLI, MPYx, GMPY4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C64x+ Pipeline

<table>
<thead>
<tr>
<th>No Unit</th>
<th>.L</th>
<th>.M</th>
<th>.S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Delay Slots DINT RINT</td>
<td>0 Delay Slots ADDSUB ADDSUB2</td>
<td>3 Delay Slots CMPY CMPYR CMPYR1 MPY32 (64-bit result)</td>
<td></td>
</tr>
<tr>
<td>N/A SPKERNEL SPKERNELR SPLOOP</td>
<td>DPACK2 DPACKX2 SADDSUB SADDSUB2 SHFL3 SSUB2</td>
<td>CMPY MPY32SU DDOTPH2R DDOTPH2 MPY32U</td>
<td></td>
</tr>
<tr>
<td>5 Delay Slots CALLP</td>
<td>0 Delay Slots DMV RPACK2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

◆ All C64x+ instructions can be dispatched every cycle, that is they only tie up their functional unit for 1 cycle

Compatibility

Compatibility is a great advantage of the ‘C6000 family. Not only is the ‘C64x a superset of the ‘C62x, it’s also object code compatible.

This means you don’t need to re-compile your object libraries to execute them on the ‘C64x or C64x+.

Though, to gain full speed advantage of the new CPUs, you will need to recompile with the appropriate -mv option.
Optional: ‘C67x Pipeline Variations

The ‘C67x is also a superset of – and object code compatible with – the ‘C62x architecture. It has two pipeline differences from the ‘C62x:

- Like the ‘C64x, some floating-point instructions on the ‘C67x require additional delay slots.
- Unlike either the ‘C62x or ‘C64x, five of its double-precision floating-point instructions require functional unit latency.

‘C67x Delay Slots / Latencies

<table>
<thead>
<tr>
<th>‘C67x Delay Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>ABSSP</td>
</tr>
<tr>
<td>ADDAD</td>
</tr>
<tr>
<td>CMPEQSP</td>
</tr>
<tr>
<td>CMPGTSP</td>
</tr>
<tr>
<td>CMPLTSP</td>
</tr>
<tr>
<td>RCPSP</td>
</tr>
<tr>
<td>RSRQSP</td>
</tr>
<tr>
<td>ABSDP</td>
</tr>
<tr>
<td>RCPDP</td>
</tr>
<tr>
<td>RSRQDP</td>
</tr>
<tr>
<td>SPDP</td>
</tr>
<tr>
<td>CMPEQDP</td>
</tr>
<tr>
<td>CMPGTDP</td>
</tr>
<tr>
<td>CMPLTDP</td>
</tr>
<tr>
<td>INTDP</td>
</tr>
<tr>
<td>INTDPU</td>
</tr>
<tr>
<td>LDDW</td>
</tr>
</tbody>
</table>

Looking at the additional ‘C67x instructions, we see:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Delay Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDSP</td>
<td>3</td>
</tr>
<tr>
<td>DPINT</td>
<td>3</td>
</tr>
<tr>
<td>DSP</td>
<td>3</td>
</tr>
<tr>
<td>DPTRUNC</td>
<td>3</td>
</tr>
<tr>
<td>INTSP</td>
<td>3</td>
</tr>
<tr>
<td>INTSPU</td>
<td>3</td>
</tr>
<tr>
<td>MPYSP</td>
<td>3</td>
</tr>
<tr>
<td>SPINT</td>
<td>3</td>
</tr>
<tr>
<td>SPTRUNC</td>
<td>3</td>
</tr>
<tr>
<td>SUBSP</td>
<td>3</td>
</tr>
<tr>
<td>ADDDP</td>
<td>6</td>
</tr>
<tr>
<td>SUBDP</td>
<td>6</td>
</tr>
<tr>
<td>MPYI</td>
<td>8</td>
</tr>
<tr>
<td>MPYID</td>
<td>9</td>
</tr>
<tr>
<td>MPYDP</td>
<td>9</td>
</tr>
</tbody>
</table>

Our previous "delay-slot" chart details the delay slots required by the four ‘C62x instruction types. Remember, delay slots are the additional cycles required after the E1 phase of the pipeline. Therefore the instruction latency is 1 plus the number of delay slots; e.g. LDW = 1+4 = 5.

<table>
<thead>
<tr>
<th>‘C6700 Results Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>ABSSP</td>
</tr>
<tr>
<td>ADDAD</td>
</tr>
<tr>
<td>CMPEQSP</td>
</tr>
<tr>
<td>CMPGTSP</td>
</tr>
<tr>
<td>CMPLTSP</td>
</tr>
<tr>
<td>RCPSP</td>
</tr>
<tr>
<td>RSRQSP</td>
</tr>
<tr>
<td>ABSDP</td>
</tr>
<tr>
<td>RCPDP</td>
</tr>
<tr>
<td>RSRQDP</td>
</tr>
<tr>
<td>SPDP</td>
</tr>
<tr>
<td>CMPEQDP</td>
</tr>
<tr>
<td>CMPGTDP</td>
</tr>
<tr>
<td>CMPLTDP</td>
</tr>
<tr>
<td>INTDP</td>
</tr>
<tr>
<td>INTDPU</td>
</tr>
<tr>
<td>LDDW</td>
</tr>
</tbody>
</table>

What is Latency? The total cycles an instruction requires.
Optional: ‘C67x Pipeline Variations

Functional Unit Latency

We've added one additional column to a previous chart: functional unit latency (FUL).

### Functional Unit Latency (FUL)

- # cycles an instruction ties-up a functional unit

<table>
<thead>
<tr>
<th>Description</th>
<th>Delay Slots</th>
<th>#NOPs</th>
<th>FUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Cycle</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Multiply</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Branch</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

- Only on ‘C67x, and then ...
  affects only eight DP instructions

That's our acronym for functional-unit latency. This describes how many cycles a given instruction "ties" up a functional unit. All ‘C62x (and ‘C64x) instructions only require functional units for a single-cycle, thus the FUL column is filled with ones in the above chart. A few double-precision floating-point instructions actually need functional units for 2-4 cycles.

Let's compare a few instructions to see their FUL effects.

### FUL . Total Result Latency

<table>
<thead>
<tr>
<th>Instruction</th>
<th>FUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD (1.1)</td>
<td>E1</td>
</tr>
<tr>
<td>MPY (1.2)</td>
<td>E1  E2</td>
</tr>
<tr>
<td>MPYSP (1.4)</td>
<td>E1  E2  E3  E4</td>
</tr>
<tr>
<td>MPYDP (4.10)</td>
<td>E1  E2  E3  E4  E5  E6  E7  E8  E9  E10</td>
</tr>
<tr>
<td>FMPYDP (1.4)†</td>
<td>E1  E2  E3  E4</td>
</tr>
</tbody>
</table>

Notes:  
† Only ‘C67x double-precision instructions have a FUL > 1

The first two instructions are ‘C62x (also ‘C64x and ‘C67x) instructions while the third and fourth instructions demonstrate the effect of FUL on some ‘C67x floating-point instructions. Also, notice that we've categorized these instructions with "FUL.Latency". While the 'C6000 CPU Reference Guide categorizes instructions differently, we've found this to be a convenient, intuitive, and useful method of summarizing the ‘C67x pipeline affects for each instruction.
• The single-cycle integer addition only requires a single functional-unit cycle and has no delay slots; hence, its total latency is "1". We summarized it with "1.1".

• The MPY instruction still only requires a single functional-unit cycle, but requires an additional cycle (single delay-slot) to complete the result and write it into the destination register. With a total result latency of 2, it’s summarized with "1.2".

• The single-precision, floating-point multiply takes a few extra delay slots but doesn't tie up the functional-unit for any additional cycles giving a value of "1.4".

• The final example, a double-precision, floating-point multiply demonstrates a longer functional-unit latency. This instruction requires the .M unit for four cycles. Also, it doesn't write the results until E9 and E10. Putting this together we've summarized the 9 delay-slot MPYDP instruction with "4.10".

Here's a summary of the extra 'C67x instructions along with their FUL.Latency categorization.

<table>
<thead>
<tr>
<th>'.S Unit'</th>
<th>'.L Unit'</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSSP (1.4)</td>
<td>ADDSP (1.3)</td>
</tr>
<tr>
<td>ABSDP (1.2)</td>
<td>ADDDP (2.7)</td>
</tr>
<tr>
<td>CMPEQSP (1.1)</td>
<td>DPINT (1.4)</td>
</tr>
<tr>
<td>CMPLTSP (1.1)</td>
<td>DPSP (1.4)</td>
</tr>
<tr>
<td>CMPEQDP (2.2)</td>
<td>INTDP (1.5)</td>
</tr>
<tr>
<td>CMPGTDP (2.2)</td>
<td>INTDPU (1.5)</td>
</tr>
<tr>
<td>CMPLTDSP (2.2)</td>
<td>SPINT (1.4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>'.S Unit'</th>
<th>'.L Unit'</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPDP (1.2)</td>
<td>SPTRUNC (1.4)</td>
</tr>
<tr>
<td>ABSDP (1.4)</td>
<td>SUBSP (1.4)</td>
</tr>
<tr>
<td>CMPLTSP (1.2)</td>
<td>SUBDP (2.7)</td>
</tr>
<tr>
<td>CMPEQDP (2.2)</td>
<td></td>
</tr>
<tr>
<td>CMPGTDP (2.2)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>'.M Unit'</th>
<th>'.D Unit'</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPYSP (4.10)</td>
<td>ADDAD (1.1)</td>
</tr>
<tr>
<td>MPYDP (4.10)</td>
<td>LDDW (1.5)</td>
</tr>
<tr>
<td>MPYI (4.9)</td>
<td></td>
</tr>
<tr>
<td>MPYID (4.10)</td>
<td></td>
</tr>
</tbody>
</table>

As mentioned earlier, only the double-precision instructions actually require extra functional-unit latencies … Why is this?

The 'C67x has an enhanced 16-bit multiplier. To generate a double-precision multiplication values requires the CPU to reuse the multiplier hardware for multiple passes. While this requires extra cycles to complete, you benefit from lower cost – and you don't have to mess with an extra algorithm to create double-precision results! This is the quickest way to double-precision available today!
Optional Topics

**VelociTI vs. Standard VLIW**

VLIW (Very Long Instruction Word) architectures have many advantages including C compiler efficiency, excellent multi-tasking capabilities, and parallelism (i.e. increased performance). However, one major disadvantage is code size, and in some applications, performance can also suffer. TI has developed an enhanced VLIW architecture (VelociTI) that retains the advantages and minimizes the disadvantages. This discussion will help compare and contrast the differences between standard VLIW and VelociTI architectures in terms of code size and performance.

Let’s start with an explanation of the standard VLIW architecture and compare it to VelociTI™ in terms of fetch and execute packets…

Standard VLIW can be viewed as containing multiple hardware units assigned to different tasks. As shown below, VLIW processors typically contain ALU, MPY, Data and Program units. All units can be used in parallel if the “fetch packet” contains eight (in this example) parallel instructions, which do not require more than two of any unit type. If, for example, only two instructions are used in parallel, NOPs are executed in the unused units. This would be fine as long as the NOPs were *internally* generated and were not required to reside in expensive memory. However, standard VLIW requires each fetch packet to contain (in this example of using eight units) eight instructions that are all executed in parallel every cycle. But, what if the code only contains two parallel instructions (as shown in the example code below)? The answer, for standard VLIW, is that NOPs must reside in memory for all unused units in each fetch packet – taking up precious memory locations. In the example below, you can easily see that a significant number of NOPs fill up memory and therefore dramatically increase code size.

<table>
<thead>
<tr>
<th>Standard VLIW (FP = EP)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code Example</strong></td>
</tr>
<tr>
<td>B .S1</td>
</tr>
<tr>
<td>MVK .S2</td>
</tr>
<tr>
<td>ADD .L1</td>
</tr>
<tr>
<td>ADD .L2</td>
</tr>
<tr>
<td>MPY .M1</td>
</tr>
<tr>
<td>MPY .M1</td>
</tr>
<tr>
<td>LDH .D1</td>
</tr>
<tr>
<td>LDW .D2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Standard VLIW Units (FP = EP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU ALU MPY MPY DAT DAT PRG PRG</td>
</tr>
<tr>
<td>NOP NOP NOP NOP NOP NOP B MVK</td>
</tr>
<tr>
<td>ADD ADD MPY NOP NOP NOP NOP NOP</td>
</tr>
<tr>
<td>NOP NOP NOP MPY LDH LDW NOP NOP</td>
</tr>
<tr>
<td>ADD ADD MPY MPY LDH LDW B MVK</td>
</tr>
</tbody>
</table>

**VelociTI (FP ≠ EP)**

<table>
<thead>
<tr>
<th>VelociTI (FP ≠ EP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD ADD MPY MPY LDH LDW B MVK</td>
</tr>
</tbody>
</table>

FP = Fetch Packet
EP = Execute Packet
To address this limitation, TI's VelociTI architecture allows smaller packets of parallel instructions (called execute packets) to be contained within fetch packets thus reducing the number of NOPs required in memory. As you can see, the example code would require 4 256-bit instructions in standard VLIW versus only 1 fetch packet (i.e. 8 32-bit instructions) using VelociTI. For standard VLIW, a fetch packet is the same as an execute packet. VelociTI’s answer is to treat execute packets separate from fetch packets.

**Note:** VLIW defines instruction as the entire 256-bit quantity. Contrast that with VelociTI, it defines the term instruction as it traditionally is defined in non-VLIW processors: as the operation for each individual functional unit (32-bits). Of course, this left TI wondering what to call the whole 256-bit group of instructions fetched together; hence the term fetch packet was created. And then, since an entire fetch packet doesn’t have to be executed at one time, TI needed to create a name for the VelociTI instructions that are executed together; hence, execute packet.

Let’s pause for a moment and look at some basic definitions:

**VelociTI (FP ≠ EP)**

**Definitions**
- Fetch Packet: 8 32-bit instr (256 bits)
- VLIW: Very Long Instr Word (256 bits)
- EP: Execute Packet (group of || instr)
- Instruction: 32-bit opcode
- VelociTI: TI’s VLIW Architecture w/EP’s
The greatest advantage of VelociTI over standard VLIW is code size. Imagine a program that uses serial instructions. Standard VLIW would require seven NOPs for every “real” instruction. However, VelociTI would be able to pack eight “useful” instructions into one fetch packet, thereby reducing code size (up to 8:1) and the number of program fetches which results in less expensive memory costs and lower power consumption.

**VelociTI vs. Standard VLIW**

<table>
<thead>
<tr>
<th>Standard VLIW</th>
<th>VelociTI reduces code size up to 8:1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fewer program fetches</td>
</tr>
<tr>
<td></td>
<td>Less power consumption</td>
</tr>
<tr>
<td></td>
<td>Lower memory costs</td>
</tr>
</tbody>
</table>

vs.

TI’s VelociTI

Is performance actually increased when using VelociTI vs. standard VLIW? The answer is, it depends on the device’s architecture. If both architectures use on-chip program memory (and a 256-bit bus), the performance is the same. However, if program memory is off chip (using a 32-bit bus), and referring to our previous example, standard VLIW would require 24 fetch cycles vs. VelociTI’s eight. Hence, VelociTI, in this case, also increases performance. In contrast, if the eight instructions were all in parallel, there would be no performance differences.
Execute Packet Alignment

Now we’ve reached the issue of how VelociTI handles execute packets (EP's) crossing fetch packet (FP) boundaries. The first rule is, EP's cannot cross FP boundaries. If you look at the example below, the third execute packet (EP3) will not fit into the first fetch packet (FP1) and is therefore placed in the next FP. But what happens to the two 32-bit locations at the end of FP1? The tools automatically add NOPs (in parallel with the previous EP – in this case, EP2) to fill up FP1. Yes, this does add some extraneous NOPs, but this is a small price to pay for the overall decrease of code size vs. standard VLIW and, because the NOPs are in parallel with previous instructions, does not decrease performance. When you compile and link your code, look at the disassembly window (or absolute listing) to see how many and where these parallel NOPs exist. Typical % increase of code size due to “NOP-packing” is approximately 10-20%.

'C62x/67x VelociTI EP/FP Alignment

Code Example

<table>
<thead>
<tr>
<th>B</th>
<th>.S1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB</td>
<td>.L1</td>
</tr>
<tr>
<td>MVK</td>
<td>.S2</td>
</tr>
<tr>
<td>ADD</td>
<td>.L2</td>
</tr>
<tr>
<td>ADD</td>
<td>.L1</td>
</tr>
<tr>
<td>MPY</td>
<td>.M1</td>
</tr>
<tr>
<td>MPY</td>
<td>.M1</td>
</tr>
<tr>
<td>MPY</td>
<td>.M2</td>
</tr>
<tr>
<td>LDH</td>
<td>.D1</td>
</tr>
<tr>
<td>LDB</td>
<td>.D2</td>
</tr>
</tbody>
</table>

Execute packets cannot cross fetch packet boundaries

<table>
<thead>
<tr>
<th>B</th>
<th>SUB</th>
<th>MVK</th>
<th>ADD</th>
<th>ADD</th>
<th>MPY</th>
<th>NOP</th>
<th>NOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EP2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EP3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To align EP's within FP's, the tools add parallel NOPs
Optional Topics

Advanced Instruction Packing on the ‘C64x

The ‘C64x allows EPs to straddle fetch-packet boundaries. This eliminates the need for parallel-NOPs to pad between execute-packets. This is one of few new tricks the ‘C64x employs to minimize code size.

Alignment (C64x, C64x+, C672x)

Code Example

```
B  .S1
  |  SUB  .L1
  |  MVK  .S2
  |  ADD  .L2
ADD  .L1
  |  MPY  .M1
  |  MPY  .M2
  |  LDH  .D1
  |  LDB  .D2
```

Execute packets can cross fetch packet boundaries

Alignment (C64x, C64x+, C672x)

```
B  SUB  MVK  ADD  MPY  MPY  MPY
  EP1  EP2  EP3
ADD  EP1
  EP2
MPY  FP1
  FP2
MPY  EP3
LDB  EP3
LDH  Etc.
  .
  .
  .
  .
  .
  .
  .
  .
  .
  .
  .
```
Introduction

The dot-product function you benchmarked in Chapter 2 was compiled with great efficiency. As you will see in Chapter 9, the compiler can often achieve 100% efficiency; even when hand-coding assembly, we cannot do any better.

Even so, we’re going to use this simple function to practice writing in assembly and examine calling assembly functions from C. Without a thorough understanding assembly code – and thus the C6000 CPU architecture – it would be difficult to obtain the most efficient C code.

In the next chapter, we’ll explore using the assembly optimizer – which should make our jobs easier, if we ever have to delve into assembly in the real world.

Outline

Calling Functions From C (C Environment)

◆ Mixing C and Assembly
  • Why Mix Them?
  • Methods of Mixing C & Assembly
◆ C Function Implementation
  • C Callable Assembly Routine
  • Passing Arguments
  • Returning Results
◆ Writing In Assembly
  • Using C’s globals in ASM
  • Do I have to save any registers?
◆ Review
◆ (Optional) Lab 4
Chapter Topics

Calling Assembly from C................................................................................................................................. 4-1

Mixing C and Assembly.................................................................................................................................. 4-3
  Methods of Mixing C and Assembly.............................................................................................................. 4-3

Function Calls – Just follow the rules............................................................................................................. 4-4

Calling Assembly from C ............................................................................................................................... 4-5
  Finding the function....................................................................................................................................... 4-5
  Passing Arguments and Results..................................................................................................................... 4-6

Writing Assembly code.................................................................................................................................. 4-9
  Accessing C Global Variables (from Assembly)............................................................................................ 4-9
  One Last Rule – Do I have to save any registers?......................................................................................... 4-10
  Register Worksheet....................................................................................................................................... 4-11
Mixing C and Assembly

The best way to mix C and Assembly language routines is to call one from the other. Since most users prefer to work with C code as much as possible you’ll usually find programs calling Assembly routines from C (the title of this module).

Why would you even want to do this? Didn’t we already state that Assembly Language (on any processor) is more difficult to write than C?

Well then, what reasons can you come up with?

Methods of Mixing C and Assembly

1. Call assembly routine from C
   - Standard asm routine (Chap 4)
   - Linear asm function (cproc) (Chap 5)
   - Library function call (Chap 9)

2. Inline assembly (n/c)

3. Use intrinsic functions (Chap 9)
Function Calls – Just follow the rules

Once you’ve chosen to mix C and Assembly routines together we need to set forward some rules. Why do we need rules? Just like any interface, the two sides (in our case C and ASM) require a common method of passing information back and forth. Additionally, they also need to be respectful of how each other uses resources (i.e. registers) so that one doesn’t corrupt the others environment.

Calling an ASM Routine

- Coding interface requires a means of \texttt{handing-off} data and control info
- Since the compiler already exists, we will use the \texttt{compilers interface rules}

As stated above, if you want to use TI’s compiler you’ll have to follow their set of interface rules. Here’s a quick summary of the rules, we’ll be going through each one individually.

Calling assembly from C

- Use leading \_underscore to access C labels
- Pass Arguments
- Return Results

Assembly coding

- Using C’s globals in ASM
- Do I have to save any registers?

In a later chapter we’ll examine a couple more aspects of the C environment:

- How can you use/access the compilers stack?
- How can you optimize access to global variables?
Calling Assembly from C

Finding the function

First off, calling an assembly function from C means the Compiler will need to find your assembly function.

Accessing labels (or symbols, however you want to call them) between C and Assembly requires two things:

1. Whatever name is used in C must have a preceding underscore appended to it. Why? I’m not sure, but most compilers work this way.

2. As was described previously, you must declare global (inter-file) references.

```
C-Callable Asm Routine – Global Scope

//Parent.C
int child(int, int);
int x = 7, y, w = 3;
void main (void)
{
    y = child(x, 5);
}

//Child.C
int child(int a, int b)
{
    return(a + b);
}

;Child.ASM
.global _child
_child:
    ; end of subroutine
```

How do we pass arguments?
Passing Arguments and Results

Here are the Argument Passing Rules:

C Compiler Register Usage

Arguments are passed in registers as shown

- Return value in A4
- Return address in B3

//Child.C
int child(int a, int b)
{
    return(a + b);
}

The first argument is passed to the assembly subroutine in register A4 – argument \( a \), in this example. Similarly, the second argument (e.g. \( b \)) is passed to the assembly subroutine in register B4. The assembly subroutine returns the result value back via register A4. The return address gets stored in register B3.

In summary, the arguments are passed through the registers as shown:

- Argument 1 → A4
- Argument 2 → B4
- Argument 3 → A6
- Argument 4 → B6
- Argument 5 → A8
- Argument 6 → B8
- Argument 7 → A10
- Argument 8 → B10
- Argument 9 → A12
- Argument 10 → B12
- Return Value → A4
- Return Address → B3
Given these guidelines, solve the following example:

```c
//Child.C
int child(int a, int b)
{
    return(a + b);
}
```

```asm
;Child.ASM
.global _child
_child:
    add __, __, __
b __
nop 5
; end of subroutine
```

```c
//Parent.C
int child(int, int);
int x = 7, y, w = 3;
void main (void)
{
    y = child(x, 5);
}
```

```asm
_cid: add __, __, __, __, __
b __
nop 5
; end of subroutine
```
Here are the locations for each value passed to/from the child routine:

<table>
<thead>
<tr>
<th>To Child:</th>
<th>From Child:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a:</td>
<td>A4</td>
</tr>
<tr>
<td>b:</td>
<td>B4</td>
</tr>
<tr>
<td>Return Address:</td>
<td>B3</td>
</tr>
</tbody>
</table>

Result: A4

Did you get it correct?

Calling Assembly from C

//Parent.C
int child(int, int);
int x = 7, y, w = 3;
void main (void)
{
  y = child(x, 5);
}

//Child.C
int child(int a, int b)
{
  return(a + b);
}

;Child.ASM
.global _child
_child:
  add a4,b4,a4
  b b3
  nop 5
; end of subroutine

Arguments
Return/Result
Writing Assembly code

Accessing C Global Variables (from Assembly)

First of all, why would we want to declare all our global variables in C rather than assembly?

Well, because it’s easier. For starters, C is more natural for most programmers. Beyond this, if you specify an initial value the compiler’s initialization routine will do all the work for you. Otherwise you’d be stuck writing your own initialization routine.

Note: TI’s C compiler leaves uninitialized global/static variables undefined.

Accessing Global Variables from ASM

Complete the assembly code to load “w” into register A0.

Child.ASM

.C6000 Optimization Workshop - Calling Functions from C
One Last Rule – Do I have to save any registers?

There’s one more guideline to keep in mind when writing assembly language functions for C, you cannot corrupt registers A10 – A15 or B10 – B15. *Maybe we should put that another way; you can use the first 20 registers without having to worry about saving them first!* Doesn’t that sound better?

![Diagram of register preservation](image)

These must be saved and restored if you use them in Assembly

If your processor has 64 reg’s these do not need to be saved

If you are using the ‘C64x which has 32 registers per side, you can also use the lower 16 registers per side without needing to save/restore their context.

The compiler is responsible for saving any register (within the first 20 and/or the lower 32) that it’s using. That gives you free reign on them.

Why did our compiler writers pick the 20/12 split? They evaluated the performance of many different split points and found this to be the most efficient.
Register Worksheet

There isn’t anything profound with the following worksheet but it provides a handy method for keeping track of (assigning) register usage when writing in linear or standard assembly. We have provided a couple copies of this worksheet at the end of this module. Additionally, there is a Microsoft Word version of this worksheet (register worksheet.doc) along with the lab files.

| Register File | # | A0 | B0 | A1 | B1 | A2 | B2 | A3 | B3 | A4 | B4 | A5 | B5 | A6 | B6 | A7 | B7 | A8 | B8 | A9 | B9 | A10 | B10 | A11 | B11 | A12 | B12 | A13 | B13 | A14 | B14 | A15 | B15 |
|---------------|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Comments      |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Register Allocation Worksheet

- Registers A1, A2, B0-B2 can be used as conditional registers for all instructions.
- Registers A4-A7, B4-B7 can be used for circular addressing pointers.
- Registers A8-A15, B8-B15 must be saved by C-called assembly subroutines.
- Other references refer to the register usage by C Compiler.

*DP: Registers A1, B1, B15 can be used as conditional registers for all instructions.

Stack: Registers A10-A15, B10-B15 must be saved by C-called assembly subroutines.
This page left blank.
Using the Assembly Optimizer

Introduction

In the preceding chapters, we have discussed writing C and assembly code for the dot-product routine. These two tools are not the only choices for writing optimized code for the ’C6000. The other choice that we have is called Linear Assembly, which is similar to standard assembly. The Assembly Optimizer tool turns Linear Assembly into standard assembly with the help of just a few simple commands or directives. These directives will be covered here along with a lab exercise where you’ll code your dot-product in Linear Assembly.

Note, while it may seem unusual, the Linear Assembler is documented in the TMS320C6000 C Compiler’s Users Guide, not the Assembly Language Tools Users Guide. It seems odd, but the same folks who write the compiler also write the Assembly Optimizer.

Learning Objectives

Outline

- Using the Assembly Optimizer
- How to Write Linear Assembly
- Making Calls From Linear Assembly
- Build Options
- Lab 5 - Writing Linear Assembly
- Additional Topics
Chapter 5 Topics

Using the Assembly Optimizer ................................................................. 5-1

Writing Linear Assembly Code ............................................................... 5-3
Linear Assembly – Dot Product Example .............................................. 5-4
  Specifying Procedure to Optimize .................................................. 5-5
  Passing and Returning Arguments ............................................... 5-6
  Using Symbolic Names ................................................................. 5-7
  Complete Example .................................................................... 5-9
Calls in Linear Assembly .................................................................. 5-10

Invoking the Assembly Optimizer ...................................................... 5-12

Summary of Language Differences .................................................... 5-13

Additional Topics ........................................................................ 5-14
  Partitioning Linear Assembly .................................................. 5-14
  Viewing Memory and Endianess ............................................. 5-15
Writing Linear Assembly Code

The assembly optimizer allows you to write assembly code without being concerned with the pipeline structure of the 'C6000 or with assigning registers. It accepts *linear assembly* code, which is assembly code that has not been register-allocated and is unscheduled. The assembly optimizer assigns registers and uses loop optimization to turn linear assembly into highly parallel assembly.

When writing linear assembly code we follow a different path through the development tool flow:

![Software Tool Flow Diagram](attachment:diagram.png)

The next section details writing linear assembly code using the Dot-product example. First, here are a few basics regarding linear assembly files:

- Use "*.sa" extension to invoke the Assembly Optimizer.
- Only code within a procedure is optimized. The assembly optimizer copies any code that is outside of procedures to the output "*.asm" file.
- Linear assembly procedures can:
  - Pass parameters
  - Return results
  - Use symbolic variable names
  - Ignore pipeline issues (delay slots)
  - Automatically return to calling function
  - Call other functions (written in C or Linear Assembly)
Linear Assembly – Dot Product Example

Let’s begin the Linear Assembly example with the dot-product code from Chapter 1.

### Linear Assembly

- Don’t use NOP’s
- Don’t use parallel instructions (||)
- Not required: - Functional units
  - Registers not required

```assembly
_dotp:  zero       sum
loop:   ldh *pm++, m
        ldh *pn++, n
        mpy m, n, prod
        add prod, sum, sum
        sub count, 1, count
        [count] b loop
```

- Some Assembly Required
  (asm directives, that is)

The Assembly Optimizer understands each instruction, including the number of delay slots it requires. Therefore, you don’t need to include NOPs in Linear Assembly code. Basically, let the tools handle the NOPs and creating parallelism for you.

If desired, you can specify either the functional unit or specific registers — but please don’t put in the NOPs. Specifying either units or registers may restrict the tool’s abilities to optimize fully, although you may find some occasions where this can be used to help the optimizer improve your assembly code.
Specifying Procedure to Optimize

Two assembler directives specify the beginning and ending of linear assembly code. Only code residing between `.cproc` and `.endproc` is optimized. This gives you greater specificity of the optimizers’ scope.

```
_specp: .cproc

zero sum

loop:
    ldh *pm++, m
    ldh *pn++, n
    mpy m, n, prod
    add prod, sum, sum
    sub count, 1, count
    [count] b loop

.endproc
```

`.cproc` and `.endproc` define the ‘procedure’ (i.e. function) to be optimized
Passing and Returning Arguments

Since .cproc was designed to follow the C compiler function calling interface, it only stands to reason that you can pass and return arguments. Linear Assembly follows the same argument passing rules discussed during Chapter 4.

To make writing Linear Assembly easier, symbols may be used in place of register names. As mentioned earlier, this allows the Assembly Optimizer to select the most optimal register assignments.

But, if you’re using symbolic variables, how can you associate them with an incoming argument? The answer is simple; you can specify the arguments passed into the function on the same line as the .cproc command.

Passing/Returning Arguments

```
_dotp: .cproc pm, pn, count
       zero sum
loop:
   ldh  *pm++, m
   ldh  *pn++, n
   mpy  m, n, prod
   add  prod, sum, sum
   sub  count, 1, count
   [count] b loop
.return sum
.endproc
```

We want to implement this function:

```
int dotp (short *a, short *x, int count)
```

Returning a result simply requires using the .return assembler directive. The previous example returns “sum”. Per our earlier discussion, the Assembly Optimizer will put the final value of “sum” into register A4.
Using Symbolic Names

Using variable names (symbolic names) for data and pointers helps both you and the optimizer. For the programmer, it makes code easier to write … and later on, easier to read.

For the optimizer, it provides flexibility. It removes constraints that require data to reside in specific registers, therefore letting the tools allocate data and pointers into any register – in either register set – which provides the most efficiency.

The tools require that you specify the variable names (i.e. symbols) used in your procedure. This is done using the .reg assembler directive.

Using Symbolic Registers

```
_dotp: .cproc pm, pn, count
  .reg m, n, prod, sum
  zero sum
loop:
  ldh *pm++, m
  ldh *pn++, n
  mpy m, n, prod
  add prod, sum, sum
  [count] b loop
  .return sum
.endproc
```

Note: The symbols declared with .cproc (incoming arguments) do not need to be specified using .reg.
This page is supposed to be ________.
Complete Example

Here’s a look at the final dot-product code example.

**Linear Assembly - Summary**

```assembly
_dotp:  .cproc   pm, pn, count
        .reg    m, n, prod, sum
        zero    sum

loop:
    ldh   *pm++, m
    ldh   *pn++, n
    mpy   m, n, prod
    add   prod, sum, sum
    sub   count, 1, count
    [count]  b   loop

.return   sum
.endproc
```

This Linear Asm subroutine performs this function:

\[
\text{int dotp (short } \ast a, \text{ short } \ast x, \text{ int count )}
\]

Note, while you can use variables declared in C, remember that you are writing assembly language. What this means is that if you want to use a C variable called “m”, you first need to load a pointer with the value “pm” or some other name that contains the address of where m is located (think of it as &m). The pointer may then be used to access a C variable by using the following instructions:

```assembly
.global _cVar
.cproc
.reg   p, myVar
mvkl   _cVar, p
mvkh   _cVar, p
ldh     *p++, myVar
```

**Hint:** You’ll need to remember this tip to complete the Lab 5 optional exercise.
Calls in Linear Assembly

In Linear Assembly you are restricted from branching out of a procedure.

Why could this be bad? Without the ability to call a subroutine from within a procedure you would not be allowed to write an entire program in Linear Assembly (if you so desired).

The solution was to provide the .call assembler directive. This allows you to call any C compliant function – whether written in C or Linear Assembly (or even standard assembly).

Here’s a simple example:

```
prototype: int dotp(void)

_dotp: .cproc
.reg one, two
.mvk 5, one
.call two = _testcall(one)
.return two
.endproc

int testcall(int input)
{
    return(input + 5);
}
```
Here’s a linear assembly routine calling another linear assembly routine.

```
prototype: int dotp(void)

_dotp: .cproc
 .reg one, two
 mvk 5, one
 .call two = _testcall(one)
 .return two
.endproc

prototype: int testcall(int input)

_testcall: .cproc input
 add input, 5, input
 .return input
 .endproc
```
Invoking the Assembly Optimizer

Code Composer Studio recognizes Linear Assembly code by the file’s .sa extension and calls the Assembly Optimizer before invoking the assembler and linker.

Like the compiler, the Assembly Optimizer can be controlled with options. In fact, it uses the same options as the compiler. We suggest you debug code with a two step process, just as you did with C code:

1. First, debug with full symbolic debugging -- Debug build configuration (–g –s). This makes debugging (single-stepping) possible.

2. After the code is verified correct, enable the Assembly Optimizer to optimize your routine by using the same optimization switches as we used with the compiler.

The assembly optimizer uses the same build options as the C compiler

1. Use **Debug** build options
2. After verification, build with **Release**

Following this process makes debugging much easier. As with C, after full optimization there may not be a one-to-one correlation between .SA and .ASM code. This two step procedure makes it much easier.
## Summary of Language Differences

### Language Tradeoffs

<table>
<thead>
<tr>
<th></th>
<th>C/C++</th>
<th>Assembly</th>
<th>Linear Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register (A/B) Partitioning</td>
<td>Automatic</td>
<td>Manual</td>
<td>Optional</td>
</tr>
<tr>
<td>Register Allocation</td>
<td>Automatic</td>
<td>Manual</td>
<td>Optional</td>
</tr>
<tr>
<td>Functional Unit Allocation</td>
<td>Automatic</td>
<td>Manual</td>
<td>Automatic</td>
</tr>
<tr>
<td>Instruction Scheduling</td>
<td>Automatic</td>
<td>Manual</td>
<td>Automatic</td>
</tr>
<tr>
<td>Software Pipelining</td>
<td>Automatic</td>
<td>Manual</td>
<td>Automatic</td>
</tr>
<tr>
<td>Ease of Use</td>
<td>Easy</td>
<td>Hard</td>
<td>In-between</td>
</tr>
<tr>
<td>Performance</td>
<td>Good</td>
<td>Best</td>
<td>Good</td>
</tr>
<tr>
<td>Portability Across Compiler Releases</td>
<td>Good</td>
<td>Best</td>
<td>Ok</td>
</tr>
</tbody>
</table>
### Additional Topics

**Partitioning Linear Assembly**

#### Specifying Registers with `.map`

<table>
<thead>
<tr>
<th>_d:</th>
<th><code>.cproc</code> pm, pn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>.reg</code> m, prod, sum</td>
</tr>
<tr>
<td></td>
<td><code>.reg</code> n</td>
</tr>
<tr>
<td>lp:</td>
<td><code>ldh</code> *pm++, m</td>
</tr>
<tr>
<td></td>
<td><code>ldh</code> *pn++, n</td>
</tr>
<tr>
<td></td>
<td><code>mpy</code> m, n, prod</td>
</tr>
</tbody>
</table>

- `.map` specifies that Asm Optimizer uses a specific reg for a given symbol.
- `.map` declarations are only local to `.cproc` procedures.
- Explicitly clear all register specifications using `.clearmap`.
- Note: In `.map` example, A8 can hold other data if n is not live.

#### Less Restrictive Partitioning

<table>
<thead>
<tr>
<th>_dotp:</th>
<th><code>.cproc</code> pm, pn, count</th>
</tr>
</thead>
<tbody>
<tr>
<td>_dotp:</td>
<td><code>;</code></td>
</tr>
<tr>
<td>_dotp:</td>
<td><code>.reg</code> m, n, prod, sum</td>
</tr>
<tr>
<td>_dotp:</td>
<td><code>.map</code> sum/A4</td>
</tr>
<tr>
<td>_dotp:</td>
<td><code>.pref</code> prod/B4/B6/B9</td>
</tr>
<tr>
<td>_dotp:</td>
<td><code>.rega</code> m</td>
</tr>
<tr>
<td>_dotp:</td>
<td><code>.regb</code> n</td>
</tr>
</tbody>
</table>

- `.rega` or `.regb` specifies a register should be allocated on one side or another.
- `.pref` expresses a register preference, but not a requirement.

---

Notice, code doesn't change, when using `.map`
Viewing Memory and Endianess

Viewing Memory

Given:
short table_a[40] = {0x0028, 0x0027, ..., 0x0001};
short table_x[40] = {0x0001, 0x0002, ..., 0x0028};

Why does “table_a” look scrambled?

Memory View Window

Click on icon to open Property Page Manager
How CCS Displays Values in Memory

Byte Address: 203 202 201 200

00 27 00 28

Little Endian and Casting Memory

Given: short table_a[40] = {0x0028, 0x0027, ..., 0x0001};

Little Endian means LSB is stored first into memory
How can we best view “shorts”?

Given: short table_a[40] = {0x0028, 0x0027, ..., 0x0001};

<table>
<thead>
<tr>
<th>200</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>201</td>
<td>20</td>
</tr>
<tr>
<td>202</td>
<td>30</td>
</tr>
<tr>
<td>203</td>
<td>40</td>
</tr>
</tbody>
</table>

With Memory window format set to:
16-bit Hex - TI Style

Little vs. Big Endian

Given: int x = 0x40302010;

Little Endian

<table>
<thead>
<tr>
<th>200</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>201</td>
<td>20</td>
</tr>
<tr>
<td>202</td>
<td>30</td>
</tr>
<tr>
<td>203</td>
<td>40</td>
</tr>
</tbody>
</table>

Big Endian

<table>
<thead>
<tr>
<th>200</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>201</td>
<td>30</td>
</tr>
<tr>
<td>202</td>
<td>20</td>
</tr>
<tr>
<td>203</td>
<td>10</td>
</tr>
</tbody>
</table>
How CCS Displays Big Endian

Given:

\[ \text{int } x = 0x40302010; \]

Notice, the byte ordering change (This is how CCS does it)

Big Endian

\[
\begin{array}{cccc}
200 & 40 & 30 & 0 \\
201 & 30 & 40 & 1 \\
202 & 20 & 10 & 2 \\
203 & 10 & 20 & 3 \\
\end{array}
\]

x8 (be)

To use Big Endian:
- -me (compiler)
- rts6400e.lib (library)
- -me (ccs debugger)

How Hardware Determines Endianness

Given:

\[ \text{int } x = 0x40302010; \]

Little Endian

\[
\begin{array}{cccc}
200 & 10 & 20 & 0 \\
201 & 20 & 40 & 1 \\
202 & 30 & 30 & 2 \\
203 & 40 & 10 & 3 \\
\end{array}
\]

x8 (le)

x16 (le)

C6701

LENDIAN

0 = Big
1 = Little

CSR

EN

8

- LENDIAN pin is only sampled at reset
- The CSR<sub>8</sub> bit is read-only
- The Little Endian pin name varies slightly from device-to-device
- Some devices are hard-coded to little endian – see datasheet
Introduction

In the first couple chapters we discussed assembly language along with the architecture and pipeline. In the last two chapters you wrote an assembly language subroutine that was called as a function. Before going further, let’s cover a few additional architecture (i.e. assembly language) details. Specifically, details of pointer addressing and program control. This chapter concludes with a couple memory addressing exercises.

Outline

- Data types and alignment
  - C6000 Data Types
  - Data Alignment
  - Data Alignment Exercise
  - Upcoming Changes: EABI, C66x
- Data operands (registers/constants)
- Cross paths
- Pointer Operands
- Program control
- (Optional) Exercises
- (Optional) Appendix
Chapter Topics

Architecture Details .......................................................................................................................... 6-1

Data types and alignment .................................................................................................................. 6-3
C Data Types ..................................................................................................................................... 6-3
Alignment .......................................................................................................................................... 6-4
Endian Ordering/Mode ....................................................................................................................... 6-7
Alignment Exercise .......................................................................................................................... 6-8
Upcoming Changes for EABI and C6600 ....................................................................................... 6-9

Data Operands (registers/constats) ................................................................................................... 6-11
Cross Paths ...................................................................................................................................... 6-12
Data Cross-Paths ............................................................................................................................... 6-13
Address Cross Paths ......................................................................................................................... 6-15
Conditional Cross Paths .................................................................................................................. 6-18
Cross Paths – Summary .................................................................................................................... 6-18
Cross Paths – Review ......................................................................................................................... 6-19
Data-Path Summary ......................................................................................................................... 6-20

Pointer Operands ............................................................................................................................... 6-22
Indexing and Pre-Offset .................................................................................................................... 6-23
Element vs. Byte Displacement ....................................................................................................... 6-24
Pointer Indexing/Offset Summary ..................................................................................................... 6-25

Program Control ............................................................................................................................... 6-26
Subroutines - Call and Return ........................................................................................................... 6-26
C6000 Branch Instructions ................................................................................................................ 6-27
C64x/C64x+ Branch Instructions ....................................................................................................... 6-30

Pointer Indexing/Offset Exercises .................................................................................................... 6-34

(Optional) Additional Logical / Bitfield Instructions ....................................................................... 6-35
Data types and alignment

C Data Types

OK, so discussing data types isn’t a sexy topic. It is an important topic, though, if you’re interested in achieving maximum performance.

`C6000 C Data Types`

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>ASCII</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>Binary, 2’s complement</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>Binary, 2’s complement</td>
</tr>
<tr>
<td>long</td>
<td>40</td>
<td>Binary, 2’s complement</td>
</tr>
<tr>
<td>long long</td>
<td>64</td>
<td>Binary, 2’s complement</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>IEEE 32-bit</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>IEEE 64-bit</td>
</tr>
<tr>
<td>long double</td>
<td>64</td>
<td>IEEE 64-bit</td>
</tr>
<tr>
<td>pointers</td>
<td>32</td>
<td>Binary</td>
</tr>
</tbody>
</table>

Here are a few short rules to keep in mind regarding C data types on the C6000:

1. Use short types for integer multiplication. As with most fixed-point DSPs, our ‘C62x devices use a 16x16 integer multiplier. If you specify an int multiply, a software function in the runtime support library will be called. (Note, the ‘C67x devices do have a 32x32 multiply instruction, MPYID.)

2. Use int types for counters and indexes. As we examine during the next chapter, all registers and data paths are 32-bits wide.

3. Avoid accidentally mixing long and int variables. Many compilers allocate 32-bits for both types, thus some users interchange these types. The ‘C6000 allocates longs at 40-bits to take advantage of 40-bit hardware within the CPU. If you mix types, the compiler may be forced to manage this – which will most likely cost you some performance. (Note, this is changing with ELF/EABI, as explained during a later topic in this chapter.)

Why 40-bits? The extra 8-bits are often used to provide headroom in integer operations. Also, they can act like an 8-bit “carry bit”.

4. On ‘C67x devices, 32-bit float operations are performed in hardware. The ‘C6000 supports IEEE 32-bit floating-point math.

5. The double precision floating-point hardware supports IEEE 64-bit floating-point math.

6. Pointers, at 32-bits, can reach across the entire ‘C6000 memory-map.
### Data types and alignment

#### Alignment

<table>
<thead>
<tr>
<th>Data Alignment in Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DataType.C</strong></td>
</tr>
<tr>
<td>char z = 1;</td>
</tr>
<tr>
<td>short x = 7;</td>
</tr>
<tr>
<td>int y;</td>
</tr>
<tr>
<td>double w;</td>
</tr>
<tr>
<td>void main (void)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>y = child(x, 5);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte (LDB) Boundaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Short (LDH) Boundaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Int/Float (LDW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Double (LDDW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Each unique data type uses a different load instruction to load memory efficiently and accurately. To load a character, the load byte (LDB) instruction is used. It assumes everything loaded with this instruction will fall on a byte boundary. If a load short (LDH) instruction follows, it skips a byte to realign itself on a halfword or 16-bit boundary and loads 16-bits of information for each variable. The next instruction we see is a load word (LDW) instruction. This aligns everything on a word or 32-bit boundary. In this example, no memory space has to be skipped since it fell on a word boundary automatically. The load double word (LDDW) instruction loads variables on a double word or 64-bit boundary.
Alignment of Arrays & Structures

Data Alignment in Memory

```
DataType.C
char z = 1;
short x = 7;
int y;
double w;
void main (void)
{
    y = child(x, 5);
}
```

Double (LDDW)

```
7       
8  
C  
F
```

The top of Arrays are always aligned on larger type boundaries
- 32-bit boundary for C62x & C67x
- 64-bit boundary for C64x, C64x+ & C674x
- 128-bit boundary for C66x

Alignment of Structures

- Structures are aligned to the largest type they contain
- For data space efficiency, start with larger types first to minimize holes
- Arrays within structures are only aligned to their typesize
**Forcing Alignment**

Default boundaries can be altered by overriding the compiler with a pragma instruction. A pragma is simply a message written into the source code that tells the compiler to compile the program in a different way than the default method. The pragma below simply aligns the variable \( x \) on a 4 byte boundary instead of the default 2 byte boundary for short variables.

```plaintext
#pragma DATA_ALIGN(x, 4)
short z;
short x;
```

---

**Aligning Structures**

We need to look at alignment of structures in two ways. (1) You can align the top of a struct with:

```plaintext
#pragma STRUCT_ALIGN( type, constant expression );
```

typedef struct st_tag{
    int a;
    short b;
} st_typedef;

#pragma STRUCT_ALIGN( st_tag, 128);

- Forces the alignment of the entire structure
- It does not align structure elements

Looking at an alignment exercise...
Aligning elements within a structure is more difficult, as there isn’t a #pragma to help with this. Here are a couple of options to force elements to a specific alignment boundary.

### Forcing Alignment within Structures

While arrays are aligned to 32 or 64-bit boundaries, arrays within structures are not, which might affect optimization.

Here are a couple ideas to force arrays to 8-byte alignment:

1. **Use dummy variable to force alignment**
   
   ```
   typedef struct ex1_t{
      short b;
      long long dummy1;
      short a[40];
   } ex1;
   ```

2. **Use unions**

   ```
   typedef union ex2_t{
      short a2[80];
      long long a8[10];
   } ex2;
   ```

### Endian Ordering/Mode

Endian ordering is an addressing protocol in which bytes are numbered. There are two types of endian modes or orderings; big endian and little endian. The endian mode is hardware-specific and is determined at reset.

Little endian is an addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Big endian is the opposite of little endian and numbers the bytes from left to right within a word. The more significant byte in a word of big endian order has a lower number for its address.

The C6000 defaults to little-endian format. Endian format affects the way the MPY operation works, HPI data accesses, and how the C compiler produces code. The C compiler produces code in little-endian format by default, but the compiler option (–me) can be used to produce code in big-endian format. Bit 8 of the CSR (Control Status Register) controls the endian mode for the C6000. At default, it is set to 1, which is the default for little endian mode. To change to big endian mode, a value of 0 would need to be loaded into bit 8 of the CSR. Also, specific bits in the HPI registers determine the endian order for data transfers. Again the default is little endian mode. More information about the HPI can be found in the TMS320C6000 Peripheral Reference Guide.
Alignment Exercise

Data Access Exercise

<table>
<thead>
<tr>
<th>A0</th>
<th>Pointer aligned?</th>
<th>A1</th>
</tr>
</thead>
<tbody>
<tr>
<td>800h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>803h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>801h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>801h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Non-aligned accesses on ‘C64x

- LDNW STNW
- LDNDW STNDW

- Only caveat, cannot put another load/store in || with non-aligned access
- Example at end of Ch 7

Note: This “bad example” was created to demonstrate the mechanics of data alignment only. This type of data alignment issues will never happen in C. The compiler is smart enough to handle alignments. These "bad" alignments only happen in assembly if you force them to happen in assembly.
Upcoming Changes for EABI and C6600

EABI : ELF ABI
◆ Starting with v7.2.0 the C6000 Code Gen Tools (CGT) will begin shipping two versions of the Linker:
  1. COFF: Binary file-format used by TI tools for over a decade
  2. ELF: New binary file-format which provides additional features like dynamic/relocatable linking
◆ You can choose either format
  • v7.2.0 default will become ELF (prior to this, choose ELF for new features)
  • Continue using COFF for projects already in progress using "--abi=coffabi" compiler option (support will continue for a long time)
◆ Formats are not compatible
  • Your program's binary files (.obj, .lib) must all be built with the same format
  • If building libraries used for multiple projects, we recommend building two libraries – one with each format
◆ Migration Issues
  • EABI long's are 40 bits; new TI type (__int40_t) created to support 40 data
  • COFF adds a leading underscore to symbol names, but the EABI does not
  • See: http://processors.wiki.ti.com/index.php/C6000_EABI_Migration

C66x : New __float2_t Type
Recommendations on the use/non-use of the "double" type
◆ In order to better support packed data compiler optimizations in the future, the use of the type "double" for *packed data* is now discouraged and its support may be discontinued in the future. ("double" support is NOT going away!)
◆ Changes do NOT break compatibility with older code (source files or object files).
◆ Recommendations:
  • long long: Should be used for 64-bit packed integer data
  • double: Should only be used for double-precision floating point values.
  • __float2_t: Holds two floats; use instead of double for holding two floats.
◆ Intrinsics (intrinsics are discussed more chapter 9):
  • There are new __float2_t manipulation intrinsics (see below) that should be used to create and manipulate objects of type __float2_t.
  • C66 intrinsics with packed float data are now declared using __float2_t instead of double.
  • When using any intrinsic that involves __float2_t, c6x.h must be included.
  • Certain intrinsics that used double to store fixed-point packed data have been deprecated. They will still be supported in the near future, but their descriptions will be removed from the compiler user’s guide (spru187). Use the long long versions instead.
    Deprecated: _mpy2, _mpyh, _mpyl, _mpysu4, _mpyu4, and _smpy2.
C66x : 128-bit

- C66x adds 128-bit data type
  - Needed for certain SIMD operations on C6600 (i.e. quad-16x16 multiplies)
  - New container type for storing 128-bits of data: __x128_t
  - Objects of this type are aligned to a 128-bit boundary in memory
  - Compiler provided header file defines new type: c6x.h
  - This type may be used only when compiling for C66x (-mv6600), available starting CGT v7.2
  - Compiler loads __x128_t object into four registers (a register quad)

- The following operations are supported:
  - Declarations:
    local, global, pointer, array, member of a struct, class, or union
  - Assign a __x128_t object to another __x128_t object
  - Pass to function – or use as return value (Pass by value)
  - Use 128-bit intrinsics to set and extract contents (see list below)

128-bit Type : Supported / Not-Supported

- The following operations are supported:
  - Declarations:
    local, global, pointer, array, member of a struct, class, or union
  - Assign a __x128_t object to another __x128_t object
  - Pass to function – or use as return value (Pass by value)
  - Use 128-bit intrinsics to set and extract contents (see list below)

- The following operations are not supported:
  - Native-type operations, such as +, -, *, etc
  - Cast an object to a __x128_t type
  - Access the elements of a __x128_t using array or struct notation
  - Pass a __x128_t object to I/O functions like printf. Instead, extract the values from the __x128_t object by using appropriate intrinsics.

See more information about these types in the C Compiler user’s guide, as well as the release notes for each version of the Compiler (starting with v7.0.0).
Data Operands (registers/constants)

It’s now time to discuss some details about instruction operands. We’ve experienced some of the basics, but it is important to understand the limitations as well. The data paths available on the C6000 and resulting cross paths force certain constraints on how operands are used.

Each instruction requires at least 2 and sometimes 3 operands. Regarding constant and register operands, one source and the destination must be a register. The other source can be a register or 5-bit constant. The C6000 also supports 40-bit registers as sources and destinations. Loads/stores require a slightly different set of rules, which will be covered shortly.

The preceding graphic discusses CPU’s basic data operands. Refer to the ‘C6000 CPU Reference Guide’ for a detailed description of each instruction and appropriate data operands.
Cross Paths

The operands within an instruction are constrained by the data paths in the CPU. It is important to understand these limitations as you write code. Being a register based Load/Store architecture; it’s obvious that registers are used for the source and destinations of most operations. The CPU has many data paths that allow register contents to be routed to/from the various functional units. These paths are free from restrictions while accessing or storing to the register set located on the same side of the CPU; i.e. the “1” units can freely access registers in the “A” register set and likewise with the “2” units and the “B” register set.

There are some restrictions, though, when functional units must access registers across the CPU; i.e. when a “1” unit must use a “B” register. These cross-path accesses can be categorized into three types.

Cross Paths

What is a cross path?
Register operands cross from one side to the other.

The C6000 supports three INDEPENDENT types of operand cross paths:

<table>
<thead>
<tr>
<th>Data:</th>
<th>Registers as data operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address:</td>
<td>LDW destination</td>
</tr>
<tr>
<td></td>
<td>STW source</td>
</tr>
<tr>
<td>Condition:</td>
<td>Unlimited use for [ ]</td>
</tr>
</tbody>
</table>
Data Cross-Paths

Data cross paths are associated with basic arithmetic instructions, such as MPY, ADD, SUB. Only one path exists from the A side to the B side (2x – cross to the 2 side); similarly, only one path exists from the B side to the A side (1x). Therefore, an execute packet can only contain one “cross” per direction.

Along with the diagram of the “A” side data path below we’ve included a little quiz. Will the lines of code listed below work, or not?

<table>
<thead>
<tr>
<th>Will these work? Why/why not?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD .L1x A0,A1,B1,A4</td>
</tr>
<tr>
<td>MPY .M1x B0,B1,A4</td>
</tr>
<tr>
<td>MPY .M1x A0,A1,B4</td>
</tr>
<tr>
<td>B .S2x A3</td>
</tr>
<tr>
<td>MPY .M1x A0,B1,A5</td>
</tr>
<tr>
<td>ADD .L1x A0,B1,A4</td>
</tr>
<tr>
<td>MPY .M1x A0,B1,A5</td>
</tr>
</tbody>
</table>

Yes or No?
Cross Paths

Data Cross Paths

1. This is OK since only one B register crosses over to the “1” side.

2. Multiply won’t work since only one cross-path is allowed per EP.

3. This multiply won’t work because the cross-path can only be a source, not a destination.

4. OK. Only one cross-path per direction.

5. Can’t have two cross-paths in one direction per EP – even though both the ADD and the MPY use the same operand (B1).

While this isn’t allowed on the ‘C62x or ‘C67x, it works on the ‘C64x. The ‘C64x allows a single cross-path value to be routed to (up to) 2 functional units per cycle.

C64x Cross-Path

<table>
<thead>
<tr>
<th>1-cycle Delay</th>
<th>No Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD ( .S1 ) A0, A0, A1</td>
<td>ADD ( .S1 ) A0, A0, A1</td>
</tr>
<tr>
<td>ADD ( .S2X ) A1, B0, B1</td>
<td>ADD ( .S2X ) A0, B0, B1</td>
</tr>
</tbody>
</table>

 Advantages
- allows cross to all units
- allows source to two units

 Disadvantages
- Due to high performance, a one cycle delay is added when an “updated” register value crosses the processor

 H/W pipeline protected:
- delay is inserted by hardware, thus no additional NOP needed
- maintains ‘C62x compatibility
- all ‘C62x code works, although timing may be affected

† C64x and above
Address Cross Paths

Review of On-Chip Buses

Before examining address cross-paths, let’s review the on-chip buses connected to the CPU. As shown below, each side of the processor has its own bus structure for accessing memory. The T1 bus is connected to the “A” registers. The T2 transfers data in/out of the “B” register set. While each bus structure is limited to moving data in-to or out-of a register located on it’s own side, address may come from the opposite .D unit.

Using address cross-paths

A second cross path exists which is independent of the data cross path. The address cross paths are used when performing load/store operations using pointers (addresses). Let’s start off with a simple example and then migrate to more difficult scenarios:

In this example, the pointers and registers are used on the same side. We’ve seen this before and no difficulties are indicated.

Notice the nomenclature:

- An “A” pointer requires the instruction to be on the .D1 unit.
- The “A” destination register means that the T1 bus must be used.
**Loading to Either Side**

As you can see in this example, the unit and pointer register must come from the same side. However, the destination register can use either side. If a register from side B is loaded by a pointer in side A, this uses a load/store cross path. In this example, the address contained in A0 is passed to the DA2 bus (data address 2) and loads B5. No other instructions in this execute packet can use this resource.

```
<table>
<thead>
<tr>
<th>T1 - Data</th>
<th>A5</th>
</tr>
</thead>
<tbody>
<tr>
<td>.D1</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>*A0</td>
<td></td>
</tr>
<tr>
<td>LDW.D1T1</td>
<td>*A0,A5</td>
</tr>
<tr>
<td>LDW.D1T2</td>
<td>*A0,B5</td>
</tr>
<tr>
<td>T2 - Data</td>
<td>B5</td>
</tr>
<tr>
<td>.D2</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>
```

**Standard Parallel Loads**

In this example, two loads are performed in one execute packet and the pointer registers and units come from the same side. This is legal because there is no duplication of resources. All CPU buses (Data1, DA1, DA2, Data2) are being fully utilized.

```
<table>
<thead>
<tr>
<th>T1 - Data</th>
<th>A5</th>
</tr>
</thead>
<tbody>
<tr>
<td>.D1</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>*A0</td>
<td></td>
</tr>
<tr>
<td>LDW.D1T1</td>
<td>*A0,A5</td>
</tr>
<tr>
<td>LDW.D1T2</td>
<td>*A0,B5</td>
</tr>
<tr>
<td>T2 - Data</td>
<td>B5</td>
</tr>
<tr>
<td>.D2</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>*B0</td>
<td></td>
</tr>
<tr>
<td>LDW.D2T1</td>
<td>*B0,B5</td>
</tr>
</tbody>
</table>
```

Parallel Loads Using Address Cross Paths

This example also shows two loads occurring in parallel (they’re in same execute packet), but the register operands and destinations come from differing sides. This is legal because each cross path (address from one side is loading a register from the other side) is used without duplication of resources. The address contained in A0 is transferred to the B side using T2. D1 generates an address and loads the value contained at that address into B5.

Quiz

Fill the blanks ... Does this work?
Conditional Cross Paths

But, what about conditional instructions? If a register from the opposite side is used to hold the conditional value, is this considered a “cross path”. The answer is no, because the value in the register does not actually cross from one side to the other – in other words, it is not an operand, but simply affects the execution of the instruction. As we’ve learned before, all C6000 instructions can be conditional. It is legal to use a conditional register from a side different from the chosen unit. For example, a B-side register can be used with an instruction using an A-side unit. There are no limitations to the number of instructions with miss-matched conditional-register/unit pairs.

Conditionals Cross Paths

- If conditional register comes from the opposite side, it does NOT use a data or address cross-path
- Examples:
  [B2]  ADD .L1x A2, B0, A4
  [A1]  LDW .D2 *B0, A5

Cross Paths – Summary

Cross Paths - Summary

- Data
  - Destination register - same side as unit
  - Source register - one crosspath per cycle per side
  - C64x: Crosspath transfer can go to 2 units
  - C66x: Allows long long (or double) crosspath
  - Crosspath transfer can go to all 4 units
- Address
  - Pointer must be on same side as unit
  - Data can be transferred to/from either side
  - Parallel accesses: both cross or neither cross
- Conditional
  - Unlimited
Cross Paths – Review

Consider the following code. Which cross paths are being used? Will this code assemble properly?

To simplify the problem, fill in whether each instruction contains:

- Data or Address crosspath (or none)
- If a data crosspath, which direction does it go 1→2 or 2→1?
- If an address crosspath, does it use the T1 or T2 bus?
- By the way, do the numerous uses of cross-path registers as conditionals count?

### Cross Paths - Code Review

Which cross paths are being used in this example?

<table>
<thead>
<tr>
<th></th>
<th>D/A</th>
<th>Dir/T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[B1]</td>
<td>ADD</td>
<td>.L1x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[B1]</td>
<td>MPY</td>
<td>.M1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[A2]</td>
<td>ADD</td>
<td>.L2x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>![B1]</td>
<td>LDW</td>
<td>.D1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Will this code assemble? Why/why not?

The answer is … no. There is one conflict. Both the ADD.L2x and B.S2 use the register crosspath from the A → B side. (Conditional registers have no bearing in this execute packet.)

You must rewrite your code to eliminate this conflict.

Note: If you branch to a label rather than to the address contained in a register (B.S2 label) you’ll avoid the register cross-path conflict. The Branch instruction is detailed at the end of this chapter.
Data-Path Summary

Taken from the CPU Reference Guide, these three slides summarize the data-paths available with the C6000. These diagrams indicate the ports into and out of the register file along with the register and source operands going to each functional unit.

What changes are evident between the two data-path diagrams?

An additional 32-bit input bus was added for the MSB's of the LDDW instruction. This bus is shared with the 8-bit .L and .S destination bus.
Here is the 'C64x data-path. Notice that it also has two 32-bit inputs into the register file to support LDDW's. A change that was made with the 'C64x is the two 32-bit outputs from the register file. These outputs no longer share 8-bits with a long source or destination as they used to. This change allows the 'C64x to do two 64-bit stores (STDW) in parallel with a long operation.

'C64x Data-Path Summary

C6600 now supports 64-bit buses for most data paths (except where called out above). Even the crosspath buses are now 64-bits wide. The .M units now support two 64-bit buses for each source (and the dest) to enable the 128-bit register quads – used for complex and vector math operations.
**Pointer Operands**

In previous chapters, we have covered the basics of pointer operands - loading pointers using MVKL/MVKH, auto-increment/decrement using ++, *pointer, etc. However, the pointer arithmetic's are much more flexible. The goal of this section is to cover the more subtle details of using pointer operands in preparation for writing the dot-product routine.

**Pointers (Review)**

- After incrementing by 1, we need to jump four elements
- Is ADD the best way to do this?

Shown below is a table that describes the options available for pointer operands. As you can see, all possibilities of increment/decrement with/without modification are available. The displacement [disp] can be either a register or a 5-bit constant (B14/B15 support 15-bit constants). Typically, the modification is performed in a linear fashion (either increment or decrement). However, certain registers can be used to perform modulo or circular addressing.

**Indexing Pointers**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>Pointer Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>*R</td>
<td>Pointer</td>
<td>No</td>
</tr>
<tr>
<td>*+R[disp]</td>
<td>+ Pre-offset</td>
<td>No</td>
</tr>
<tr>
<td>-*R[disp]</td>
<td>- Pre-offset</td>
<td>No</td>
</tr>
<tr>
<td>*++R[disp]</td>
<td>Pre-increment</td>
<td>Yes</td>
</tr>
<tr>
<td>*--R[disp]</td>
<td>Pre-decrement</td>
<td>Yes</td>
</tr>
<tr>
<td>*R++[disp]</td>
<td>Post-increment</td>
<td>Yes</td>
</tr>
<tr>
<td>*R--[disp]</td>
<td>Post-increment</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- [disp] specifies # elements - size in DW, W, H, or B
- disp = R or 5-bit constant
Indexing and Pre-Offset

Using pointer operands as previously described, a pointer can be simply modified by incrementing (++) or decrementing (– –) by one “data type”. If desired, however, indexing allows the programmer to modify the chosen pointer by a value more than one either by using a constant or the value contained in another register.

For example, you might want to add one, two, and five. Simply use the desired value within brackets – indicating “data type” modification – to “bump” the pointer by that amount. Using the indexing feature saves code as seen in the above comparison.

The programmer can also choose to access a memory location that is an offset from a base pointer. This is called pre-offset. This capability is useful for structures where the base pointer is initialized to the beginning of the structure and all elements within the structure are accessed by an offset based on the base pointer. See the example below.

For example, you might want to add one, two, and five. Simply use the desired value within brackets – indicating “data type” modification – to “bump” the pointer by that amount. Using the indexing feature saves code as seen in the above comparison.

The programmer can also choose to access a memory location that is an offset from a base pointer. This is called pre-offset. This capability is useful for structures where the base pointer is initialized to the beginning of the structure and all elements within the structure are accessed by an offset based on the base pointer. See the example below.
Element vs. Byte Displacement

If the displacement is specified with parentheses ( ) the pointer is modified by the selected number of bytes.

<table>
<thead>
<tr>
<th>Scaled</th>
<th>[ ]</th>
<th>Element</th>
<th>*++R[disp]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Scaled</td>
<td>( )</td>
<td>Byte</td>
<td>*++R(disp)</td>
</tr>
</tbody>
</table>

Why do we cover ( ) displacement?

- ( ) isn’t common when assembly coding
- Compiler uses it extensively, though, so it’s helpful for reading interlist files
- Only LDNDW allows both scaled/non-scaled displacement

The scaled value is truncated by the assembler for all instructions except LDNDW. Thus, the assembled instruction looks like `LDW *++A0 [1], A5`. Note: The assembler issues a truncation warning when this occurs.
### Pointer Indexing/Offset Summary

#### Pointer Operands

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>Pointer Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>*R</td>
<td>Pointer</td>
<td>No</td>
</tr>
<tr>
<td>*+R[disp]</td>
<td>+ Pre-offset</td>
<td>No</td>
</tr>
<tr>
<td>*-R[disp]</td>
<td>- Pre-offset</td>
<td>No</td>
</tr>
<tr>
<td>*++R[disp]</td>
<td>Pre-increment</td>
<td>Yes</td>
</tr>
<tr>
<td>*--R[disp]</td>
<td>Pre-decrement</td>
<td>Yes</td>
</tr>
<tr>
<td>*R++[disp]</td>
<td>Post-increment</td>
<td>Yes</td>
</tr>
<tr>
<td>*R--[disp]</td>
<td>Post-increment</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- **[disp]** specifies # elements - size in DW, W, H, or B
  - [disp] = R or 5-bit constant

- *(disp)** specifies # bytes
  - (disp) = 5-bit constant
  - +B14/B15(disp) allows 15-bit constant
  - Assembler converts to element size [ ]
Program Control

In this section, we will show how to implement subroutines and discuss how the branch instruction works. In the upcoming lab, you will write the dot-product routine as a subroutine to be called from the main program you developed in chapter 5.

Subroutines - Call and Return

The basic process of implementing a subroutine involves two operations: call and return. During a call, the program counter for the next instruction to be executed is saved (on a stack, in a register, or in memory somewhere) and then a branch is made to the subroutine. At the end of the subroutine, a return is made, which places the saved address back into the program counter and then branches back to this address - i.e. a return from subroutine.

If you look in the instruction set, you will not find a CALL or RET instruction. These operations are actually performed by the same instruction - a branch.

Implementing a CALL on the C6000 similarly saves the address of the next instruction (in our example, the label Next) using MVKL and MVKH. We chose to store the return address into a register (e.g. A5 or B3 or any CPU data registers). After the address is saved, the calling routine branches to the subroutine.

At the end of the subroutine, a branch is executed, returning operation back to the Next instruction in the calling routine. How does the subroutine know where to return? It uses the return address previously stored in register (e.g. B3). Therefore, the subroutine simply branches to the address:

```c
B .S2 B3 ;branch to address contained in B3
```

The C6000 does not need call and return instructions since they can be implemented with branches. The decision to eliminate the unnecessary instructions reduces the hardware necessary to save the program counter and reduces the number of instructions. This RISC-like implementation reduces the levels of logic, which allows for greater instruction rates while lowering power and cost.

Note: The example shown here does not represent an optimized subroutine call and return. Optimization will be covered in later chapters.
C6000 Branch Instructions

Here are the details of the C6000 branch instructions:

1. **Relative Branch**
   - Label limited to $\pm 2^{20}$ offset
   - 21-bit field = PC – label
   - Used most of the time for loops

   $B$ label

2. **Absolute Branch**
   - Uses register, which can hold entire 32-bit address
   - Operates only on .S2 unit
   - Use when branching to functions that may be linked far away

   $B$ reg

What kind of branches does the C compiler use?

How C uses Branching

<table>
<thead>
<tr>
<th>C Syntax</th>
<th>Compiler Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>int dotp(a, x)</td>
<td>$B$ _dotp</td>
</tr>
<tr>
<td>far int dotp(a, x)</td>
<td>MVKL _dotp, reg</td>
</tr>
<tr>
<td></td>
<td>MVKH _dotp, reg</td>
</tr>
<tr>
<td></td>
<td>$B$ reg</td>
</tr>
</tbody>
</table>

- Linker *relocation* error occurs if relative branch is used when an absolute is needed
- Compiler option --trampolines provides to automatically solves this problem (starting with CCS 3.1 --trampolines is the default)
Far Call Trampolines

How it works:
- When using trampolines, the compiler always uses near calls
- Linker automatically changes near calls to reach far destinations, as needed

Advantages of trampolines?
- Avoids run-time penalty of far mode
- Potential code size improvement
- No user intervention
- Practical cases require few trampolines
### Branching/Trampoline Summary

<table>
<thead>
<tr>
<th></th>
<th>C Syntax</th>
<th>Compiler Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Relative Branch</strong></td>
<td>int dotp(a, x)</td>
<td>B _dotp</td>
</tr>
<tr>
<td><strong>Absolute Branch</strong></td>
<td>far int dotp(a, x)</td>
<td>MVKL _dotp, reg MVKH _dotp, reg</td>
</tr>
<tr>
<td></td>
<td>int dotp(a, x)</td>
<td>B reg</td>
</tr>
<tr>
<td><strong>Automatic</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>int dotp(a, x)</td>
<td>B _dotp</td>
</tr>
<tr>
<td></td>
<td>Compiler default: --trampoline</td>
<td>Uses trampoline if required</td>
</tr>
</tbody>
</table>

- In other words, the best method for picking branches in C is to let the tools handle it.
- Only if you know a call will be far, and need to save a couple cycles, should you modify your C code with the **far** keyword.
C64x/C64x+ Branch Instructions

Additional C64x/C64x+ Branches

- 'C64x CPU provides three additional branch instructions
  - BPOS Branch if positive
  - BDEC Branch if positive and decrement loop counter
  - BNOP Branch with NOP

- The C64x+ adds:
  - CALLP Protected Call

- These instructions provide code performance and code-size benefits

(Optional) C64x/C64x+ Branch – Additional Detail

BPOS vs. BDEC

- 0 (zero) is positive
- Reduces instructions needed for conditional loop
- Use any register for loop counter. (Notice A6 on the right)
- Allows two conditions, for example:
  Encode branch if \((A6 >= 0) \ AND \ (A2 != 0)\) by using
  \[[A2] \ BPOS \ loop, A6\]
- BDEC combines branch on positive (BPOS) with decrement
Using BPOS in dotp
- Must decrement loop counter by **one** since BPOS branches on 0 (zero)
- Allows code to use any register for subtract
- Guards against decrementing past 0 (zero)

<table>
<thead>
<tr>
<th>Loop:</th>
<th>MVK 256, A2</th>
<th>MVK 255, A8</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDH</td>
<td>*A5++, A0</td>
<td>LDH *A5++, A0</td>
</tr>
<tr>
<td>LDH</td>
<td>*A6++, A1</td>
<td>LDH *A6++, A1</td>
</tr>
<tr>
<td>NOP</td>
<td>4</td>
<td>NOP 4</td>
</tr>
<tr>
<td>MPY</td>
<td>A0, A1, A3</td>
<td>MPY A0, A1, A3</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td>ADD</td>
<td>A3, A4, A4</td>
<td>ADD A3, A4, A4</td>
</tr>
<tr>
<td>SUB</td>
<td>A2, 1, A2</td>
<td>SUB A8, 1, A8</td>
</tr>
<tr>
<td>B</td>
<td>loop</td>
<td>BPOS loop, A8</td>
</tr>
<tr>
<td>NOP</td>
<td>5</td>
<td>NOP 5</td>
</tr>
</tbody>
</table>

BPOS vs. BDEC in dotp
- BDEC branches and decrements at the same time
  - Must decrement loop counter by **two**
  - Performs subtract for you (frees up functional unit)

<table>
<thead>
<tr>
<th>Loop:</th>
<th>MVK 255, A8</th>
<th>MVK 254, A8</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDH</td>
<td>*A5++, A0</td>
<td>LDH *A5++, A0</td>
</tr>
<tr>
<td>LDH</td>
<td>*A6++, A1</td>
<td>LDH *A6++, A1</td>
</tr>
<tr>
<td>NOP</td>
<td>4</td>
<td>NOP 4</td>
</tr>
<tr>
<td>MPY</td>
<td>A0, A1, A3</td>
<td>MPY A0, A1, A3</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td>ADD</td>
<td>A3, A4, A4</td>
<td>ADD A3, A4, A4</td>
</tr>
<tr>
<td>SUB</td>
<td>A8, 1, A8</td>
<td>BPOS loop, A8</td>
</tr>
<tr>
<td>BPOS</td>
<td>loop, A8</td>
<td>BDEC loop, A8</td>
</tr>
<tr>
<td>NOP</td>
<td>5</td>
<td>NOP 5</td>
</tr>
</tbody>
</table>
BNOP

Reduces code size by saving an instruction.
(That is, if you cannot replace NOPs with “useful” stuff.)

And finally, the ADDKPC instruction is used by the compiler and assembly optimizer to perform a more efficient call. This instruction adds a constant offset to the Program Counter.

**ADDKPC**

```plaintext
0000  B  loop
0004  MVKL ret, B3
0008  MVKH ret, B3
000C  NOP 3
0010  ret MVK 4, A0
```

Fewer instructions needed for function call.

Why call it ADDKPC?

- **Compile Time:** assembler puts constant “offset” into instruction
- **Run Time:** offset is added to the current PC
  
  B3 = PC + offset  
  (8 = 4 + 4)  
  B3 = ret

<table>
<thead>
<tr>
<th>[cond]</th>
<th>B3</th>
<th>return addr – PC</th>
<th>4</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>return reg</td>
<td>offset (7 bits)</td>
<td>#nops</td>
<td>8 - 4</td>
<td>4</td>
</tr>
</tbody>
</table>
CALLP Instruction (C64x+ Only)

```
Sub1:  ADD .L1 A6, A3, A6
       MVKL .S2 Next, B3
       MVKH .S2 Next, B3
       B .S1 Sub2
       NOP 5
Next:  LDH .D1 *A7++, A6

Sub1:  ADD .L1 A6, A3, A6
       CALLP .S2 Sub2, B3
Next:   LDH .D1 *A7++, A6

Sub2:  BNOP .S2 B3, 4
       ADD .L1 A8, A3, A8
```

"CALL"

```
Next:  LDH .D1 *A7++, A6

Return using BNOP
```

CALLP (protected call)
- Similar to B, CALLP branches to Sub2 using 21-bit relative displacement
- Next EP Address is placed into B3 (if .S2) or A3 (if .S1)
- Implied "NOP 5" is inserted into pipeline after CALLP
- Used by compiler with –ms0 thru –ms3 (which are discussed in Chapter 10)
## Pointer Indexing/Offset Exercises

As the note states, each of these problems (1-5) are designed to be treated independently. In other words, they all start from the same initial register and memory values shown below. (This is in contrast to the second Exercise.)

### Addressing Exercise

```
| A0 | 8 |
| A3 | 4 |
```

(Note: Questions are independent, not sequential)

<table>
<thead>
<tr>
<th>Questions</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. LDH *A0--[A3],A5</td>
<td>A0= ___ A5= ___</td>
</tr>
<tr>
<td>2. LDH *++A3(3), A5</td>
<td>A3= ___ A5= ___</td>
</tr>
<tr>
<td>3. LDB *+A0[A0], A5</td>
<td>A0= ___ A5= ___</td>
</tr>
<tr>
<td>4. LDH *--A3[0], A5</td>
<td>A3= ___ A5= ___</td>
</tr>
<tr>
<td>5. LDB *-A0(3), A5</td>
<td>A0= ___ A5= ___</td>
</tr>
</tbody>
</table>

### Optional Exercise

Addressing (little endian)

<table>
<thead>
<tr>
<th>Code</th>
<th>Initial conditions</th>
<th>A0</th>
<th>A1</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldw.d1</td>
<td>*A0,A1</td>
<td>02001230</td>
<td>00011016</td>
<td>0200_1230</td>
<td>00011019</td>
</tr>
<tr>
<td>sub.d1</td>
<td>A1,3h,A1</td>
<td></td>
<td></td>
<td>34</td>
<td>11223344</td>
</tr>
<tr>
<td>stw.d1</td>
<td>A1,*+A0[4]</td>
<td></td>
<td></td>
<td>38</td>
<td>EEEEE7780</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3C</td>
<td>99AABBCC</td>
</tr>
<tr>
<td>ldh.d1</td>
<td>*A0++[4],A1</td>
<td></td>
<td></td>
<td>44</td>
<td>BADDF00D</td>
</tr>
<tr>
<td>ldb.d1</td>
<td>*--A0,A2</td>
<td></td>
<td></td>
<td>48</td>
<td>2598CAFE</td>
</tr>
<tr>
<td>sub.d1</td>
<td>A2,16,A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldh.d1</td>
<td>+++A0[A2],A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add.d1</td>
<td>A1,A2,A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lddh.d1</td>
<td>*++A0[3],A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldw.d1</td>
<td>*--A0[0],A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zero</td>
<td>A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mvlhl.s1</td>
<td>320ch,A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add.d1</td>
<td>A2,A3,A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lddh.d1</td>
<td>*++A0[6],A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub.d1</td>
<td>A3,A1,A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: NOP’s have been removed to allow code to fit on one page.
(Optional) Additional Logical / Bitfield Instructions

(Optional) Appendix

Logical and Bitfield Operations
- Left Most Bit Detect (LMBD)
- Normalization (NORM)
- Shifts (SHL, SHR)
- Saturate and Shift Left (SSHL)
- Bit-field Set/Clear (SET, CLR)

LMBD and NORM

<table>
<thead>
<tr>
<th>LMBD  .L src1, src2, dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1 (R_{LSB} or const): 0 or 1</td>
</tr>
<tr>
<td>src2: register to test</td>
</tr>
<tr>
<td>dst: #bits up to src1 (0 or 1)</td>
</tr>
</tbody>
</table>

Search: 1, dst = 2

<table>
<thead>
<tr>
<th>src2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

NORM .L src2, dst
Locates first non-redundant sign bit in src2 and places the count as an unsigned int in dst

<table>
<thead>
<tr>
<th>src2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

NORM returns 3
Shift Operations

- **SHL**
  - **SHL .S src2,src1,dst**
  - (src2 << src1)
  - src1: R or 5-bit constant

- **SHR**
  - **SHR .S src2,src1,dst**

- **SSHL**
  - saturate to 32 bits

**Example:**
- **SHL .S1 A5:A4,6,A7:A6**
  - Bits 64:40 are set equal to zero

- **Example:**
  - **SSHL saturates the result if the MSB changes state after the shift operation**

  - **SSHL is Saturation?**
  - **SSHL** saturates the result if the MSB changes state after the shift operation

**Example 1 - Shift source left 13 bits:**
- **source = 0x000FE000** (leading bit 0 ⇒ + number)
- **SHL = 0xFC000000** (leading bit 1 ⇒ - number)
- **SSHL = 0x7FFFFFFF** (largest positive number)

**Example 2 - Shift source left 24 bits:**
- **source = 0xFFFFF003** (leading bit 1 ⇒ - number)
- **SHL = 0x03000000** (leading bit 0 ⇒ + number)
- **SSHL = 0x80000000** (largest negative number)

- **We’ll discuss saturation in more detail during the Numerical Issues module**
Bit-Field SET/CLR

SET/CLR .S src2, csta, cstb, dst

cstb = MSB

csta = LSB

Bit-Field SET/CLR (Dynamic)

SET/CLR .S src2, src1, dst

cstb = MSB

csta = LSB
This page intentionally left blank.
Introduction

Now that you’ve written assembly code, it’s time to examine various techniques to optimize your code. You’ll have an opportunity to experiment with them during the upcoming lab exercise.

The next chapter introduces Software Pipelining, which provides another – more efficient – means of implementing these techniques.

Learning Objectives

<table>
<thead>
<tr>
<th>Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>✦ Describe four methods of optimization</td>
</tr>
<tr>
<td>✦ General C6000 Multiply Instructions</td>
</tr>
<tr>
<td>✦ Explore packed-data processing</td>
</tr>
<tr>
<td>✦ Describe the processor-specific instructions on the C64x/C64x+</td>
</tr>
<tr>
<td>✦ (Lab) Implement word-wide optimization</td>
</tr>
</tbody>
</table>
Chapter Topics

Optimization Methods .............................................................................................................................. 7-1

- Parallel Instructions .......................................................................................................................... 7-4
- Fill Delay Slots ............................................................................................................................... 7-6
- Unrolling Loops ............................................................................................................................... 7-8
- Word-Wide Optimizations (Using LDW) ......................................................................................... 7-12

C6000 Multiply Instructions ................................................................................................................ 7-14
- Classic Multiplies – All C6x devices (C62x, C67x) .......................................................................... 7-14
- Sidebar – Packed Data Processing ............................................................................................... 7-16
- C64x Multiplies and Packed Data Processing Summary .............................................................. 7-16
- C64x+ Multiply Instructions .......................................................................................................... 7-19
- C66x Multiplies .............................................................................................................................. 7-24
- Summary of MMACS by Generation ........................................................................................... 7-25

Specialized Instructions .................................................................................................................... 7-27
- Specialized C64x Instructions ..................................................................................................... 7-27
- Specialized C64x+ Instructions ................................................................................................. 7-28

Quick Intro to Video Compression (Trying out SUBABS4, DOTPU4) ............................................ 7-29
Optimization Methods

There are four basic optimizations you can implement.

<table>
<thead>
<tr>
<th>Four Basic Optimization Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>☑ Use instructions in <strong>parallel</strong></td>
</tr>
<tr>
<td>☑ <strong>Fill delay slots</strong> with useful instructions - replace NOPs</td>
</tr>
<tr>
<td>☑ <strong>Loop unrolling</strong></td>
</tr>
<tr>
<td>☑ <strong>Word-wide</strong> optimization</td>
</tr>
</tbody>
</table>

The first optimization gets multiple functional units operating at the same time. Most optimally, you’d get all eight simultaneously operating on eight individual instructions.

The second optimization eliminates NOP instructions previously used to pad instruction delay slots.

The third eliminates or reduces loop management overhead by “unrolling” the loop.

Finally, the fourth optimization comes from the fact that the ‘C62x has a 32-bit data bus while the most common data types used in DSP operations are 16-bits wide. If you're doing 32-bit floating-point operations, you can make use of the 64-bit data paths on the ‘C67x.
Parallel Instructions

‘C6x assembly code uses || bars to signify parallel instructions:

\[
\begin{align*}
\text{add} & .L1 & A1,A2,A2 \\
\text{sub} & .L2 & B1,B2,B1 \\
\end{align*}
\]

You can put up to eight instructions in parallel as long as each one of them uses a different functional unit. Earlier in the workshop we defined the term Execute Packet (EP). Execute packets are another name for a set of parallel instructions. In our example above, ADD and SUB together make up an execute packet. If you remember, all instructions are fetched in Fetch Packets (FP). Therefore, this (ADD/SUB) execute packet is fetched with six other instructions, but executed independently of the other instructions. To simplify code processing, EP’s are not permitted to ‘straddle’ FP boundaries.

Let’s look at our previous Dot-Product example, what instructions can we put in parallel?

Using Parallel Instructions

```
loop:
  ldh.d1 *A8++,A2
  ldh.d1 *A9++,A3
  nop 4
  mpy.m1 A2,A3,A4
  nop
  add.l1 A4,A6,A6
  sub.l2 B0,1,B0
  [b0] b.s1 loop
  nop 5
```

What can you put in parallel?
If you guess the loads, you’re right. Don't forget, though, that the .D1 unit cannot perform two loads at the same time.

### Parallel Instructions

#### What can you put in parallel?

<table>
<thead>
<tr>
<th>Loop</th>
<th>What can you put in parallel?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldh.d1 *A8++,A2</code></td>
<td>Load Instructions</td>
</tr>
<tr>
<td><code>ldh.d1 *A9++,A3</code></td>
<td></td>
</tr>
<tr>
<td><code>nop</code></td>
<td></td>
</tr>
<tr>
<td><code>mpy.m1 A2,A3,A4</code></td>
<td></td>
</tr>
<tr>
<td><code>jmp</code></td>
<td></td>
</tr>
<tr>
<td><code>add.l1 A4,A6,A6</code></td>
<td></td>
</tr>
<tr>
<td><code>sub.l2 B0,1,B0</code></td>
<td></td>
</tr>
<tr>
<td><code>[b0] b.s1</code></td>
<td></td>
</tr>
<tr>
<td><code>nop</code></td>
<td></td>
</tr>
</tbody>
</table>

#### Parallel Instruction Summary

When implementing parallel instructions:

1. Get your code functioning, **then**
2. Implement parallel instructions

Putting instructions in parallel can often introduce subtle errors. It’s much easier to debug your parallel code if you know the algorithm was functioning properly before you began putting instructions in parallel. This caution applies to any optimizations. This example shows a 40 cycle improvement. While more dramatic examples can be found for parallel instructions, this demonstrates how they work within our dot-product example.
Optimization Methods Summary

✓ No Optimization
16 cycles x 40 iterations = 640

✓ Parallel Optimization
15 cycles x 40 iterations = 600

◆ Filling Delay Slots
◆ Loop Unrolling
◆ Word Wide Optimization

Fill Delay Slots

You’ve probably already heard how we jokingly refer to NOP, “Not Optimized Properly”. Funny thing is, it’s a true statement. Most NOP instructions can be replaced or eliminated. It takes code reordering and a little ingenuity to accomplish, though. Again, the Software Pipelining procedure provides a simple means of implementing this on loop code.

Use the optimization mark-up tool to mark up the code.

What can you do to eliminate NOPs?

Try eliminating the NOPs from the preceding code. Go ahead and mark it up with your pencil.
Here’s the code we created:

```
loop:
|   | ldh.d1  *A8++,A2
|| ldh.d2  *B9++,B5
   | sub  B0,1,B0
[b0] | b  loop
   | nop  2
 nop
 mpy.ml x A2,B5,A4
 nop
 add.l1  A4,A6,A6
```

By moving the branch up into the delay slots of the load instructions, we’ve killed two birds with one stone:

- Load delay slots reduced from four to two
- Branch delay slots have disappeared

Looking at the optimization summary, the routine drops to half the original number of cycles.

```
<table>
<thead>
<tr>
<th>Optimization Methods Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ No Optimization</td>
</tr>
<tr>
<td>16 cycles x 40 iterations = 640</td>
</tr>
<tr>
<td>✓ Parallel Optimization</td>
</tr>
<tr>
<td>15 cycles x 40 iterations = 600</td>
</tr>
<tr>
<td>✓ Filling Delay Slots</td>
</tr>
<tr>
<td>8 cycles x 40 iterations = 320</td>
</tr>
<tr>
<td>◆ Loop Unrolling</td>
</tr>
<tr>
<td>◆ Word Wide Optimization</td>
</tr>
</tbody>
</table>
```

Once again, software pipelining will provide a more efficient means of eliminating NOPs along with implementing parallel instructions.
Optimization Methods

Unrolling Loops

Unrolling loops involves replacing iterations of the loop with additional copies of the loop itself. Often it leads to faster, larger code. Here’s a conceptual example (we’re ignoring instruction delay-slots):

Example Loop

```assembly
mvk 4,B0
loop:  ldh
   ||  ldh
   mpy
   add  ; adds 1, 2, 3, & 4
[B0]  sub B0,1,B0
[B0]  b  loop
```

By unrolling this loop once you have:

Example 1

```assembly
mvk 3,B0
loop:  ldh
   ||  ldh
   mpy
   add  ; adds 1, 2, & 3
[B0]  sub B0,1,B0
[B0]  b  loop
```

As you can see, this obviously takes additional code space, but what advantage does it provide? By eliminating the loop overhead for the fourth add it runs a little faster.

Example 2

```assembly
mvk 2,B0
loop:  ldh
   ||  ldh
   mpy
   add  ; adds 1 & 3
   ldh
   ||  ldh
   mpy
   add  ; adds 2 & 4
[B0]  sub B0,1,B0
[B0]  b  loop
```

Example two demonstrates another way to view loop unrolling. In this case the loop has been unrolled, but the additional copy of the loop code remains inside the loop. Of course, an even loop count needs to be divided by two in this case. For odd number loops you’ll most likely want to combine Examples 2 and 3.
When taken to its extreme this method produces what’s often called linear code – a.k.a. straight-line code.

**Example 3**

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ldh</td>
<td>ldh</td>
</tr>
<tr>
<td>mpy</td>
<td>mpy</td>
</tr>
<tr>
<td>add</td>
<td>add</td>
</tr>
</tbody>
</table>
|     | ; add 1
|     | ldh |
| mpy | mpy |
| add | add |
|     | ; add 2
|     | ldh |
| mpy | mpy |
| add | add |
|     | ; add 3
|     | ldh |
| mpy | mpy |
| add | add |
|     | ; add 4
```

In the third example all the loop overhead code has been eliminated. Here’s a summary of all three variations.

**Loop Unrolling Example Summary** *(sans NOPs)*

```
<table>
<thead>
<tr>
<th>Ex.</th>
<th>Loop Count</th>
<th>Total Adds</th>
<th>Cycles</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>4</td>
<td>4</td>
<td>21</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>19</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>17</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>4</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
```

- Loop count may not always equal the number of algorithm iterations.
- Unrolling loops usually increases speed but also code size.

The example shown here is conceptual in that it doesn’t account for branch, load, and multiply delay-slots. With the flexibility of the VelociTI architecture, though, the code-size vs. performance trade-off is minimized (esp. vs. std VLIW). Unrolling only speeds up code to a certain point, i.e. the law of diminishing returns prevails. In other words, you only want to unroll a loop to a certain point, beyond that you only increase code-size, not speed.

Parallel instructions and filling delay-slots are implemented by the software pipelining process. This process also implements loop unrolling, thus minimizing your need to do it manually. At times, though, you can unroll loops further to gain additional performance over what the software optimization tools create – as demonstrated in the optional multi-cycle example (Weighted Vector Sum) in the optional discussion at the end of this chapter.
When Can We Unroll a Loop?

Looking at the following two examples, Example 1 will work fine for any COUNT value greater than one. On the other hand, Example 3 requires the minimum value for COUNT to be two, otherwise the result will be incorrect.

For each example above, what constraints are there on COUNT?

In Example 3, is there a minimum loop count required? How about multiple of?

In fact, it’s not just the minimum value that can decide functionality of the examples, Example 3 also requires COUNT to be a multiple of 2. (Additional code could be added to handle the odd case(s), but this would increase code-size.)
Trip Count (Must Iterate pragma)

Trip count represents the initial count value for a loop (before loop unrolling); that is, the number of iterations required. As in our previous examples, you might “count” the number of adds to be performed and use this as the basis of your trip count. Knowing the value of the loop counter becomes important as we begin optimizing our system.

The more information you have about your problem, the better your code efficiency will be. In fact, one reason why assembly programming often creates code with higher efficiency is the ability to use knowledge such as trip counts.

In the past, most tools have not provided a means to pass along this additional information. The C6x code generation tools, on the other hand, provide a means of passing along this information. While the tools can often figure out the required information on their own, there are times when it is impossible for the tools to know beforehand. In these cases you may be able to improve your code by using either the: .trip directive or MUST_ITERATE pragma.

### Specifying the Trip Count

**Trip Count**: The number of times your algorithm iterates (before loop unrolling)

**Help the tools optimize code** size and performance by providing this information:
- Assembly Optimizer: .trip min, max, multiple of
- C Compiler: #pragma MUST_ITERATE (min, max, multiple of)

<table>
<thead>
<tr>
<th>mvk COUNT,B0</th>
<th>loop1: .trip 8,400,8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ldh</td>
</tr>
<tr>
<td></td>
<td>ldh</td>
</tr>
<tr>
<td></td>
<td>mpy</td>
</tr>
<tr>
<td></td>
<td>add</td>
</tr>
<tr>
<td>[B0]</td>
<td>sub B0,1,B0</td>
</tr>
<tr>
<td>[B0]</td>
<td>b loop1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mvk COUNT,B0</th>
<th>loop3: .trip 8,400,2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ldh</td>
</tr>
<tr>
<td></td>
<td>ldh</td>
</tr>
<tr>
<td></td>
<td>mpy</td>
</tr>
<tr>
<td></td>
<td>add</td>
</tr>
<tr>
<td>[B0]</td>
<td>sub B0,1,B0</td>
</tr>
<tr>
<td>[B0]</td>
<td>b loop3</td>
</tr>
</tbody>
</table>

- You are guaranteeing to the tools the range of your loop counter
- If you don’t know the range, don’t use these optimizations
- Examples of MUST_ITERATE pragma in later chapter

The MUST_ITERATE pragma – along with the need for the maximum count – will be discussed in a later chapter.
Word-Wide Optimizations (Using LDW)

Before looking at the word-wide technique, let’s quickly review the original 16-bit version of the Dot-Product routine used in our previous lab exercise:

![Dot Product Using LDH Diagram]

Using the word-wise optimization you can see that we’ll pull two 16-bit values from each array with each load:

![Optimizing - Using the Whole Bus Diagram]

We can then process each of these values. What’s the hook? Well, how do we multiply the values in the top part of each register? The MPY instruction only multiplies values from the lower 16-bits of the two register operands. As you can see below, the trick is to use two unique and different multiply instructions. MPY multiplies the lower 16-bits while MPYH (MPY-High) handles the upper 16 bits.
Optimization Methods

Optimizing - Using LDW/MPYH

\[
\begin{align*}
\text{LDW.D1} & \quad \text{A0} \\
\text{LDW.D1} & \quad \text{A1} \\
\text{MPY.M1} & \quad \text{A0, A1, A3} \\
\text{MPYH.M1} & \quad \text{A0, A1, A7} \\
\end{align*}
\]

\[
\begin{align*}
\text{LDW.D1} & \quad \text{A5++, A0} \\
\text{LDW.D1} & \quad \text{A6++, A1} \\
\text{MPY.M1} & \quad \text{A0, A1, A3} \\
\text{MPYH.M1} & \quad \text{A0, A1, A7} \\
\end{align*}
\]

\[
\begin{align*}
A7 & = A3 \\
\text{ADD.L1} & \quad \text{A3, A7, A4} \\
\end{align*}
\]

Use LDDW double-word (64-bit) loads rather than two LDW 32-bit for floating-point. Can we put MPY’s in parallel?

Word-wide optimization demonstrates the one use of loop unrolling. This case involves performing two dot-product calculations per loop (unrolling by one within the loop itself) and reducing the loop count by one-half.

Looking at the same example, albeit, a bit more optimized:

Optimize Loop - Keep Two Running Sums

\[
\begin{align*}
\text{LDW.D1} & \quad \text{A0} \\
\text{LDW.D1} & \quad \text{B0} \\
\text{MPY.M1x} & \quad \text{A0, B0, A5} \\
\text{MPYH.M2x} & \quad \text{A0, B0, B5} \\
\text{ADD.L1} & \quad \text{A5, A6, A6} \\
\text{ADD.L2} & \quad \text{B5, B6, B6} \\
\end{align*}
\]

We’ll see more word-wide optimizations in a few minutes…
C6000 Multiply Instructions

Classic Multiplies – All C6x devices (C62x, C67x)

There are four basic multiply instructions that allow you to perform 16-bit multiplies on any combination of 16-bit values contained in the two 32-bit source registers.

### C62x Multiply Instructions

- **Four base multiply instructions:**
  - MPYH
  - MPY
  - MPYLH
  - MPYHL

- **Each multiply has 4 signed/unsigned variations (SS, SU, US, UU)**

  For example:
  - MPY Normal 16-bit multiply with two signed operands
  - MPYU Two unsigned operands
  - MPYUS One unsigned, one signed operand
  - MPYSU One signed, one unsigned operand

Actually, there are four variations for each of these instructions. For example:

- MPY Normal 16-bit multiply with two signed operands
- MPYU Two unsigned operands
- MPYUS One unsigned, one signed operand
- MPYSU One signed, one unsigned operand

The advantages of these variations are direct support for mixed data types. In particular, this helps the compiler. Compilers often must add overhead to convert unsigned to signed types before completing multiplication. Of course, you’d have to do the same in assembly. Performance is maximized by direct hardware support of these operations.
The new multiply instructions added to the C64x are mentioned later in the chapter.
Sidebar – Packed Data Processing

Packed data processing is defined here as working on more than one data value per instruction. Sometimes this type of execution is called SIMD: single-instruction multiple-data. Due to the flexible nature of C6000’s support of packed data processing, we’ll use the more generic term.

ADD2/SUB2 Example

```
ADD / ADD2
All C6000 devices have:
• ADD2
• SUB2

ADD Example
LDH *A5,A0  
LDH *A6,A1  
ADD A0,A1,A2  

A0 0000 0001  
A1 0000 7001  
---------  
A2 0000 7002  

ADD2 Example
LDW *A5,A0  
LDW *A6,A1  
ADD2 A0,A1,A2  

A0 0002 0001  
A1 7002 7001  
---------  
A2 7004 7002  
```

C64x Multiplies and Packed Data Processing Summary

The C64x offers additional instructions that speed up computation. Here are some samples of C64x code and instructions.

```
MPY / MPYH Summary

```

---

**ADD Example**

```
LDH *A5,A0  
LDH *A6,A1  
ADD A0,A1,A2  

A0 0000 0001  
A1 0000 7001  
---------  
A2 0000 7002  
```

**ADD2 Example**

```
LDW *A5,A0  
LDW *A6,A1  
ADD2 A0,A1,A2  

A0 0002 0001  
A1 7002 7001  
---------  
A2 7004 7002  
```

---

**MPY / MPYH Summary**

```
MPY .M1 A0,A1,A5  
MPYH .M2 A0,A1,A5  
```
**MPY2**

\[
\begin{align*}
& a_1 \times x_1 = a_0 \times x_0 + a_1 \times x_1 + a_3 \times x_3 \\
& + a_0 \times x_0 + a_2 \times x_2 \\
& B_6 \times A_6 + \text{final sum} \\
& = A_3 \times A_2 \\
& \text{MPY2}\ A_0, B_0, A_3 : A_2 \\
& \text{MPYH.M2} \ A_0, B_0, B_5 \\
& \text{ADD.L1} \ A_2, A_6, A_6 \\
& \text{ADD.L2} \ A_3, B_6, B_6 \\
& \text{ADD.L1} \ A_2, A_6, A_6 \\
& \text{ADD.L1} \ A_3, B_6, B_6 \\
& \text{ADD.L1} \ A_3, A_4, A_4
\end{align*}
\]

**DOTP**

\[
\begin{align*}
& c_1 \times x_1 = c_0 \times x_0 + c_1 \times x_1 + c_0 \times x_0 \\
& + \text{running sum} \\
& = A_3 \\
& \text{DOTP2.M1x} \ A_0, B_0, A_3 \\
& \text{ADD.L1} \ A_3, A_4, A_4
\end{align*}
\]

Is this the best that the C64x can do?
DOTP2 with LDDW

\[
\begin{align*}
&\text{a3 a2} \quad \begin{array}{c}
\text{a1} \\
\text{x3} \quad \text{x2}
\end{array} \quad \text{A1:A0}

\text{LDDW.D1}\text{*A4++},\text{A1:A0} \\
\text{x1} \quad \text{x0} \quad \text{B1:B0}
\end{align*}
\]

\[
\begin{align*}
&\text{a3}*\text{x3} + \text{a2}*\text{x2} \\
&\text{a1}*\text{x1} + \text{a0}*\text{x0}
\end{align*}
\]

\[
\begin{align*}
&\text{B2} = \text{A2} \\
&\text{B3} = \text{A3}
\end{align*}
\]

\[
\begin{align*}
&\text{ADD A2, A3, A3} \\
&\text{ADD B2, B3, B3}
\end{align*}
\]

In Ch 8, we'll get all these instructions working in parallel

Summary of C64x Packed Data Instructions

'C64x Packed Data Processing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Quad 8-Bit</th>
<th>Dual 16-Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDx, SUBx</td>
<td>X</td>
<td>X</td>
<td>Adds/Subtracts</td>
</tr>
<tr>
<td>SADDx</td>
<td>X</td>
<td>X</td>
<td>Saturated Adds</td>
</tr>
<tr>
<td>MPYx</td>
<td>X</td>
<td>X</td>
<td>Multiplies</td>
</tr>
<tr>
<td>DOTPx</td>
<td>X</td>
<td>X</td>
<td>Dot Products (a<em>x1)+(b</em>x2)</td>
</tr>
<tr>
<td>DOTPxRx</td>
<td>X</td>
<td>X</td>
<td>Dot Products with Rounding</td>
</tr>
<tr>
<td>PACKx</td>
<td>X</td>
<td>X</td>
<td>Pack Operations</td>
</tr>
<tr>
<td>SPACKx</td>
<td>X</td>
<td>X</td>
<td>Saturated Pack Operations</td>
</tr>
<tr>
<td>UNPKx4</td>
<td>X</td>
<td>X</td>
<td>Unpack Operations</td>
</tr>
<tr>
<td>CMPx</td>
<td>X</td>
<td>X</td>
<td>Compares</td>
</tr>
<tr>
<td>MINx, MAXx</td>
<td>X</td>
<td>X</td>
<td>Min/Max Operations</td>
</tr>
<tr>
<td>SHRx2</td>
<td>X</td>
<td>X</td>
<td>Shifts</td>
</tr>
<tr>
<td>ABS2</td>
<td>X</td>
<td>X</td>
<td>Absolute Value</td>
</tr>
<tr>
<td>LDNx, STNx</td>
<td>X</td>
<td>X</td>
<td>Non-aligned Loads/Stores</td>
</tr>
</tbody>
</table>
C64x+ Multiply Instructions

**MYP32**

### MPY32 A, B, msb:lsb

![MPY32 Diagram]

- **MPY32** vs. 32 x 32 Multiply Function from Runtime Support Library
- 1 instruction vs. entire function

### DDOTP (Block FIR Example)

#### Block Real FIR

```c
for (i = 0; i < ndata; i++) {
    sum = 0;
    for (j = 0; j < ncoef; j++){
        sum = sum + (d[i+j] * c[j]);
    }
    y[i] = sum;
}
```

<table>
<thead>
<tr>
<th>loop iteration</th>
<th>[i, j]</th>
<th>[0, 0]</th>
<th>[0, 1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>d0c0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d1c1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d2c2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d3c3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- data
- coef
- + ↔ y
### Block Real FIR Example (DDTOPL2)

```c
for (i = 0; i < ndata; i++) {
    sum = 0;
    for (j = 0; j < ncoeff; j++) {
        sum = sum + (d[i+j] * c[j]);
    }
    y[i] = sum;
}
```

#### Parallel DDOTPL2’s

- **DDOTPL2.M1** `d3d2:d1d0, c1c0, sum1:sum0`
- **DDOTPL2.M2** `d7d6:d5d4, c3c2, sum3:sum2`

<table>
<thead>
<tr>
<th>Loop Iteration [i, j]</th>
<th>[0, 0]</th>
<th>[0, 1]</th>
<th>[0, 2]</th>
<th>[0, 3]</th>
<th>[0, 4]</th>
<th>[0, 5]</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d0c0</code> + <code>d1c1</code></td>
<td><code>d1c0</code> + <code>d2c1</code></td>
<td><code>d2c0</code></td>
<td><code>d3c1</code></td>
<td><code>d3c0</code></td>
<td><code>d4c0</code></td>
<td><code>d5c1</code></td>
</tr>
<tr>
<td><code>d2c2</code></td>
<td><code>d3c2</code></td>
<td><code>d4c3</code></td>
<td><code>d5c2</code></td>
<td><code>d5c1</code></td>
<td><code>d6c2</code></td>
<td><code>d6c1</code></td>
</tr>
<tr>
<td><code>d3c3</code></td>
<td><code>d4c4</code></td>
<td><code>d5c3</code></td>
<td><code>d6c4</code></td>
<td><code>d6c3</code></td>
<td><code>d7c3</code></td>
<td><code>d8c4</code></td>
</tr>
<tr>
<td><code>d4c5</code></td>
<td><code>d5c4</code></td>
<td><code>d6c5</code></td>
<td><code>d7c4</code></td>
<td><code>d8c5</code></td>
<td><code>d6c3</code></td>
<td></td>
</tr>
<tr>
<td><code>d5c6</code></td>
<td><code>d6c6</code></td>
<td><code>d7c5</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>d6c7</code></td>
<td><code>d7c6</code></td>
<td><code>d8c6</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>d7c7</code></td>
<td></td>
<td><code>d8c7</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Four 16x16 multiplies
- In each M unit every cycle adds up to 8 MACs/cycle, or 8000 MMACS
- Bottom Line: Two loop iterations for the price of one
C6000 Multiply Instructions

Real Block FIR

```
[A_j] SFLOOPD 4
|| MVC .S2 R_i0, ILC ;set ILC
|| ADD .S2 R_i0, 1, R_i0 ;+1/4
|| ADDAH .D2 B_DLYaddr, nCoefs-1+4, B_DLYOUTaddr
|| MVK .S1 nCoefs/4-3, A_T
|| ADDAB .D1 A_DLYaddr, 8, A_DLYaddr

*- stage A  -------------------------------------------------------------------*

LDDW .D1T2 *++A_DLYaddr, B_d7d6:B_d5d4 ;[ 1,1]
|| LDDW .D2T1 *++B_DLYaddr, A_d3d2:A_d1d0 ;[ 1,1]
|| LDDW .D2T2 *++B_COEFaddr, B_c3c2:B_c1c0 ;[ 2,1]
|| LDDW .D1T1 *++A_COEFaddr, A_c3c2:A_c1c0 ;[ 2,1]

SPMASK
|| LDDW .D1T1 *A_DLYaddr[1], A_dbda:A_d9d8 ;[ 3,1]

SPMASK
|| MVC .S2 R_i0, RILC ;set RILC

SPMASK
|| LDDW .D2T1 *B_INaddr++, A_TEMP1:A_TEMP0 ;ld 1st input
|| LDDW .D2T2 *B_INaddr++, B_TEMP1:B_TEMP0 ;ld 2nd input
|| ZERO .L1 A_st ;clear st flag
|| ADDAB .D1 DP, outputs+8, A_OUTaddr
|| MVK .S2 nCoefs/3-1, B_TC
|| MVK .S1 nCoefs/3-1, A_TC

*- stage B  -------------------------------------------------------------------*

SPMASK
|| SUB .L2X A_OUTaddr, 8, B_OUTaddr
NOP
|| DMV .S2X B_d5d4, A_d3d2, B_d5d4_:B_d3d2_ ;[ 7,1]
|| DDOTPL2 .M1 A_d3d2:d1d0, c1c0, s0:s0

SPMASK
|| DMV .S1X A_d6d8, B_d7d6, A_dbad:B_dbad_ ;[ 8,1]
|| DDOTPL2 .M2 B_d7d6:E_d0d4, B_c1c0, A_dbad:B_dbad_ ;[ 8,1]
|| DDOTPL2 .M1 B_d6d8:E_d0d4, B_c0c2, A_pla:p4 ;[ 8,1]
|| STHWD .DT2 B_TEMP1:TEMP0, B_DLYOUTaddr++ ;set 1st input

New double-throughput 16-bit mpy instructions
```

Block Real FIR

- **C64x Implementation**
  
  DOTP2 d1d0, c1c0, s0 :d[1]*c[1] + d[0]*c[0]
  
  DOTP2 d2d1, c1c0, s1 :d[2]*c[1] + d[1]*c[0]

- **C64x Plus Implementation**

  DDOTPL2 d3d2:d1d0, c1c0, s1:s0

- **Reduces .M requirement by half**

  C64x: 194 cycles, 624 bytes (N=40, T=16)
  
  C64x+: 126 cycles, 496 bytes (N=40, T=16)
**DDOTP4**

**DDOTP4**

```plaintext
DDOTP4 (.unit) src1, src2, dst_o:dst_e
```

---

**CMPY (Block Real FIR Example)**

**Complex Multiply (CMPY)**

```
<table>
<thead>
<tr>
<th>A0</th>
<th>r1</th>
<th>i1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A1</th>
<th>r2</th>
<th>i2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r1<em>r2 - i1</em>i2</td>
<td>i1<em>r2 + r1</em>i2</td>
</tr>
</tbody>
</table>
```

- Four 16x16 multiplies per .M unit
- Using two CMPYs, a total of eight 16x16 multiplies per cycle
- Floating-point version (CMPYSP) uses:
  - 64-bit inputs (register pair)
  - 128-bit packed products (register quad)
  - You then need to add/subtract the products to get the final result
Complex MPY

- **CMPY (.M) src1, src2, dst_o:dst_e**
  - Four 16-bit inputs (real0,imag0, real1,imag1)
  - Two 32-bit outputs (real, imag)

- **CMPYR (.M) src1, src2, dst**
  - Four 16-bit inputs (real0,imag0, real1,imag1)
  - One packed 32-bit output (16 bit real, 16 bit imag)
  - Rounded by adding $2^{15}$

- **CMPYR1 (.M) src1, src2, dst**
  - Four 16-bit inputs (real0,imag0, real1,imag1)
  - One packed 32-bit output (real, imag)
  - Rounded by adding $2^{14}$ plus left shift

Block Complex FIR

- **C64x Implementation**
  
  DOTP2  dre_dim, cim_cre, pim
  DOTPN2 dre_dim, cre_cim, pre

- **C64x Plus Implementation**
  
  CMPY  dre_dim, cre_cim, pre:pim

- **Reduces .M requirement by half**

  - **C64x:**  674 cycles, 572 bytes ($N=40, T=16$)
  - **C64x+**: 344 cycles, 460 bytes ($N=40, T=16$)

  $N =$ Length of example block, $T =$ data size
C66x Multiplies

QMPY32 and QMPYSP

### QMPY32 (fixed), QMPYSP (float)

- **A3:A2:A1:A0**
  - \(c3\) \(c2\) \(c1\) \(c0\)
  - \(x\) \(x\) \(x\) \(x\)

- **A7:A6:A5:A4**
  - \(x3\) \(x2\) \(x1\) \(x0\)

- **A11:A10:A9:A8**
  - \(c3\times x3\) \(c2\times x3\) \(c1\times x1\) \(c0\times x0\)

- Single .M unit

- **Four 32x32 multiplies per .M unit**
- **Total of eight 32x32 multiplies per cycle**
- **Fixed or floating-point versions**
- **Output is 128-bit packed result (register quad)**

### DCMPY

**Dual Complex Multiply (DCMPY)**

- **src1**
  - \(r1\) \(i1\) \(r2\) \(i2\)

- **src2**
  - \(ra\) \(ia\) \(rb\) \(ib\)

- **dest**
  - \(r1*ra-i1*ia\) \(i1*ra+r1*ia\) \(r2*rb-i2*ib\) \(i2*rb+r2*ib\)

- Single .M unit

- **Eight 16x16 multiplies (plus 3 accumulations) per .M unit**
- **Using two DCMPYs, a total of eight 16x16 multiplies per cycle**
- **Floating-point version (CMPYSP) uses:**
  - 64-bit inputs (register pair)
  - 128-bit packed products (register quad)
  - You then need to add/subtract the products to get the final result
**C6000 Multiply Instructions**

**CMAXMUL**

**Complex Matrix Multiply (CMAXMUL)**

\[
\begin{bmatrix}
M9 \\
M8
\end{bmatrix} = \begin{bmatrix}
M7 \\
M6
\end{bmatrix} \times \begin{bmatrix}
M3 \\
M2 \\
M1 \\
M0
\end{bmatrix}
\]

\[M9 = M7\times M3 + M6\times M1\]
\[M8 = M7\times M2 + M6\times M0\]

Where \(Mx\) represents a packed 16-bit complex number

- Single .M unit implements complex matrix multiply using 16 MACs (all in 1 cycle)
- Achieve 32 16x16 multiplies per cycle using both .M units

---

**Summary of MMACS by Generation**

**MMACS Summary by Generation**

How many 16-bit MMACs (millions of MACs) can the 'C6201 perform?

- **400 MMACs** (two .M units x 200 MHz)

How about latest CPU cores?

<table>
<thead>
<tr>
<th>CPU Core (example device)</th>
<th>C64x ('C6416)</th>
<th>C64x+ ('C6416)</th>
<th>C674x ('C6A8168)</th>
<th>C66x ('C6678)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMACS (16x16)</td>
<td>4,000</td>
<td>10,000</td>
<td>12,000</td>
<td>40,000</td>
</tr>
<tr>
<td># .M units</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td># MACS per .M per cycle</td>
<td>x 2</td>
<td>x 4</td>
<td>x 2</td>
<td>x 16</td>
</tr>
<tr>
<td>Speed</td>
<td>x 1 GHz</td>
<td>x 1.2 GHz</td>
<td>x 1.5 GHz</td>
<td>x 1.25 GHz</td>
</tr>
<tr>
<td>Total</td>
<td>= 4000</td>
<td>= 10000</td>
<td>= 12000</td>
<td>= 40000</td>
</tr>
</tbody>
</table>
This page is almost blank.
Specialized Instructions

Specialized C64x Instructions

### C64x Specialized Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITC4</td>
<td>Bit Count</td>
<td>Machine Vision</td>
</tr>
<tr>
<td>GMPY4</td>
<td>Galois Field MPY</td>
<td>Reed Solomon Support</td>
</tr>
<tr>
<td>SHFL</td>
<td>Bit Interleaving</td>
<td>Convolutional Encoder</td>
</tr>
<tr>
<td>DEAL</td>
<td>Bit De-interleaving</td>
<td>Cable Modem</td>
</tr>
<tr>
<td>SWAP4</td>
<td>Byte Swap</td>
<td>Endian Swap</td>
</tr>
<tr>
<td>XPNDx</td>
<td>Bit Expansion</td>
<td>Graphics</td>
</tr>
<tr>
<td>MPYHLx, MPYLIx</td>
<td>Extended Precision 16x32 MPYs</td>
<td>Audio</td>
</tr>
<tr>
<td>AVGx</td>
<td>Quad 8-bit, Paired 16-bit Average</td>
<td>Motion Compensation</td>
</tr>
<tr>
<td>SUBABS4</td>
<td>Quad 8-bit Absolute of Differences</td>
<td>Motion Compensation</td>
</tr>
<tr>
<td>SSHVL, SSHVR</td>
<td>Signed Variable Shift</td>
<td>GSM</td>
</tr>
</tbody>
</table>

Ten new instructions increase performance in specific applications
- SHFL, DEAL, and GMPY4 instructions are handy for error correcting algo’s
- DEAL improves 64QAM byte to symbol conversion by 15x
- SUBABS4 bolsters motion estimation by a factor of 7 ½

---

### SUBABS4

- Calculates the absolute value of difference between the packed 8-bit datum in sources

\[
\begin{align*}
A1 & : a3 & a2 & a1 & a0 \\
A3 & : b3 & b2 & b1 & b0 \\
|a3-b3| & |a2-b2| & |a1-b1| & |a0-b0|
\end{align*}
\]

- Aids in motion-estimation (and other) algorithms that compute the “best match” between two sets of 8-bit values

LDW *A0, A1
LDW *A2, A3
SUBABS4 A1, A3, A5

---

Skip to Lab 7
Specialized Instructions

Specialized C64x+ Instructions

New C64x+ Specialized Instructions

◆ More Multiply Bandwidth
  • 32-bit Multiplication
    • Signed/unsigned
    • 32 or 64-bit result
    • Saturation
  • Eight 16x16 MAC Operations/cycle
  • Complex Multiplication

◆ FFT, DCT Enhancements
  • A+B || A-B operations

◆ Packing Operations
  • DPACK2.L, DPACKX2.L, RPACK2.S, DMV

◆ Instructions for convolutional codecs (not discussed)
  • GMPY: 32-bit Galios-field multiply
  • XORMPY: GMPY with a zero polynomial
  • SHFL3: 3-way bit interleave

MAX2, MIN2

Vector Max/Min

◆ MAX2/MIN2
  • C64x: .L unit instructions
  • C64x+: .L or .S units

◆ 40 Sample Vector Max
  • C64x: 38 cycles, 288 bytes
  • C64x+: 31 cycles, 164 bytes
Quick Intro to Video Compression (Trying out SUBABS4, DOTPU4)

This brief introduction to video compression won’t make anyone a video expert, rather, it is to show the fundamentals of video motion estimation/compensation. With a minimal understanding of this algorithm, we can see how some of the C64x instructions can be applied to this problem.

Quick Intro to Video Compression

M-JPEG or Motion JPEG
- Each video frame compressed as a JPEG image

MPEG
- Uses Motion Compensation to exploit temporal redundancy
- That is, why keep a copy of the same thing over and over?
- Frame Types
  - "I" Frame
    - Intra-encoded frame
    - Compressed similar to JPEG
  - "P" Frame
    - Predicted Inter-frame
    - Re-uses redundant information between frames

In other words, rather than keeping all the pixels for every block on every frame, why not just tell the video system to redisplay a block of the image over and over again.
Then again, as shown next, what if the block moves from frame to frame?

![Diagram: What if it moves?]

- Objects in a frame will probably not stay in the same place
- Even if the objects don’t move, the camera may
- Let’s look at this a different way…

To figure out if (or where) a part of the image is repeated from frame to frame, a video frame is broken down into macroblocks. A common size for these is 16x16 pixels, though different video encoding standards use different sizes.

![Diagram: Divide the Picture into Macroblocks]

- Object has moved from frame to frame
For each block in a frame (say Frame 2 from below), the previous frame can be searched to find the best match for that block. From a quality perspective, it might be best to search the whole frame, but that’s too time consuming. Usually encoders define a search window and search through that window on a byte by byte basis. That is, they compare the block from Frame 2 against every block in the search window. Different methods of comparison can be used, but the Sum of Absolute differences is a common choice.

**Divide the Picture into Macroblocks**

Frame 1

Frame 2

On a byte-by-byte basis, compare the block of interest to the search area in the previous frame to find the best fit

- Very CPU intensive
- Common Methods:
  - Sum of Absolute Differences (SAD) – most common
  - Sum of Squared Error (SSE)
The SUBABS4 works quite well for implementing the Sum of Absolute Differences.

On a byte-by-byte basis, compare the block of interest to the search area in the previous frame to find the best fit:
- **Very CPU intensive**
- **Common Methods:**
  - Sum of Absolute Differences (SAD) – most common
  - Sum of Squared Error (SSE)
  - Motion vector is encoded rather than JPEG compressed version of block
8x8 MAD Algorithm

The Image Library (IMGLIB) provides the Sum of Absolute Differences in the form of a Minimum Absolute Difference (MAD) algorithm. This can be used to create the necessary routine for video motion estimation.

Here is a rough demonstration of the specialized C64x instructions used in the 8x8 MAD.

Both the LDNDW and SUBABS4 are excellent choices to solve this algorithm’s requirements. Also, this example happens to use the DOTPU4 instruction in a novel way. The goal isn’t to perform four-unsigned dot products, rather, it’s used to add four bytes all combined in a single register.
To view all the details about the minimum absolute difference algorithm implementation, please refer to the IMGLIB documentation (SPRU023.pdf).

```
void mad8x8_c(unsigned char *refImg, unsigned char *srcImg, unsigned int pitch, unsigned int *motvec)
{
    ... 
    for()
    {
        ... 
        ref_ptr = ref_ptr++;
        ref = _mem8(ref_ptr);
        ref3210 = _loll(ref);
        ref7654 = _hill(ref);
        err7654 = _subabs4(src7654, ref7654);
        err3210 = _subabs4(src3210, ref3210);
        curr_mad0 = _dotpu4(err7654, const_1111h);
        curr_mad1 = _dotpu4(err3210, const_1111h);
    }
    curr_mad = curr_mad0 + curr_mad1;
}
```

We’ve shown this example diagram and code because it demonstrated three specialized instructions: LDNDW, SUBABS4, and DOTPU4.
Software Pipelining

Introduction

Having examined various techniques for optimizing code, this chapter examines a procedure – Software Pipelining – that simplifies the application of these techniques.

Learning Objectives

<table>
<thead>
<tr>
<th>Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>✷ Understand concepts of software pipelining</td>
</tr>
<tr>
<td>✷ Use software pipelining procedure</td>
</tr>
<tr>
<td>✷ Code the word-wide software pipelined dot-product routine</td>
</tr>
<tr>
<td>✷ Determine if your pipelined code is more efficient with or without prolog/epilog</td>
</tr>
</tbody>
</table>
Chapter 8 Topics

Software Pipelining .......................................................................................................................... 8-1

What is Software Pipelining? ............................................................................................................. 8-3
Why Learn Software Pipelining? ........................................................................................................ 8-3
Introduction to Software Pipelining .................................................................................................. 8-4

Software Pipelining Terms .................................................................................................................. 8-6

Software Pipelining Procedure ......................................................................................................... 8-8
Step 1 - Write Algorithm in C and Verify .......................................................................................... 8-9
Step 2 - Write ‘C6x Linear Assembly Code ....................................................................................... 8-11
Step 3 - Create Data-Flow Dependency Graph .................................................................................. 8-12
Step 4 - Allocate Resources ............................................................................................................. 8-20
Step 5 - Create Scheduling Table ..................................................................................................... 8-21
Step 6 - Writing ‘C6000 Code ............................................................................................................ 8-25

Optional Topics .................................................................................................................................... 8-26
Conditional Subtract? ....................................................................................................................... 8-26
Epilog – How and If you should use it ............................................................................................... 8-27
SPLOOP Buffer (C64x+) .................................................................................................................... 8-32
Lab 8 Solutions ............................................................................................................................... 8-33
Software Pipelining Multi-Cycle Loops .......................................................................................... 8-36
Software Pipelining

Software pipelining is how we describe the set of heuristics used to create highly efficient loop code. We’ll break this down into a few procedural steps and implement them by hand. This chapter uses the dot-product routine you’ve used for the past couple of labs.

While the dot-product routine can be pipelined into a single-cycle loop, Chapter 9 explores pipelining loops that require multiple cycles. In Chapter 5 we used the assembly optimizer tool to perform software pipelining automatically.

What is Software Pipelining?

Software Pipelining

- Creates highly optimized **loop-code**
  - Implements parallel instructions
  - Fills delay slots
  - Maximizes functional units

- **Compiler or Assembly-Optimizer**
  - Implemented using option `–o2` or `–o3`

Why Learn Software Pipelining?

We could tell you we teach software pipelining so that you’ll “appreciate the tools”, but this would be a bit misleading. Sure, you will appreciate the automation these new tools bring to optimization. The real benefits are delineated below.

Why Learn Software Pipelining?

1. **Understand how tools create optimal code**, so you can ...
   - Read the tool’s output code
   - Check tools efficiency
2. **To write hand-optimized assembly**
3. **Most engineers just want to know how it works!**
**Introduction to Software Pipelining**

To introduce the concepts behind software pipelining, let’s use the following example:

```
Software Pipeline Example

<table>
<thead>
<tr>
<th>LDH</th>
<th>LDH</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPY</td>
<td>ADD</td>
</tr>
</tbody>
</table>
```

How many cycles would it take to perform this loop 5 times? (Disregard delay-slots).

______________ cycles

Implementing this code without software pipelining would take three cycles per iteration of the loop as shown in the following figure. Performing the loop five times would net a total of fifteen (15) cycles. Since we’re looking at this conceptually, we have ignored the necessary delay slots.

---

**Note:** Later, we explore a real example which includes handling instruction latencies.
Looking at this with pipelining shows a very different situation overall.

- In this case cycle one (1) remains the same.
- The first MPY occurs during cycle two. But, since the .D units were unused during cycle two, why not begin two new loads?
- Similarly, when the first add begins, can’t the second MPY and the third loads begin?
- This pattern continues until the sixth cycle. Why didn’t we perform two loads in cycle six? Oh yeah, since only five (5) loop iterations were planned, it makes sense to only perform five loads.
- Completion occurs in cycle seven (7), with the fifth and final ADD.

Upon examination, the pipelined example completes in less than half the cycles required for the non-pipelined version.
Software Pipelining Terms

The key to software pipelining is the Loop. Loop doesn’t mean anything different than in normal coding practice. It contains all the instructions necessary to implement the loops function. What is different, though, is that after software pipelining, when the “loop” is reached, each iteration of the loop performs all of the loop instructions, but these instructions may be operating on “different iterations” of the original loop code.

Let’s look at the example below to help clarify what we mean. Since our code only had four instructions, we know that these can easily be performed in a single-cycle. If our loop had more than eight instructions, we’d know for sure that it would take two or more cycles. Our single-cycle loop is performed three times. The first of these performs all the loop instructions, but it’s doing the first ADD, the second MPY, and the third LD’s.
If we’d been forced to implement a two-cycle loop it would have taken six cycles to perform these three loop iterations. (A two-cycle example is discussed in Chapter 9).

The **Prolog** is the “build-up” to the loop. Running the prolog code is often called “priming” the loop. The length of the prolog depends upon the latency between the start and end of the loop code; i.e. the number of instructions and their latency.

The final term, **Epilog**, refers to the final instructions, which must be completed after the loop iterations. You might think of the epilog as the complement to the prolog. To complete this discussion here’s the code from our conceptual example.

**Pipelined Code**

<table>
<thead>
<tr>
<th>prolog: LDH ; load 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loop: ADD ; add 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

| || ADD ; add 2 |
| || MPY ; mpy 3 |
| || LDH ; load 4 |
| || LDH |

The epilog is optional; it can be “rolled” into the loop. The advantages and disadvantages to using an epilog are in the optional discussion at the end of this chapter. We also discuss how the epilog can be created since the procedure we are about to show you doesn’t create epilog.
Software Pipelining Procedure

Six steps are defined to implement software pipelining. The first two are highlighted, as these are the only steps necessary when using the Compiler or Assembly Optimizer tools. This chapter discusses and provides an example for the entire process. In fact you’ll get to try this yourself in the lab exercises. When we used the Compiler and Assembly Optimizer tools earlier in the workshop, they software pipelined the code for us when we used full optimization.

<table>
<thead>
<tr>
<th>S/W Pipelining Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Write algorithm in C &amp; verify</td>
</tr>
<tr>
<td>2. Write ‘C6x Linear Assembly Code</td>
</tr>
<tr>
<td>3. Create dependency graph</td>
</tr>
<tr>
<td>4. Allocate registers</td>
</tr>
<tr>
<td>5. Create scheduling table</td>
</tr>
<tr>
<td>6. Translate scheduling table to ‘C6x code</td>
</tr>
</tbody>
</table>

We’ll examine each of these steps using our previous dot-product (half word version) example.
Step 1 - Write Algorithm in C and Verify

While this step isn’t an obvious part of assembly optimization it’s included here to emphasize its importance ‘C6x code writing. The efficiency produced by the ‘C6x Compiler combined with popularity and ease of C coding demands that this step be included. In fact, many users will find these results satisfactorily meet their performance needs. If so, the job is done most quickly and easily.

Under unique coding situations or if additional performance is required this step provides a means to verify the algorithm. It never pays to optimize an incorrect algorithm implementation.

S/W Pipelining Example (Step 1)

```c
short dotp(short *m, short *n, short count)
{
    int i;
    short product;
    short sum = 0;

    for (i=0; i < count; i++)
    {
        product = m[i] * n[i];
        sum += product;
    }
    return(sum);
}
```
Quick Evaluation

Before moving beyond C code, you should do an evaluation on where it’s worth optimizing this loop further. The idea of “knowing” your optimization target is discussed further in Chapter 10.

For now, let’s do a quick evaluation to see if this routine can possibly run at the rate of one-cycle per loop iteration. Loads must be performed on the .D unit, multiplies on .M, and branches on .S. Counting up the functional units we find:

<table>
<thead>
<tr>
<th>Instruction(s)</th>
<th>Unit</th>
<th>Number Available (per cycle)</th>
<th>Number Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDH, LDH</td>
<td>.D</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>MPY</td>
<td>.M</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>.S</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ADD, SUB</td>
<td>.L (/D/.S)</td>
<td>2 (2-6)</td>
<td>2</td>
</tr>
</tbody>
</table>

From this summary it looks like we have enough functional units to perform a loop iteration within one cycle. This is not a final indication since there are other constraints that can prevent single-cycle executions, e.g. data cross-paths. The formal process tests for all constraints and outputs the final ‘C6x assembly code.
Step 2 - Write ‘C6x Linear Assembly Code

Linear Assembly is a modified version of the assembly coding examined throughout the workshop thus far. Actually, it’s simpler to write and understand since you don’t need to specify: registers, functional units, or take into account NOPs.

Linear Assembly may use symbolic variables, like C, rather than register specifications. Of course you can still use registers, it’s just that you’re not required to do so. This same situation applies to functional units, you are not required to specify them.

The Assembly Optimizer tool inherently understands the delay slots required for each instruction. Unlike registers and functional units, you must not include instructions to account for the delay slots, rather write your assembly instructions in the order you’d like them to be executed.

Step 2 - Write ‘C6x Linear Assembly

; for (i=0; i < count; i++)
; prod = m[i] * n[i];
; sum += prod;

loop:   ldh     *p_m++, m
        ldh     *p_n++, n
        mpy     m, n, prod
        add     prod, sum, sum

[count]  sub     count, 1, count
[count]  b       loop

Review:  1. No NOPs
         2. No parallel instructions
         3. You don’t have to specify:
            • Functional units
            • Registers
Step 3 - Create Data-Flow Dependency Graph

Data dependencies become an obstacle to efficient loop programming. This is especially evident when various instructions require differing amounts of time to complete, i.e. delay slots.

Dependency graphs provide a visual way to describe your algorithm code. They graphically demonstrate data dependencies and resource limitations (data cross-paths, functional units) between instructions within a loop of code, thus making it easy to schedule proper (and highly efficient) execution of loop-code instructions.

Basic Terminology

We’ve chosen to implement standard data flow graphing techniques, modified slightly to meet our needs. Before graphing the Dot-Product example, let’s discuss some basic terminology.

Step 3 - Dependency Graph Terminology

- **Dependency Graph** – Diagram used to describe the flow of data in an algorithm.
- **Node** – Point on a dependency graph with one or more data paths flowing in and out. It is annotated with the result inside the node; process (instruction) atop; and functional unit along side.
- **Parent Node** – An instruction that writes to a variable is referred to as a parent instruction and defines a parent node.
- **Child Node** – An instruction that receives a byproduct of a parent node is known as it’s child and is called a child node.
- **Path** – Path shows the flow of data between two nodes. It indicates data dependencies in graphic terms. It’s annotated with the number of cycles required to complete the parent instruction.
- **Conditional Path** – Specifies a conditional dependency as opposed to a data dependency. Use when a child node uses a result as a conditional register. Represent these dependencies with a dashed line, as opposed to solid line, since cross-path restrictions differ for data and conditional dependencies.
**Steps to Create Dependency Graphs**

1. **Draw in nodes**
   - Start at the top
   - As you draw each node, include: instruction, result, paths

2. **Annotate paths with parent’s latency**

3. **Assign functional units**
   - Assign required functional units
     Only assign units to instructions which require specific units, i.e. \( D \) for loads
   - Partition nodes to A and B sides
     Here are some guidelines:
     - **Split \( D \)'s, \( S \)'s, \( M \)'s**
       When two of the same units are required in a loop, try to allocate one to each side of the processor
     - **Minimize data cross-paths**
       Only two data cross-paths are allowed per EP, one in each direction
     - **Balance units**
       Obviously, if there are eight instructions it wouldn’t make sense to put six instructions on the B side and only two on the A side
     - **Arbitrary**
       Even after the preceding rules, often there are still many possible solutions. Go ahead, pick one!
     - Assign specific functional units to all nodes
**Dependency Graph - Dot-Product Example**

**Dependency Graph**

**Step 1: Draw in the nodes**

Obviously, start at the beginning. Notice the load instructions are only parent nodes since they don’t rely on any previous instructions in the loop. (Note the initial addresses for m and n are normally loaded outside the loop.) The multiply, on the other hand, requires the results of both loads.

![Dot Product Example](image)

Next include the ADD instruction. Notice the output of the ADD instruction feeds back to become an input of itself on the next iteration. When a result is carried into the next iteration of the loop it’s called a *loop carry path*. (Discussed further in Chapter 9).

![Dot Product Example](image)
The dot-product’s dependency graph contains two separate data flows since the decrement of the loop counter and the branch do not read or write the variables from the other graph. In other words, there are no data dependencies between the two graphs.

Both graphs are included, though, since they are both part of the algorithm’s implementation.

Once again, a loop carry path is created by the SUB instruction. In the workshop examples, loop carry-paths are not an issue. In some cases, though, they become an issue. One case where this might happen is an algorithm that includes an instruction write to memory (ST) where the next iteration of the loop must read the updated location (LD). To read more about resolving this during hand optimization please refer to the *TMS320C6x Programmers Guide*. Fortunately, the tools easily handle this issue.
Dependency Graph
Step 2: Annotate Paths With Parent’s Latency

This is the easy step, just record the number of cycle to complete the parent nodes operation. Since all instructions require at least one cycle to execute, this will be the minimum number. Instructions that require delay-slots must have these latencies included.

Dot Product Example
Dependency Graph
Step 3: Assign functional units

Review the steps for assigning functional units located on page 8-13. Let’s take the suggestions one-by-one. First, label nodes that require specific functional units. LDH, MPY, and B are assigned below.

Don’t assign sides to these instructions, yet. For maximum flexibility, wait until you’ve partitioned the entire algorithm first. This suggestion helps with partitioning since it’s now fairly evident that the two side should be partitioned between the loads, i.e. one LDH will go on the A side and be implemented with \( .D_1 \) and the other will be implemented with \( .D_2 \) on the B side.

Where should you draw the line between the A and B sides? Let’s look at the suggestions again:

- **Split D’s, S’s, M’s**
  Well, we just discussed this one. Start a line between the two LDHs.

- **Minimize data cross-paths**
  In this example at least one cross-path exists. This can’t be avoided since both loads feed the multiply and we just decided to split the loads. By splitting the MPY and ADD another cross-path could be created. This is O.K. since it creates a cross-path in the opposite direction.

- **Balance units**
  We could chose to put three to a side, although it’s not required.

- **Arbitrary**
  It comes down to this. Various choices exist, turn the page to see our choice.
We’ve chosen to put three units to a side, splitting the LDHs one to a side, and only creating one cross-path.

After completing the final Dependency Graph step, assigning all functional units, our graph now looks like this. Note the \( .M1x \) where the MPY receives a source from the opposite side.
Next, Step 4

<table>
<thead>
<tr>
<th>S/W Pipelining Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Write algorithm in C &amp; verify</td>
</tr>
<tr>
<td>✓ Write ‘C6x Linear Assembly Code</td>
</tr>
<tr>
<td>✓ Create dependency graph</td>
</tr>
<tr>
<td>4. Allocate registers</td>
</tr>
<tr>
<td>5. Create scheduling table</td>
</tr>
<tr>
<td>6. Translate scheduling table to ‘C6x code</td>
</tr>
</tbody>
</table>
Step 4 - Allocate Resources

Three types of resources must be allocated: functional units, cross-paths, and registers.

Actually, our dependency graph shows the functional unit allocation. Looking at our summarization below, do we have enough functional units and cross-paths to complete a loop iteration in one cycle?

### Step 4 - Allocate Functional Units

<table>
<thead>
<tr>
<th></th>
<th>sum</th>
<th>prod</th>
<th>m</th>
<th>M1x</th>
<th>count</th>
<th>Do we have enough functional units to code this algorithm in a single-cycle loop?</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It appears the answer is … Yes. A single cycle loop is possible from a functional unit and cross-path standpoint.

You can allocate register resources any time from this point until translating the Scheduling Table to ‘C6x code. It’s convenient for us to do it now while we are allocating resources.

### Step 4 - Allocate Registers

<table>
<thead>
<tr>
<th>Register File A</th>
<th>#</th>
<th>#</th>
<th>Register File B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A0</td>
<td>B0</td>
<td>Count</td>
</tr>
<tr>
<td>m</td>
<td>A1</td>
<td>B1</td>
<td>n</td>
</tr>
<tr>
<td>prod</td>
<td>A2</td>
<td>B2</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td>A3</td>
<td>B3</td>
<td></td>
</tr>
<tr>
<td>&amp;m</td>
<td>A4</td>
<td>B4</td>
<td>&amp;n</td>
</tr>
<tr>
<td></td>
<td>A5</td>
<td>B5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A15/31</td>
<td>B15/31</td>
<td></td>
</tr>
</tbody>
</table>
Step 5 - Create Scheduling Table

The fifth step involves scheduling the instructions.

Scheduling Table

This is accomplished by working from the dependency graph and using a scheduling table. The scheduling table helps to keep track of code execution on a cycle-by-cycle basis through the prolog and first iteration of the loop.

<table>
<thead>
<tr>
<th>PROLOG</th>
<th>LOOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6</td>
<td>7</td>
</tr>
<tr>
<td>.L1</td>
<td></td>
</tr>
<tr>
<td>.L2</td>
<td></td>
</tr>
<tr>
<td>.S1</td>
<td></td>
</tr>
<tr>
<td>.S2</td>
<td></td>
</tr>
<tr>
<td>.M1</td>
<td></td>
</tr>
<tr>
<td>.M2</td>
<td></td>
</tr>
<tr>
<td>.D1</td>
<td></td>
</tr>
<tr>
<td>.D2</td>
<td></td>
</tr>
</tbody>
</table>

The table must be large enough to accommodate all these cycles. How large does it need to be to encompass both the Prolog and initial iteration of the loop?

In the case of the dot-product we need a table 8 cycles long. How did we come up with this number?
The length of the prolog is determined by examining the length of the longest data dependency path as demonstrated below.

**Length of Prolog**

- Count up the length of longest path(s)
- \(5 + 2 + 1 = 8\) cycles
- 8 cycles - 1 cycle loop
  \(\Rightarrow\) 7 cycle prolog
- Schedule table runs from cycle 0 to cycle 7
  - Prolog: cycles 0 - 6
  - Loop: cycle 7

In the preceding example we chose to start with cycle 0. You could start with 1, it’s arbitrary, but the length is still consistent.
**Scheduling Suggestions**

- Start with longest path
- Start scheduling as early as possible (cycle 0)
- Once schedule successive iterations continue to occur
- Reverse schedule branch and loop counter

Help your instructor fill out this chart …

### Step 5 - Create Scheduling Table

<table>
<thead>
<tr>
<th></th>
<th>PROLOG</th>
<th>LOOP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>.L1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.L2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Using these suggestions, begin by scheduling the LDH instructions. These are followed by the MPY and ADD instructions. After the addition, we begin reverse scheduling the branch. Where should the branch occur? It occurs at the end of cycle seven. Backing up, this means the branch needs to begin in cycle two. Therefore, the SUB needs to start one cycle earlier, in cycle one.

For added convenience subscripts have been added to the scheduled instructions. These represent the iteration number of each specific instruction. Notice that by the time we arrive at the first “loop” invocation we’ve only just started ADDing, but we’ve completed three multiplies and eight loads.

From this point, the loop continues to run until forty ADDs are completed.
**Step 6 - Writing ‘C6000 Code**

The final step of the procedure transfers the scheduling table to ‘C6x assembly code using the register and functional unit allocations described in step four.

### Step 6 - ‘C6x Code

<table>
<thead>
<tr>
<th>c0:</th>
<th>ldh .D1 *A1++,A2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ldh .D2 *B1++,B2</td>
</tr>
<tr>
<td>c1:</td>
<td>ldh .D1 *A1++,A2</td>
</tr>
<tr>
<td></td>
<td>ldh .D2 *B1++,B2</td>
</tr>
<tr>
<td></td>
<td>[B0] sub .L2 B0,1,B0</td>
</tr>
<tr>
<td>c2_3_4:</td>
<td>ldh .D1 *A1++,A2</td>
</tr>
<tr>
<td></td>
<td>ldh .D2 *B1++,B2</td>
</tr>
<tr>
<td></td>
<td>[B0] sub .L2 B0,1,B0</td>
</tr>
<tr>
<td></td>
<td>[B0] B .S2 loop</td>
</tr>
<tr>
<td></td>
<td>mpy .M1x A2,B2,A3</td>
</tr>
<tr>
<td>c5_6:</td>
<td>ldh .D1 *A1++,A2</td>
</tr>
<tr>
<td></td>
<td>ldh .D2 *B1++,B2</td>
</tr>
<tr>
<td></td>
<td>[B0] sub .L2 B0,1,B0</td>
</tr>
<tr>
<td></td>
<td>[B0] B .S2 loop</td>
</tr>
<tr>
<td></td>
<td>mpy .M1x A2,B2,A3</td>
</tr>
<tr>
<td></td>
<td>add .L1 A4,A3,A4</td>
</tr>
</tbody>
</table>

*** Single-Cycle Loop

```plaintext
loop: ldh .D1 *A1++,A2
      ldh .D2 *B1++,B2
      [B0] sub .L2 B0,1,B0
      [B0] B .S2 loop
      mpy .M1x A2,B2,A3
      add .L1 A4,A3,A4
```

To fit the code onto one foil we’ve abbreviated it a bit. First of all, “c0” means cycle zero and “c2_3_4” implies that the same parallel instruction is repeated for cycles two through four.

### Summary of Assembly Optimizations

<table>
<thead>
<tr>
<th>Optimization Methods Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ No Optimization</td>
</tr>
<tr>
<td>16 cycles x 40 iterations = 640</td>
</tr>
<tr>
<td>✓ Parallel Optimization</td>
</tr>
<tr>
<td>15 cycles x 40 iterations = 600</td>
</tr>
<tr>
<td>✓ Filling Delay Slots</td>
</tr>
<tr>
<td>8 cycles x 40 iterations = 320</td>
</tr>
<tr>
<td>✓ Word Wide Optimizations</td>
</tr>
<tr>
<td>8 cycles x 20 iterations = 160</td>
</tr>
<tr>
<td>✓ Software Pipelined - LDH</td>
</tr>
<tr>
<td>1 cycle x 40 loops + 7 prolog = 47</td>
</tr>
<tr>
<td>✓ Software Pipelined - LDW</td>
</tr>
<tr>
<td>1 cycle x 20 loops + 7p + final sum = 28</td>
</tr>
</tbody>
</table>
Optional Topics

Conditional Subtract?

Why did we make the subtract conditional in our code?

<table>
<thead>
<tr>
<th>Why Conditional Subtract?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>loop:</strong> ldh *p_m++, m</td>
</tr>
<tr>
<td>ldh *p_n++, n</td>
</tr>
<tr>
<td>mpy m, n, prod</td>
</tr>
<tr>
<td>add prod, sum, sum</td>
</tr>
<tr>
<td>[count] sub count, 1, count</td>
</tr>
<tr>
<td>[count] b loop</td>
</tr>
</tbody>
</table>

With Conditional Subtract:

<table>
<thead>
<tr>
<th>Without Cond Subtract:</th>
<th>With Conditional Subtract:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop (count = 1) (B)</td>
<td>Loop (count = 1) (B)</td>
</tr>
<tr>
<td>loop (count = 0) (K)</td>
<td>loop (count = 0) (K)</td>
</tr>
<tr>
<td>loop (count = -1) (B)</td>
<td>loop (count = 0) (K)</td>
</tr>
<tr>
<td>loop (count = -2) (B)</td>
<td>loop (count = 0) (K)</td>
</tr>
<tr>
<td>loop (count = -3) (B)</td>
<td>loop (count = 0) (K)</td>
</tr>
<tr>
<td>...</td>
<td>loop (count = 0) (K)</td>
</tr>
<tr>
<td>; loop doesn’t end</td>
<td>; loop ends</td>
</tr>
</tbody>
</table>

*Remember the Force* … of the pipeline, that is.

As seen on the lower left side of the preceding diagram, once the loop counter hits zero, the branch instruction is annulled (that is, not executed). BUT, the subtract still takes place, which decrements the counter – making it non-zero again.

Without pipelined instructions, this wouldn’t be a problem. Since our branch has five delay-slots, each slot executing another iteration of the loop, the counter will decrement past zero and branches will - once again - begin executing. If you did this in your code, congratulations, you created an endless loop!
Epilog – How and If you should use it

How to create epilog code

Go back and look at the code generated from our scheduling table on page 8-25. Our scheduling method creates code that includes a prolog along with the single-cycle loop but doesn’t create epilog code. If required, epilog code can be easily extracted from the above code by reducing the loop counter and explicitly writing the epilog – i.e. “clean-up” – code.

Here's a simple formula for creating the epilog code:

1. Epilog is the same length as Prolog.
2. For each Epilog execute-packet (EP), take the loop and remove both (a) the loop-overhead instructions; and, (b) the instructions from the associated prolog execute packet.

For example:

\[ e_1 = \text{loop} - p_1 - \text{loop overhead} \]

\[ e_1 = (\text{LDH} || \text{LDH} || \text{MPY} || \text{ADD} || \text{SUB} || B) - (\text{LDH} || \text{LDH}) - (\text{SUB} || B) \]

\[ e_1 = \text{MPY} || \text{ADD} \]

3. Also, don't forget to reduce the loop counter by the number of execute-packets in the epilog.

\[ \text{loop count} = 40 - 7 = 33 \]

Seven adds occur during the epilog, therefore, the loop only need generate thirty-three adds to achieve the total of forty.

<table>
<thead>
<tr>
<th>Prolog</th>
<th>Loop Kernel</th>
<th>Epilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: 1dh</td>
<td></td>
<td>1dh</td>
</tr>
<tr>
<td>p2: 1dh</td>
<td></td>
<td>1dh</td>
</tr>
<tr>
<td>p3: 1dh</td>
<td></td>
<td>1dh</td>
</tr>
<tr>
<td>p4: 1dh</td>
<td></td>
<td>1dh</td>
</tr>
<tr>
<td>p5: 1dh</td>
<td></td>
<td>1dh</td>
</tr>
<tr>
<td>p6: 1dh</td>
<td></td>
<td>1dh</td>
</tr>
<tr>
<td>p7: 1dh</td>
<td></td>
<td>1dh</td>
</tr>
</tbody>
</table>
**To Epilog or Not**

You have seen two different implementations of the software-pipelined dot-product algorithm. One contained epilog code, the other didn’t. Which version is best for you is dependent upon your application. Let’s examine each of these, plus one other variation to determine the tradeoffs.

**Prolog, Loop, and Epilog**

Here’s the simple diagram and table to indicate the differences between each variation. Using both prolog and epilog forces more instructions in your code. In the implementation the prolog and epilog are each eight cycles long. Prolog accounts for 27 instructions and epilog for another 12. Combining this with the 6 instructions in the loop and 1 to modify the count (from 40 → 33) totals forty-six 32-bit words.

<table>
<thead>
<tr>
<th>No Extra Loads (PLE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prolog</td>
</tr>
<tr>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
<tr>
<td># of Adds: 40</td>
</tr>
<tr>
<td>Code Size: 46w</td>
</tr>
<tr>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Notice three other items:

- The cycle count is 47 … or n+7, where n is the forty terms we were interested in.
- The loop count is actually 33. As described earlier, since our example required 40 additions, the loop provides 33 of them while the epilog contains the other 7.
- There are no superfluous loads. Forty are required and between the prolog and loop only 40 are performed.
Prolog and Loop (Eliminate Epilog)

Upon eliminating the epilog (or not extracting it from our scheduled code), three items are notable:

- Code size has dropped to 33 words.
- Cycle count remains the same.
- Seven extra loads are performed. Since the loop count must be raised to forty to achieve 40 adds, forty loads are performed in the loop after seven have already been performed in the prolog.

The compiler and assembly-optimizer tools can perform this code-space optimization under your instruction. The \texttt{–mh} compiler option specifies that extraneous loads are allowed. If used with an operand, e.g. \texttt{–mh 4}, it indicates that up to 4 extraneous byte-wide loads are allowed.
Loop Only

Similarly, prolog code may even be “rolled” into the loop.

In this variation, the code size drops dramatically. Now you’re left with only the six loop instructions and a very small overhead. Four instructions are required to zero out the summation and product registers (we don’t want to add in old stuff); five branches must occur before the loop code begins, to fill up the branch pipeline; and finally, the count must be increased by seven – up to 47.

Also, don’t forget to notice that seven extra loads are still performed.

Here’s a rough sample of what the code might look like:

```
b  loop
b  loop
b  loop
  || zero  m input register
  || zero  n input register
b  loop
  || zero  prod register
  || zero  sum register
b  loop
  || addk  modify count register

loop:  ldh
  ||  ldh
  ||  mpy
  ||  add
  ||  []  sub
  ||  []  b  loop
```
**Bottom Line**

Eliminating the epilog code as shown above decreases code size. Caution is required, though, if extraneous loads are not acceptable in your system. When epilog code is not used, final loop iterations perform superfluous load (LDH) instructions. From a functional standpoint this is O.K. since no additions use these values. On the other hand, if these extra LDHs are performing destructive reads – for example, from a FIFO – then epilog code is warranted.

In the case of the dot-product routine, eliminating (most of) the prolog and epilog – thus allowing extraneous loads – cuts the code size from 46 words down to just 16 words *without affecting the code’s speed*. During the next two chapters you’ll discover how to allow the compiler and assembly optimizer to make this tradeoff.

One final note, extra loads affect power dissipation. For many applications the very slight increase in power is meaningless. Load/store instructions dissipate more power than most other instructions since they must drive buses (obviously, external accesses are worse than internal). *For those where power is paramount, you must decide which is the better tradeoff: smaller code size or fewer load instructions.*
Optional Topics

**SPLOOP Buffer (C64x+)**

SPLOOP is a specialized hardware buffer within the C64x+ CPU which stores instructions during their first usage inside a software pipelined loop, then reissues them as required to implement the complete prolog, loop-kernel, and epilog. The advantages are discussed in Chapters 10 and 11. For complete details, see Chapter 7 in the *C64x/C64x+ CPU Reference Guide* (SPRU732.PDF).

---

**SPLOOP Buffer**

Exploit the regular pattern in software pipelined loops

<table>
<thead>
<tr>
<th>Prolog</th>
<th>ILC = loop cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel</th>
<th>SPLOOP 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>I1</td>
</tr>
<tr>
<td>I2</td>
<td>I2</td>
</tr>
<tr>
<td>I3</td>
<td>I3</td>
</tr>
<tr>
<td>I4</td>
<td>I4</td>
</tr>
<tr>
<td>I5</td>
<td>I5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Epilog</th>
<th>SPKERNEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td></td>
</tr>
</tbody>
</table>

- The SPLOOP buffer implements Software Pipelining in hardware, therefore, only one copy of the loop kernel must be stored in program memory
- How it works:
  - Once an instruction is loaded to the buffer, it doesn’t need to be re-fetched
  - SPLOOP instruction causes buffer to be filled and used
  - SPKERNEL signals loop end and to stop filling buffer
  - Inner Loop Counter (ILC) keeps track of iteration count

---

**SPLOOP Buffer**

Exploit the regular pattern in software pipelined loops

<table>
<thead>
<tr>
<th>Prolog</th>
<th>ILC = loop cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel</th>
<th>SPLOOP 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>I1</td>
</tr>
<tr>
<td>I2</td>
<td>I2</td>
</tr>
<tr>
<td>I3</td>
<td>I3</td>
</tr>
<tr>
<td>I4</td>
<td>I4</td>
</tr>
<tr>
<td>I5</td>
<td>I5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Epilog</th>
<th>SPKERNEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td></td>
</tr>
</tbody>
</table>

- Benefits
  - Reduce program memory accesses 75%
  - Reduce program size 14% (loop size 40%)
  - Reduce Instructions executed 8% (ld/st instructions by 14%)
- Tools
  - Compiler or Asm Optimizer uses SPLOOP for C64x+
  - Standard Assembly coding of SPLOOP is not recommended
- SPLOOP is also discussed in Chapters 10 and 11
Lab 8 Solutions

Lab 8 - Solution - (1) Linear Assembly

; for (i=0; i < count; i++)
; prod = m[i] * n[i];
; sum += prod; *** count becomes 20 ***

loop:       ldw   *p_m++, m
            ldw   *p_n++, n
            mpy   m, n, prod
            mpyh  m, n, prodh
            add   prod, sum, sum
            add   prodh, sumh, sumh
            [count] sub  count, 1, count
            [count] b   loop

; Outside of Loop
    add   sum, sumh, sum

Lab 8 - (2) Graph

Lab 8 Solutions

Lab 8 - Solution - (1) Linear Assembly

; for (i=0; i < count; i++)
; prod = m[i] * n[i];
; sum += prod; *** count becomes 20 ***

loop:       ldw   *p_m++, m
            ldw   *p_n++, n
            mpy   m, n, prod
            mpyh  m, n, prodh
            add   prod, sum, sum
            add   prodh, sumh, sumh
            [count] sub  count, 1, count
            [count] b   loop

; Outside of Loop
    add   sum, sumh, sum

Lab 8 - (2) Graph
Lab 8 - (2) Functional Units

Do we still have enough functional units to code this algorithm in a single-cycle loop?

Yes!

Lab 8 - (3) Registers

<table>
<thead>
<tr>
<th>Register File A</th>
<th>#</th>
<th>Register File B</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>B0</td>
<td>count</td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td>return address</td>
<td></td>
</tr>
<tr>
<td>&amp;a/ret value</td>
<td>A4</td>
<td>B4</td>
<td>&amp;x</td>
</tr>
<tr>
<td>a</td>
<td>A5</td>
<td>B5</td>
<td>x</td>
</tr>
<tr>
<td>count/prod</td>
<td>A6</td>
<td>B6</td>
<td>prodh</td>
</tr>
<tr>
<td>sum</td>
<td>A7</td>
<td>B7</td>
<td>sumh</td>
</tr>
</tbody>
</table>
Lab 8 - Schedule Algorithm

Lab 8 - Step 6: 'C6x Code

c0:  ldw .D1  *A4++,A5  
     ||  ldw .D2  *B4++,B5  

   c1:  ldw .D1  *A4++,A5  
     ||  ldw .D2  *B4++,B5  
     || [B0]  sub .S2  B0,1,B0  

   c2_3_4:  ldw .D1  *A4++,A5  
     ||  ldw .D2  *B4++,B5  
     || [B0]  sub .S2  B0,1,B0  
     || [B0]  B  .S1  loop  

   c5_6:  ldw .D1  *A4++,A5  
     ||  ldw .D2  *B4++,B5  
     || [B0]  sub .S2  B0,1,B0  

*** Single-Cycle Loop

loop:  ldw .D1  *A4++,A5  
     ||  ldw .D2  *B4++,B5  
     || [B0]  sub .S2  B0,1,B0  
     || [B0]  B  .S1  loop  
     ||  mpy .M1x A5,B5,A6  
     ||  mpyh .M2x A5,B5,B6  
     ||  add .L1  A7,A6,A7  
     ||  add .L2  B7,B6,B7  

C6000 Optimization Workshop - Software Pipelining
Software Pipelining Multi-Cycle Loops

As an optional discussion that further explores the concepts of software pipelining, let’s look at:

**Software Pipelining Multi-Cycle Loops**

This discussion is a great introduction to the additional materials found in the *TMS320C6000 Programmer's Guide* (SPRU198).

**Introduction**

Most of Chapter 7 discusses the basic software pipelining procedure. Here we introduce the issues involved in software pipelining multi-cycle loops. We begin by examining the Weighted Vector Sum algorithm since its three memory accesses prevent it from completing in a single-cycle loop (we only have two .D units).

Conveniently, the tools (compiler / assembly-optimizer) handle multi-cycle pipelining for you. Understanding the basic concepts, though, allows you to verify the tools effectiveness. Additionally, this knowledge may help you guide the tools to better efficiency.
Software Pipelining the Weighted Vector Sum

Let’s go step-by-step through the weighted vector sum pipelining example. You’ll see this algorithm requires multiple cycles to implement.

Step 1 – Weighted Vector Sum C code

Beginning with step one, Weighted Vector Sum C code:

```c
void WVS(short *c, short *b, short *a, short r, short n)
{
    int i;
    for (i=0; i < n; i++)
    {
        c[i] = a[i] + (r * b[i]) >> 15;
    }
}
```

**a, b:** input arrays  
**c:** output array  
**n:** length of arrays  
**r:** weighting factor

Notice that an array is output from the weighted vector sum routine unlike the dot product, which creates a single output value from two input arrays. It’s this output of the resultant vector component, combined with reading of two input array values that causes the memory bottleneck.
Step 2 – Linear Code

This is the core of the linear assembly routine.

### Step 2 - ‘C6000 Linear Asm Code

```assembly
loop:
  LDH *a++, ai
  LDH *b++, bi
  MPY r, bi, prod
  SHR prod, 15, sum
  ADD ai, sum, ci
  STH ci, *c++

[i] SUB i, 1, i
[i] B   loop
```
Step 3 – Dependency Graph

Following the dependency graph procedure, you can create:

The division between the A and B side was chosen, once again, to split up operations that required the same type of unit, to minimize the cross-paths, and to balance the usage of units on both sides. Notice that the A side ended up with two .D operations while the B side ended up with only one .D operation.
Step 4 – Allocate Resources

Allocating the resources shows what we’ve been talking about; the .D1 unit is allocated twice. All the other units except .M1 are allocated once.

 Allocate Functional Units

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓ ci</td>
<td></td>
<td>✓ ai</td>
<td></td>
<td>✓ i</td>
<td>✓ prod</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓ *c</td>
<td>✓ loop</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓ sum</td>
</tr>
</tbody>
</table>

The only way to get the use of two .D1 units for each loop iteration is to dedicate two processor cycles for each loop iteration. In this case you actually have all eight functional units at your disposal, although, they won’t all be needed for our dependency graph.

 When encountering multi-cycle loops, the term Iteration Interval describes the number of cycles it takes to complete one loop iteration. In the case of this Weighted Vector Sum example, the Iteration Interval is equal to two.
Step 5 – Scheduling Chart

What size Schedule Table should we use?

How long is the Prolog?

- What is the length of the longest path?
  - 10
- How many cycles per loop?
  - 2

Based on the length of our longest path, it should be 10 cycles long. Since we expect a 2-cycle loop (we need three .D units), the prolog is 10-2 = 8.

While this scheduling table would work, there’s actually a more convenient way to draw it.
Final Scheduling Chart

It’s more convenient to draw scheduling charts for multi-cycle loops as follows:

<table>
<thead>
<tr>
<th>Unit/cycle</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unit/cycle</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here, the number of “rows” is equal to the number of cycles in the loop (i.e. iteration interval) while the total number of cycles is based on the length of the prolog (as we saw earlier).

Why draw it out this way? Each instruction in our original algorithm gets executed how often? In the case of a two cycle loop, wouldn’t it get executed every two cycles? By drawing the table this way, once an instruction is scheduled, we just copy it all the way across the row (as we did for the single-cycle loop). Let’s look at an example …
Start Long and Early

As described earlier, begin scheduling the longest path first. In this case the longest path is obvious, it the one that contains the “load/multiply/shift/add”. Add them to the chart in that order:

### Step 5 - Create Scheduling Chart

<table>
<thead>
<tr>
<th>Unit</th>
<th>cycle</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D2</td>
<td>LDH bi</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

Next, add the final instruction of the longest path through the dependency graph, STH.

### Step 5 - Create Scheduling Chart

<table>
<thead>
<tr>
<th>Unit</th>
<th>cycle</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHR sum</td>
<td>*</td>
</tr>
<tr>
<td>.M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M2</td>
<td>MPY prod</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>.D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D2</td>
<td>LDH bi</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unit</th>
<th>cycle</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHR sum</td>
<td>*</td>
</tr>
<tr>
<td>.M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M2</td>
<td>MPY prod</td>
<td></td>
<td></td>
<td></td>
<td>STH c[j]</td>
<td>*</td>
</tr>
<tr>
<td>.D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STH c[j]</td>
</tr>
</tbody>
</table>
After scheduling the longest path, we must go back and complete the other path. In our example, we need to schedule the additional LDH instruction. It must occur five cycles before the ADD instruction in cycle 8.

### Step 5 - Create Scheduling Chart

<table>
<thead>
<tr>
<th>Unit\cycle</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unit\cycle</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Conflict</td>
</tr>
<tr>
<td>.M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Once scheduled an instruction is repeated across its row. This causes a conflict in cycle 9 on the .D1 unit.

How do we eliminate the conflict?
- STH can’t be moved earlier since it must execute after ADD.
- STH can’t be moved later otherwise it won’t be completed in the two cycle loop (cycles 8-9).
This leaves you with two options:

**Conflict Solution**

*Here are two possibilities ... Which is better?*

<table>
<thead>
<tr>
<th>Unit/cycle</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>.D1</td>
<td></td>
<td>LDH ai</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D2</td>
<td></td>
<td>LDH bi</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unit/cycle</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D1</td>
<td></td>
<td>LDH ai</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>.D2</td>
<td></td>
<td>LDH ai</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

The two options include moving “LDH ai” to cycle 2 or moving it onto the .D2 unit. Our suggestion is to move it to cycle 2 and leave it on the .D1 unit. If you move it onto the .D2 unit, modify the dependency graph and verify that no crosspath restrictions exist.

**Step 5 - Create Scheduling Chart**

<table>
<thead>
<tr>
<th>Unit/cycle</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L1</td>
<td></td>
<td></td>
<td>ADD ci</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D1</td>
<td></td>
<td>LDH ai</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>.D2</td>
<td></td>
<td>LDH bi</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unit/cycle</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D1</td>
<td></td>
<td>LDH ai</td>
<td></td>
<td></td>
<td>STH c[i]</td>
</tr>
<tr>
<td>.D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As a side note, “LDH ai” couldn’t be moved to either cycle zero or cycle one. Obviously, moving it to cycle 0 would introduce the same conflict as in cycle three. Also, moving it into zero or one would create a new problem called *Live-too-Long*.

To explain live-to-long let’s examine the ill effects of scheduling “LDH ai” in cycle 0. Remember that if LDH is scheduled in cycle 0, it also occurs in cycle 2, 4, 6, and 8. Wait! If another LDH is attempted in cycle 2, this value will be valid when completing the ADD in cycle eight. In other words, we end up using the wrong value in our calculation – we tried to hold the original “LDH ai” value *alive-too-long* in the register. We’ll cover *Live-too-Long* more later.

**Scheduling Branch and Subtract**

To complete the loop schedule, don’t forget to include the Branch and SUB instructions.

---

**Step 5 - Create Scheduling Chart**

<table>
<thead>
<tr>
<th>Unit/cycle</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADD ci</td>
</tr>
<tr>
<td>.L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S1</td>
<td></td>
<td>[i] B</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>.S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.D1</td>
<td></td>
<td>LDH ai</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>.D2</td>
<td></td>
<td>LDH bi</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unit/cycle</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.L2</td>
<td></td>
<td>[i] SUB i</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>.S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.S2</td>
<td></td>
<td></td>
<td>SHR sum</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>.M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.M2</td>
<td></td>
<td></td>
<td>MPY prod</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>.D1</td>
<td></td>
<td></td>
<td>LDH ai</td>
<td></td>
<td>STH c[i]</td>
</tr>
<tr>
<td>.D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Optimizing Performance of the WVS

How many cycles does it take to complete the loop?

<table>
<thead>
<tr>
<th>Performance</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD ci</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>STH c[i]</td>
</tr>
</tbody>
</table>

Can we do better than 1 result / 2 cycles?

How can you optimize the performance of the Weighted Vector Sum routine? Is there any way to perform the algorithm (i.e. calculate and store a new value for \( C_i \)) in less than two cycles?

The answer to this lies in functional unit usage. The .D resources are in greatest demand. We must perform 3 .D unit actions during the loop (LDH, LDH, STH). If we code this into two cycle loops, we end up with an extra, unused .D unit in one of the two cycles.

Since there is an unused resource, it must be possible to optimize further. If we decided to calculate two elements in the array in each loop iteration, how many cycles would this need? Well, this implies that 4 LDHs and 2 STHs be completed. Let’s see, that’s a requirement of six .D functional units. Six .D functional units would require two loop iterations.
Does this “buy” us anything? The short answer is: yes. Instead of requiring two cycles for one loop iteration, we now require three cycles for two loop iterations. This cuts the cycle time by one third!

### By Unrolling the WVS Algorithm

<table>
<thead>
<tr>
<th></th>
<th>Results per loop</th>
<th>.D units</th>
<th>Cycles per loop</th>
<th>Cycles per Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Unrolled</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>1.5</td>
</tr>
</tbody>
</table>

**Better yet:**

Unrolling the loop and using LDW maximizes the use of .D units.

<table>
<thead>
<tr>
<th></th>
<th>.D units</th>
<th>Cycles per loop</th>
<th>Cycles per Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Unrolled</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

You can even cut time again by using LDW in combination with loop unrolling to achieve a rate of 1 cycle/result. Wow! Here’s the linear assembly code:

```assembly
; wvs( &c, &a, &b, r, N )
; c[i] = a[i] + (r * b[i] ) >> 15 for i=1,N
; short a[...], b[...], c[...], r, N, i
; Arrays a,b,c must all begin on word boundaries.
.def _wvs ; entry point to weighted vector sums

.wvs:
*    ptr_c ; pointer to output array c
*    ptr_a ; pointer to input array a
*    ptr_b ; pointer to input array b
*    r    ; input halfword scale factor r
*    N    ; input loop count

loop:  ldw  *ptr_b++,b       ; load WD of 2 HWds from array b
       ldw  *ptr_a++,a       ; load WD of 2 HWds from array a
       smpy r,b,rb           ; rb = r*b lower HW
       smpylh r,b,rbh         ; rbh = r*b upper HW
       shru rb,16,rb          ; shift rb to LS halfword (MS HW = 0)
       clr rbh,0,15,rbh       ; rbh = MS halfword (LS HW = 0)
       add rb,rbh,rb         ; rb = rbh + rb as 2 HWds
       add2 a,rb,c           ; c = a + rb as 2 HWds
       stw c,*ptr_c++        ; store WD of 2 HWds into array c
[N]   sub N,1,N            ; decrement loop count
[N]   b loop               ; repeat loop until N=0
*    end of loop
```
Multi-Cycle Loops (and the Minimum Iteration Interval)

We discussed iteration interval earlier in this chapter. You might remember that it means the number of cycles in the loop; or more technically, the number of cycles between the start of each loop iteration.

Minimum iteration interval describes the smallest number of cycles possible in the loop. In the fixed-point dot-product algorithm, the minimum iteration interval equaled one. In the weighted-vector-sum algorithm, it was two – this was required by the need for three .D units.

What reasons could force your loop to be greater than one?

<table>
<thead>
<tr>
<th>What Requires Multi-Cycle Loops?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Four reasons:</td>
</tr>
<tr>
<td>1. Resource Limitations</td>
</tr>
<tr>
<td>2. Live Too Long</td>
</tr>
<tr>
<td>3. Loop Carry Path</td>
</tr>
<tr>
<td>4. Double Precision (FUL &gt; 1)</td>
</tr>
</tbody>
</table>

The worst case of these four constraints determine the smallest iteration interval - Minimum Iteration Interval (MII)

Resource Limitations

We’ve already demonstrated the first – Resource Limitations – with the weighted-vector-sum example. Needing additional functional units is one of the most common reasons for multi-cycle loops. Any time your algorithm requires more than eight instructions your loop will be greater than one.

Resource limitations also include: crosspaths, registers, and memory accesses (bus conflicts).
Live Too Long

We briefly discussed *Live Too Long* during the weighted-vector-sum example, but we’ll look at it in much greater depth here. Here’s a simple example:

```
0     LDH ai
1     LDH
2     LDH
3     LDH
4     LDH
5     a0 valid
6     a1
      SHR
      x0 valid
      ADD

Oops, rather than adding
  a0 + x0
we got
  a1 + x0
```

This *Live-Too-Long* problem is caused by (what’s called) a *Split-Join-Path*. The value “ai” is used along two split paths that are later joined together. The two separate paths must join together within the same iteration interval otherwise the error shown above occurs. In the above example, the right path takes 6 cycles while the left takes only 5. Since the iteration interval is only one (1), a problem exists.
What’s the solution?

You can always increase the iteration interval to solve this problem:

**Live Too Long - 2 Cycle Solution**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDH ai</td>
<td>LDH</td>
<td>LDH</td>
<td>LDH</td>
<td>a0 valid</td>
<td>x0 valid</td>
<td>ADD</td>
</tr>
<tr>
<td>LDH a1</td>
<td>LDH</td>
<td>LDH</td>
<td>LDH</td>
<td>a0 valid</td>
<td>x0 valid</td>
<td>ADD</td>
</tr>
</tbody>
</table>

**Works!**

But what’s the drawback?

2 cycle loop is slower

Yes, this solves the problem, but it also can reduce performance. Another method allows us to keep a single-cycle iteration interval for this example:

**Live Too Long - 1 Cycle Solution**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDH ai</td>
<td>LDH</td>
<td>LDH</td>
<td>LDH</td>
<td>a0 valid</td>
<td>a1</td>
<td>MV b</td>
<td>b valid</td>
</tr>
<tr>
<td>b valid</td>
<td>SHR</td>
<td>x0 valid</td>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Using a temporary register solves this problem without increasing the Minimum Iteration Interval**

By using a temporary register and the MV instruction, you can “even-up” the paths. Notice how the scheduling table helps to analyze these problems.
Loop Carry Path

Another issue that increases the Minimum Iteration Interval is. (Funny how all of these seem to be three word names?)

While Loop Carry Path existed for the ‘C62x you’ll notice it more often on the ‘C67x due to the longer instruction latencies for floating-point and double-precision instructions.

We’ll demonstrate Loop Carry Path with two examples.

IIR Filter Example

The following IIR filter is a simple Loop Carry Path example. The ”Loop” we’re talking about in Loop Carry Path isn’t the code loop, but rather a data loop. Feeding the output of the filter back into the algorithm creates a data path (i.e. data loop) that carries the result into the next iteration of the loop.

The iteration interval must be greater than or equal to the Loop Carry Path. Therefore, the minimum iteration interval is the greater of:

- Resource Constraints
- Live Too Long
- Loop Carry Path

In the case of this example, the minimum iteration interval ends up equaling the longest loop-carry-path – that is, nine.
Can we optimize this to reduce the loop carry path?

The key is to minimize the length of the data path that carries over from one loop iteration to the next. This case provides a good example of how this might be done:

### Loop Carry Path - IIR Example

**IIR Filter Loop**
\[ y_0 = a_1 \times x_1 + b_1 \times y_1 \]

**Min Iteration Interval**
- Resource = 2
  (need 3 .D units)
- New Loop Carry Path = 3
  (3 = 2 + 1)
- therefore, MII = 3

Since \( y_0 \) is stored in a CPU register, it can be used directly by MPY

(after the first loop iteration)

While the output must be stored, do we have to use the stored value? Why, it’s already in a register (the ADD destination register). Not only does this cut the STH from the loop carry path, but also the LDH. This triples the performance of our IIR filter.
Dot Product Example

Looking back at the fixed-point dot product, was there a loop carry path? That is, was there a result that was carried forward from one loop iteration to the next?

Yes, the running sum is carried forward from one iteration to the next. You didn’t need to worry about this, though, since the loop carry path was only ‘one’. It didn’t increase the minimum iteration interval.
What about the floating-point dot product, though?

Unfortunately, this is a different story:

**Floating-Point Dot-Product Example**

How does this change for floating-point?

It's now “4” ...

Min Iteration Interval

Resource = 1
Loop Carry Path = 4
⇒ MII = 4

Due to the increased instruction latencies (delay slots) for floating-point math, the loop-carry path jumps from one to four. Therefore, the minimum iteration interval also jumps to four.

There are two techniques you can apply to optimize your algorithm: *loop unrolling* or *multiple register assignment*.

**Loop Unrolling**

Since the minimum iteration interval must be four (in our example) why not unroll the loop to calculate four output terms per loop iteration. This will get the rate down to one output/cycle.

Of course, this method consumes more registers. This, plus solving split-join-paths can put pressure on your register resources – called “register pressure”.

---

*C6000 Optimization Workshop - Software Pipelining*
Double Precision Instructions (FUL > 1)

Some of the double-precision instructions ‘tie-up’ a functional unit for multiple cycles. For example, the MPYDP instruction holds the functional unit for four cycles. Since the functional unit is tied up for four cycles, this sets the minimum iteration interval to four.

Here’s a simple example loop based on the MPYDP:
A better way to demonstrate this is using the scheduling table for a four cycle loop:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>5</th>
<th>9</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>.M1</td>
<td>MPYDP</td>
<td>MPYDP</td>
<td>MPYDP</td>
<td>MPYDP</td>
</tr>
</tbody>
</table>

Since the MPYDP instruction has a functional unit latency (FUL) of “4”, .L1 cannot be used until the fifth cycle.

Hence, $\text{MII} \geq 4$

It’s now easy to see that the MPYDP instruction can be repeated at four-cycle intervals.

By the way, you can also see the MPYDP output in is available for use beginning in cycle 11. Remember the MPYDP is a “4.9” instruction; that is, four cycle Functional Unit Latency (FUL) and nine delay slots.
(Optional) Single vs. Multiple Register Assignment

Single vs. multiple assignment is part of an optimization that can be used to minimize code size and maintain performance on functions like the floating-point dot-product.

**Single Assignment**
- Single assignment requires that no registers are read which have pending results.

---

Single Assignment:

<table>
<thead>
<tr>
<th>SA:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LDW .D1</td>
<td>*A0, A1</td>
</tr>
<tr>
<td>NOP</td>
<td>4</td>
</tr>
<tr>
<td>MPY .M1</td>
<td>A1, A2, A3</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>SHR .S1</td>
<td>A3, 15, A3</td>
</tr>
<tr>
<td>ADD .L1</td>
<td>A3, A4, A4</td>
</tr>
</tbody>
</table>

- Reads current value -- var(n)
- Uses current value -- var(n)

---

**Multiple Assignment**
- Single assignment requires that no registers are read which have pending results
- Multiple assignment allows a register to hold multiple values at different times (register “TDM”)

---

Multiple Assignment:

<table>
<thead>
<tr>
<th>MA:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD .S1</td>
<td>A7, A8, A0</td>
</tr>
<tr>
<td>LDW .D1</td>
<td>*A0, A1</td>
</tr>
<tr>
<td>MPY .M1</td>
<td>A1, A2, A3</td>
</tr>
<tr>
<td>NOP</td>
<td>3</td>
</tr>
<tr>
<td>SHR .S1</td>
<td>A3, 15, A3</td>
</tr>
<tr>
<td>ADD .L1</td>
<td>A3, A4, A4</td>
</tr>
</tbody>
</table>

- Reads current value -- X(n)
- Uses last value -- X(n-1)

Register A1 contains two values:
1. During MPY: A1 = X(n-1)
   and “virtually” holds X(n)
2. By SHR: A1 = X(n)
Floating-Point Example Using Multiple Assignment

Using multiple assignment to registers (in this case A2) allows the same performance as unrolling the dot-product routine (discussed earlier in the chapter), while minimizing code size. Additionally, a one-cycle loop allows for more flexibility than a four-cycle loop. (Note: The ‘C6000 Programmer’s Guide (SPRU198) discusses this method of using staggered results. Please refer to it for more information.)

---

**Optimize Float \textit{dotp} Using Multiple Assignment**

<table>
<thead>
<tr>
<th>dotp:</th>
<th>zero</th>
<th>A2</th>
</tr>
</thead>
</table>

**loop:**
- ldw *in0, A0
- ldw *in1, A1
- mpy A0, A1, A8
- addsp A2, A8, A2
- sub B0, 1, B0
- [B0] b loop

- mv A2, A4
- addsp A2, A4, A4
- mv A2, A6
- addsp A2, A6, A6
- nop 3
- addsp A4, A6, A4
- nop 3
- stw A4, *output

- Here's another floating-point dot-product solution
- Here's the loop kernel and epilog for 10 element arrays: (40,39,38,...) * (1,2,3,...)
- It uses register TDM as opposed to loop-unrolling (shown earlier) to increase performance
- Let's examine this code step-by-step. The key is to examine how A2 is used to juggle multiple values (Multiple-Assignment)

In our example, the results of the ADDSP (3 delay slots) end up being staggered into the summation register (A2). The following slides chart out the results of this code. The diagram below charts the registers used in the preceding code example on one axis, with the cycle time charted along the other.
Charting Our Example

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>MV</td>
<td>ADDSP</td>
<td></td>
</tr>
<tr>
<td>A8 (prod)</td>
<td>40</td>
<td>78</td>
<td>114</td>
<td>148</td>
<td>180</td>
<td>210</td>
<td>238</td>
<td>264</td>
<td>288</td>
<td>310</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- These are the registers used by the routine
- Four A2 rows indicate the 4 virtual values it holds during the routine

Notice that we begin the diagram at the start of the loop and continue through the epilog. It’s assumed that the required prolog code has already primed-the-loop prior to our chart.

As mentioned, the ADDSP has 3 delay slots, thus it’s result won’t show up in A2 until cycle 5.

First Accumulation Appears in Cycle 5

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>MV</td>
<td>ADDSP</td>
<td></td>
</tr>
<tr>
<td>A8 (prod)</td>
<td>40</td>
<td>78</td>
<td>114</td>
<td>148</td>
<td>180</td>
<td>210</td>
<td>238</td>
<td>264</td>
<td>288</td>
<td>310</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A21</td>
<td>ADDSP in Loop</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The same occurs for the next three cycles. It’s easy to see why the *Programmer's Guide* calls this “using staggered results”.

### The Next Three Follow Suit

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>MV</td>
<td>ADDSP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A8 (prod)</td>
<td></td>
<td>40</td>
<td>78</td>
<td>114</td>
<td>148</td>
<td>180</td>
<td>210</td>
<td>238</td>
<td>264</td>
<td>288</td>
<td>310</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A2₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>A2₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>78</td>
</tr>
<tr>
<td>A2₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>114</td>
</tr>
<tr>
<td>A2₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>148</td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These staggered results is also *multiple-assignment* of values to register A2. That is, in cycle 2 we’re putting a different value into A2, while it still contains the previous value (conceptually, that is). You might think of it as the loop throwing 4 balls into the air, so that A2 has to juggle them.

Cycle 5 continues with the staggered results. Again, we’re keeping four, intermediate, running sums in the single A2 register.
Cycles 6-8 Create the Next 3 Running Sums

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>MV</td>
<td>ADDSP</td>
<td></td>
</tr>
<tr>
<td>A8 (prod)</td>
<td>40</td>
<td>78</td>
<td>114</td>
<td>148</td>
<td>180</td>
<td>210</td>
<td>238</td>
<td>264</td>
<td>288</td>
<td>310</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>78</td>
<td></td>
<td></td>
<td>288</td>
<td></td>
</tr>
<tr>
<td>A23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>A24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>148</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cycles 9 and 10 complete the actual loop iterations, though you’ll see that the summation results keep “falling” into A2 for the next few cycles (11 - 14).

Cycles 9-10 Begin Iteration on Running Sums

<table>
<thead>
<tr>
<th>Cycle</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15 - 17</th>
<th>18</th>
<th>19 - 21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>MV</td>
<td>ADDSP</td>
<td>MV</td>
<td>ADDSP</td>
<td>3 NOP</td>
<td>ADDSP</td>
<td>3 NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>A8 (prod)</td>
<td>264</td>
<td>288</td>
<td>310</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>220</td>
<td></td>
<td></td>
<td>508</td>
<td></td>
</tr>
<tr>
<td>A22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>288</td>
<td></td>
<td></td>
<td>598</td>
</tr>
<tr>
<td>A23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>352</td>
</tr>
<tr>
<td>A24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>148</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>352</td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To keep the result landing in A2 during cycle 11 from being lost (i.e. overwritten), it’s value is moved to another register. The following cycle adds this to the final results of the ‘loop’ code.

### Loop Ends in Cycle 10

<table>
<thead>
<tr>
<th>Cycle</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15-17</th>
<th>18</th>
<th>19-21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>MV</td>
<td>ADDSP</td>
<td>MV</td>
<td>ADDSP</td>
<td>3 NOPs</td>
<td>ADDSP</td>
<td>3 NOPs</td>
<td>NOP</td>
</tr>
<tr>
<td>A8 (prod)</td>
<td>264</td>
<td>288</td>
<td>310</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A2₁</td>
<td></td>
<td>220</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>508</td>
</tr>
<tr>
<td>A2₂</td>
<td></td>
<td>288</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>598</td>
</tr>
<tr>
<td>A2₃</td>
<td></td>
<td>352</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2₄</td>
<td>148</td>
<td></td>
<td></td>
<td>412</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td>352</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>764</td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Clean-up code runs in cycles 11-21.
- This code retrieves each sum as it falls into A2, then adds them together.
- For example, the “764” sum appears in cycle 16.

Similarly, the MV and ADDSP use the last summations from the loop.

### Final Intermediate Sum Appears in Cycle 18

<table>
<thead>
<tr>
<th>Cycle</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15-17</th>
<th>18</th>
<th>19-21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Loop</td>
<td>Loop</td>
<td>Loop</td>
<td>MV</td>
<td>ADDSP</td>
<td>MV</td>
<td>ADDSP</td>
<td>3 NOPs</td>
<td>ADDSP</td>
<td>3 NOPs</td>
<td>NOP</td>
</tr>
<tr>
<td>A8 (prod)</td>
<td>264</td>
<td>288</td>
<td>310</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A2₁</td>
<td></td>
<td>220</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>508</td>
</tr>
<tr>
<td>A2₂</td>
<td></td>
<td>288</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>598</td>
</tr>
<tr>
<td>A2₃</td>
<td></td>
<td>352</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2₄</td>
<td>148</td>
<td></td>
<td></td>
<td>412</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td>352</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>764</td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1106</td>
</tr>
</tbody>
</table>

For example, the “764” sum appears in cycle 16.
The epilog code just keeps juggling values, and adding intermediate results until they’ve all been combined into the final result – as seen in cycle 22.

### Final Result Available in Cycle 22

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15 - 17</th>
<th>18</th>
<th>19 - 21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A8 (prod)</td>
<td>264</td>
<td>288</td>
<td>310</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>A2₁</td>
<td>220</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A2₂</td>
<td>288</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A2₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>352</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A2₄</td>
<td>148</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>412</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>352</td>
<td>+</td>
<td>764</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>508</td>
<td>+</td>
<td>1106</td>
<td></td>
</tr>
</tbody>
</table>

The final result of 10 loop iterations = “1870”
Introduction

As we have seen so far in this workshop, the 'C6000 C Compiler is very good at optimizing code. Usually, the most important thing to do in order to get optimized code is to simply turn on the optimizer using the –o3 build option. If the compiler doesn't create code that is as optimized as you think it should be, it might be that the compiler simply doesn't have enough information. In order to get the best code that we can from the compiler, we need to learn how to interact with it to help it create the code that we want.

Outline

◆ Provide Yourself with More Info (-pm)
◆ Program Level Optimization (Aliasing)
◆ Restrict Memory Dependencies (Intrinsics)
◆ Access Hardware Features (Pragmas)
◆ Provide Compiler with more Info
◆ Use Optimized Libraries
◆ Summary – Coding Methodologies
◆ Lab 9
# Chapter Topics

- **Before We Get Started** .......................................................................................................... 9-3
- **Provide Yourself with More Information** ................................................................................................ 9-4
  1. Know your goal .............................................................................................................................. 9-4
  2. Read the information provided by the tools.................................................................................. 9-7
  3. Use the Compiler Consultant (discussed further during lab)....................................................... 9-7
- **Program Level Optimization** .................................................................................................... 9-8
- **Memory Aliasing** ..................................................................................................................... 9-9
- **Intrinsics** ........................................................................................................................................ 9-11
- **Provide the Compiler with More Information (Pragmas)** ............................................................ 9-19
  1. Program Level Optimization ......................................................................................................... 9-19
  2. UNROLL ....................................................................................................................................... 9-20
  3. MUST_ITERATE ........................................................................................................................... 9-20
  4. Static Keyword ............................................................................................................................ 9-21
  5. DATA_ALIGN ............................................................................................................................... 9-22
  6. Use _nassert() ............................................................................................................................ 9-23
- **Use Optimized Libraries** ............................................................................................................. 9-26
- **Summary – Coding Methodologies** .......................................................................................... 9-30
Before We Get Started

Before we get started, let’s remember our discussion from Chapter 2:

- Turn on Optimization option (-o3)
- Turn off full source-level debug options (-g)

This is the single, easiest, biggest optimization trick available.

First, Turn on the Optimizer

- As we have seen, the optimizer can work miracles, but...
- What else can we do with C after turning on the optimizer?

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-mv6700</td>
<td>Generate 'C67x code ('C62x is default)</td>
</tr>
<tr>
<td>-mv67p</td>
<td>Generate 'C672x code</td>
</tr>
<tr>
<td>-mv6400</td>
<td>Generate 'C64x code</td>
</tr>
<tr>
<td>-mv6400+</td>
<td>Generate 'C64x+ code</td>
</tr>
<tr>
<td>-mv6740</td>
<td>Generate 'C674x code</td>
</tr>
<tr>
<td>-mv6600</td>
<td>Generate 'C66x code</td>
</tr>
<tr>
<td>-o3</td>
<td>Invoke optimizer (-o0, -o1, -o2/-o, -o3)</td>
</tr>
<tr>
<td>-k</td>
<td>Keep asm files, but don't interlist</td>
</tr>
</tbody>
</table>

Debug

- g
- ss

Don’t use these options

Optimize (release)
Provide Yourself with More Information

1. Know your goal

Before you start coding, understand:

- What is your real-time requirement?
- What is the best that can be achieved?

Without a clear understanding of your target, it is hard to go about optimizing your code, or knowing when you are done.

Why Spend Time Optimizing If…

… real-time performance has already been met with plain C code

… the algorithm cannot be improved

- Code should be written only after analyzing the optimum performance.
- From this analysis you can answer the question, “Can additional time spent optimizing translate to improved performance?”
What is Optimum Performance?

What Does Optimum Mean

Optimization is defined as:
- An act, process, or methodology of making something fully perfect, functional, or effective as possible.
- Continuous process of refinement in which code being optimized executes faster and takes fewer cycles, until a specific objective is achieved (real-time execution).

How do we know when to stop?
- **Optimum**: Greatest degree attained or attainable under specified or implied conditions.
- **Bottom Line**: How do we figure out how fast a given algorithm can run on a given architecture?

How do we determine the Optimum?

1. Analysis is specific to the algorithm.

2. Raw performance for a given computation loop depends on:
   - **a)** Size of the data that is being worked upon (shorts, bytes).
   - **b)** Number of multiply operations needed.
   - **c)** Number of loads and stores needed.
   - **d)** Number of logical operations needed.

3. For “M” number operations, how long should it take to perform this algorithm?

4. When doing multiple iterations:
   - Do data dependencies prevent putting calculations in parallel?
   - Can the algorithm be optimized further by putting multiple iterations in parallel (i.e. loop unrolling)?

For example, remember the Dot Product from Ch 3...
Provide Yourself with More Information

**Dot Product Example**

\[ Y = \sum_{i=1}^{\text{count}} \text{coeff}_i \times x_i \]

```c
for (i = 1; i < count; i++) {
    Y += coeff[i] * x[i];
}
```

| MVK .S1 40,A2 | Loop: 1 |
| LDH .D1 *A5++, A0 | (1) |
| LDH .D1 *A6++, A1 | (1) |
| NOP | (4) |
| MPY .M1 A0,A1,A3 | (1) |
| NOP | (1) |
| ADD .L1 A3,A4,A4 | (1) |
| SUB .L1 A2,1,A2 | (1) |
| [A2] B .S1 loop | (1) |
| NOP | (5) |
| STH .D1 A4,*A7 | (1) |

Where did the 28 come from?

- 4 short Loads
- 2 short MACs

per C62x cycle = 4 loads / 2 MAC's

therefore it should take ~ 20 cycles

We'll see another example later in this chapter.

**And for the C64x?**

<table>
<thead>
<tr>
<th>Dot-Product Terms</th>
<th>C62x</th>
<th>C64x</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 term</td>
<td>28</td>
<td>19</td>
</tr>
<tr>
<td>256 term</td>
<td>136</td>
<td>79</td>
</tr>
</tbody>
</table>

- While 28 cycles is great, the C64x is even better
- C64x runs more than twice as fast per cycle as C62x
- Profiling results in a few more cycles than shown here, due to overhead such as function call/return
2. Read the information provided by the tools
   - Add extra info to resulting assembly file.
   - Safe for production code --- no performance impact.

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-os</td>
<td>Show source code after high-level optimization using C/C++ syntax</td>
</tr>
<tr>
<td>-mw</td>
<td>Provides extra information on software pipelined loops</td>
</tr>
</tbody>
</table>

3. Use the Compiler Consultant (discussed further during lab)
   - The Consultant can direct you where to use other optimizations discussed in this chapter.
   - See Application Note SPRAA14.
   - Invoke this with the --consultant option.
   - Not available in CCSv4 or CCSv5.
Program Level Optimization

There are many C compiler options we have not covered in this course that could help you improve your code performance and size.

Program Level Optimization (-pm)

- pm -op2 -o3 -mv6700

- pm is critical in compiling for maximum performance
- pm creates a temp.c file which includes all C source files, thus giving the optimizer a program-level optimization context
- -opn describes a program's external references

By using the –pm and –op2 compiler options along with the –o3 (file optimization), the highest optimization is achieved by the compiler. The –pm, program level optimization allows the compiler to see the entire program thus “know more” about our application. As discussed earlier, the more the compiler knows about your application, the better job it can do optimizing your code.

While –pm is a great optimization option, it cannot perform miracles. That is, it provides the compiler visability into all the C source code in the project, it just cannot "see" into object files.

Then fine print about -pm:
- -pm requires the use -o3
- Cannot be used as file or function specific option
- Without knowing which -op, option to use, TI couldn't use-pm in default Release config
- Unfortunately, -pm cannot provide optimizer with visibility into object code libraries
- External References:
  - For example, if your program modifies a global variable from another code module, -op2 cannot be used
  - Similarly, if your code calls a function in an external module (who’s source isn’t visible to the optimizer), -op2 cannot be used (and will be overridden)
Memory Aliasing

Aliasing occurs when you can access a single object in multiple ways.

What is Aliasing?

```c
int x;
int *p;

main()
{
  p = &x;
  x = 5;
  *p = 8;
}
```

One memory location, two ways to access it: `x` and `*p`

Note: This is a very simple alias example. The compiler doesn't have any problem disambiguating an alias condition like this.

Aliasing?

```c
void fcn(*in, *out)
{
  LDW *in++, A0
  NOP 4
  ADD A0, 4, A1
  STW A1, *out++
}
```

```
| in  | a  | b  | c  | d  | e  | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>out</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>out0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>out1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>out2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

C6000 Optimization Workshop - Advanced C Topics for Code Speed 9 - 9
Looking at the `fcn()` pseudo C-function, wouldn’t it run faster if it were software pipelined, as shown?

### Aliasing?

**What happens if the function is called like this?**

```c
void fcn(*in, *out)
{
  LDW *in++, A0
  ADD A0, 4, A1
  STW A1, *out++
}
```

```c
in + 4
```

Would the software pipelined code work correctly?

![Code Snippet]

The problem is, if both `*in` and `*out` are passed with overlapping memory ranges, the “fast” solution would provide the wrong answers. This is the type of aliasing the compiler worries about. What can you do?

### Alias Solutions

1. **Compiler solves most aliasing on its own.**
   - If in doubt, the result will be correct even if the most optimal method won’t be used

2. **Program Level Optimization (-pm -o3)**
   - Provide compiler visibility to entire program

3. **No Bad Aliasing Option (-mt)**
   - Tell the compiler that no bad aliases exist
   - See Compiler User’s Guide for definition of “bad”
   - Previous weighted vector summation example performance was increased by 5x

4. **Restrict Keyword (ANSI C)**
   - Similar to `-mt`, but on a array-level basis

   ```c
   void fcn(short *restrict in, short *out)
   ```

Along with these suggestions, we highly recommend you check out:
- TMS320C6000 Programmer’s Guide
- TMS320C6000 Optimizing C Compiler User’s Guide
Intrinsics

Intrinsic operations are automatically inlined into the code. The inlining happens automatically whether or not you use the optimizer.

| _add2( ) | _sadd( ) |
| _clr( ) | _set( ) |
| _ext/u( ) | _smpy( ) |
| _lmbd( ) | _smphy( ) |
| _mpy( ) | _ssh1( ) |
| _mpyh( ) | _ssub( ) |
| _mpylh( ) | _subc( ) |
| _mpyhl( ) | _sub2( ) |
| _nassert( ) | _sat( ) |

Refer to C Compiler User’s Guide for more information

- Think of intrinsic functions as a specialized function library written by TI
- #include <c6x.h> has prototypes for all the intrinsic functions
- Intrinsics are great for accessing the hardware functionality which is unsupported by the C language
- To run your C code on another compiler, download C62x intrinsic C-source:
  spra616.zip
- int x, y, z;
  z = _lmbd(x, y);

Here’s another example using intrinsics uses casting to call the function with the correct data-types. In the case of ADD2, the output and two inputs should be 32-bit integers (each containing two 16-bit integers). For a full list of the C6000 intrinsics – and their prototypes – please refer to the TMS320C6000 Optimizing C Compiler Users Guide.

Using Intrinsics with Casting

```c
short a[50], b[50];
int y;

y = _add2(*(int *)a, *(int *)b);
```
Using the _mpyh() intrinsic gives you access to the MPYH hardware instruction functionality.

### Comparing the Coding Methods

<table>
<thead>
<tr>
<th>C Code</th>
<th>C Code Using Intrinsics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>y = a * b;</code></td>
<td><code>y = _mpyh (a, b);</code></td>
</tr>
</tbody>
</table>

- **Intrinsics...**
  - Can use C variable names instead of register names
  - Are compatible with the C environment
  - Adhere to C’s function call syntax
The ‘C67x also has a set of intrinsics to support its special capabilities.

### ‘C67x Intrinsics

<table>
<thead>
<tr>
<th>Single Precision</th>
<th>Double Precision</th>
<th>Integer</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>_fabsf()</td>
<td>_fabs()</td>
<td>_mpyid()</td>
<td>_dpint()</td>
</tr>
<tr>
<td>_rcpsp()</td>
<td>_rcpdp()</td>
<td></td>
<td>_spint()</td>
</tr>
<tr>
<td>_rsqrsp()</td>
<td>_rsqrdp()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Refer to *C Compiler User's Guide* for more information
The 'C64x has an extensive set of intrinsics to support the instructions that were added to improve packed-data processing and other capabilities.

### 'C64x Intrinsics

<table>
<thead>
<tr>
<th>Data Access</th>
<th>MPY</th>
<th>ADD/SUB</th>
<th>DOTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>_amem2()</td>
<td>_mpyli()</td>
<td>_sadd2()</td>
<td>_dotp2()</td>
</tr>
<tr>
<td>_amem4()</td>
<td>_mpyhir()</td>
<td>_saddus2()</td>
<td>_ldotp2()</td>
</tr>
<tr>
<td>_amemd8()</td>
<td>_mpylir()</td>
<td>_saddu4()</td>
<td>_dotpn2()</td>
</tr>
<tr>
<td>_amem2_const()</td>
<td>_mpysu4()</td>
<td>_add4()</td>
<td>_dotpnrsu2()</td>
</tr>
<tr>
<td>_amem4_const()</td>
<td>_mpyu4()</td>
<td>_sub4()</td>
<td>_dotpsu2()</td>
</tr>
<tr>
<td>_amemd8_const()</td>
<td>_mpyhi()</td>
<td>_subabs4()</td>
<td>_dotpsu4()</td>
</tr>
<tr>
<td>_mem2()</td>
<td>_mvd()</td>
<td></td>
<td>_dotp4()</td>
</tr>
<tr>
<td>_mem4()</td>
<td>_smpy2()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_memd8()</td>
<td>_gmpy4()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_mem2_const()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_mem4_cons()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_memd8_const()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*amem = aligned (LDDW), mem = not aligned (LDNW)*

Refer to *C Compiler User's Guide* for more information

### 'C64x Intrinsics

<table>
<thead>
<tr>
<th>Compare</th>
<th>Min/Max/Avg</th>
<th>(Un)Pack</th>
<th>Shift</th>
<th>Misc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>_cmpeq2()</td>
<td>_max2()</td>
<td>_pack2()</td>
<td>_shlmb()</td>
<td>_swap4()</td>
</tr>
<tr>
<td>_cmpeq4()</td>
<td>_max4()</td>
<td>_packh2()</td>
<td>_shrmb()</td>
<td>_bitc4()</td>
</tr>
<tr>
<td>_cmpgt2()</td>
<td>_min2()</td>
<td>_packh4()</td>
<td>_shr2()</td>
<td>_bitr()</td>
</tr>
<tr>
<td>_cmpgtu4()</td>
<td>_minu4()</td>
<td>_packh14()</td>
<td>shru2()</td>
<td>_shfl()</td>
</tr>
<tr>
<td>_xpnd2()</td>
<td>_avg2()</td>
<td>_packh12()</td>
<td>sshvl()</td>
<td>_deal()</td>
</tr>
<tr>
<td>_xpnd4()</td>
<td>_avgu4()</td>
<td>_packlh2()</td>
<td>sshvr()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>_abs2()</td>
<td>_spack2()</td>
<td>_rotl()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_spacku4()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_unpkhu4()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_unpklu4()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Refer to *C Compiler User's Guide* for more information
### 32 from 64

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned _loll (long long src);</td>
<td>Returns the low (even) register of a long long register pair</td>
</tr>
<tr>
<td>unsigned _lo (double src);</td>
<td>Returns the low (even) register of a double register pair</td>
</tr>
<tr>
<td>unsigned _hill (long long src);</td>
<td>Returns the high (odd) register of a long long register pair</td>
</tr>
<tr>
<td>unsigned _hi (double src);</td>
<td>Returns the high (odd) register of a double register pair</td>
</tr>
<tr>
<td>long long _itoll (unsigned src2, unsigned src1);</td>
<td>Builds a new long long register pair by reinterpreting two unsigned values, where src2 is the high (odd) register and src1 is the low (even) register</td>
</tr>
</tbody>
</table>

### New C64x+ Intrinsics

<table>
<thead>
<tr>
<th>.L</th>
<th>.M</th>
<th>.S</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ADDSUB ()</td>
<td>_CMPY ()</td>
<td>_DMV ()</td>
</tr>
<tr>
<td>_ADDSUB2 ()</td>
<td>_CMPYR ()</td>
<td>_RPACK2 ()</td>
</tr>
<tr>
<td>_DPACK2 ()</td>
<td>_CMPYR1 ()</td>
<td></td>
</tr>
<tr>
<td>_DPACKX2 ()</td>
<td>_DDOT4 ()</td>
<td></td>
</tr>
<tr>
<td>_SADDSUB ()</td>
<td>_DDOTP4 ()</td>
<td></td>
</tr>
<tr>
<td>_SADDSUB2 ()</td>
<td>_DDOTPH2 ()</td>
<td></td>
</tr>
<tr>
<td>_SHFL3 ()</td>
<td>_DDOTPH2R ()</td>
<td></td>
</tr>
<tr>
<td>_SUBL2 ()</td>
<td>_DDOTPL2R ()</td>
<td></td>
</tr>
<tr>
<td>_MPY ()</td>
<td>_SMPY32 ()</td>
<td></td>
</tr>
<tr>
<td>_MPY2IR ()</td>
<td>_XORMPY ()</td>
<td></td>
</tr>
</tbody>
</table>

Refer to *C Compiler User’s Guide* for more information
### New C66x Intrinsics (1)

<table>
<thead>
<tr>
<th>Creation</th>
<th>Reinterpret</th>
<th>Extraction</th>
<th>C6600</th>
</tr>
</thead>
<tbody>
<tr>
<td>_itod</td>
<td>_itof</td>
<td>_hi</td>
<td>_dcmpyr1</td>
</tr>
<tr>
<td>_ftod</td>
<td>_ftoi</td>
<td>_lo</td>
<td>_dcmpyr1</td>
</tr>
<tr>
<td>_ftof2</td>
<td>_ltod</td>
<td>_hill</td>
<td>_cmatmypi1</td>
</tr>
<tr>
<td>_itol</td>
<td>_itol128</td>
<td>_loll</td>
<td>_cmatmypi1</td>
</tr>
<tr>
<td>_itoll</td>
<td>_lltol28</td>
<td>_hif</td>
<td>_cmatmypi1</td>
</tr>
<tr>
<td>_dtoll</td>
<td>_dtol128</td>
<td>_lof</td>
<td>_cmatmypi1</td>
</tr>
<tr>
<td></td>
<td>_f2tol128</td>
<td>_hif2</td>
<td>_qsmpy32r1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_lof2</td>
<td>_cmpy32r1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_hi128</td>
<td>_ccmpy32r1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_hi2 128</td>
<td>_qmpy32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_lo128</td>
<td>_dsmpy2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_lod128</td>
<td>_dmpy2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_lof2 128</td>
<td>_dmpyu2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_get32 128</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_get32f 128</td>
<td></td>
</tr>
</tbody>
</table>

More C66x intrinsics ...

### New C66x Intrinsics (2)

<table>
<thead>
<tr>
<th>C6600</th>
<th>C6600</th>
<th>C6600</th>
<th>C6600</th>
<th>C6600</th>
</tr>
</thead>
<tbody>
<tr>
<td>_dpsmu4h</td>
<td>_dpsmu4hll</td>
<td>_dsdr2</td>
<td>_dshl2</td>
<td>_land</td>
</tr>
<tr>
<td>_dpsmu4h_dadd</td>
<td>_dpsmu4h_daddc</td>
<td>_shl12</td>
<td>_dsnu12</td>
<td>_landn</td>
</tr>
<tr>
<td>_dpsmu4h_dsadd</td>
<td>_dpsmu4h_dsadd2</td>
<td>_dexpnd4</td>
<td>_dcrot90</td>
<td>_lor</td>
</tr>
<tr>
<td>_dpsmu4h_dsub</td>
<td>_dpsmu4h_dsub2</td>
<td>_dcrot90</td>
<td>_dcrot270</td>
<td>_dcmpy</td>
</tr>
<tr>
<td>_dpsmu4h_dsubl</td>
<td>_dpsmu4h_dsub2</td>
<td>_dmvdp</td>
<td>_dmvdp</td>
<td>_dmpysu4</td>
</tr>
<tr>
<td>_dpsmu4h_dshl</td>
<td>_dpsmu4h_dshl2</td>
<td>_max2</td>
<td>_dmax2</td>
<td>_dmpyu4</td>
</tr>
<tr>
<td>_dpsmu4h_dshr</td>
<td>_dpsmu4h_dshr2</td>
<td>_dmin2</td>
<td>_dmin2</td>
<td>_dtop4h</td>
</tr>
<tr>
<td>_dpsmu4h_dshl1</td>
<td>_dpsmu4h_dshl2</td>
<td>_dmaxu4</td>
<td>_dmaxu4</td>
<td>_dtop4hll</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>_ddotpsu4h</td>
</tr>
<tr>
<td>_dpsmu4h_ddotp4</td>
<td></td>
<td></td>
<td></td>
<td>_ddotpsu4h</td>
</tr>
<tr>
<td></td>
<td>_dpsmu4h_ddotp4</td>
<td>_ddotpsu4h</td>
<td>_ddotpsu4h</td>
<td></td>
</tr>
</tbody>
</table>

Refer to *C Compiler User's Guide* for more information
More Intrinsic Examples

Intrinsic Examples

- \( c = \_\text{dotp2}(a0a1, b0b1) \);
- \( y = \_\text{max2}(a0a1, b0b1) \);
- \( \text{Im}_\text{re} = \_\text{cmpy}(a1a0, b1b0) \);
- \( z = \_\text{add2}(a0a1, b0b1) \);
- \( \text{long long } h3h2h1h0 = \_\text{amem8}(&h[4]) \);
- \( \text{int } h1h0 = \_\text{loll}(h3h2h1h0) \);
- \( \text{int } h3h2 = \_\text{hill}(h3h2h1h0) \);
- \( \text{long long } h3h2h1h0 = \_\text{itoll}(h3h2, h1h0) \);
- \( \text{int } h0h1 = \_\text{packlh2}(h1h0, h1h0) \);

Notice that we use “helpful” variable names

Here's another example using intrinsics to perform the LDDW optimized dot-product routine that we have been looking at in this workshop.

Example: Using Intrinsics with DOTP2

```c
for (i = 0; i < len; i += 4) {
    a3_a2 = _hill(_amemd8_const(&a[i]));
    a1_a0 = _loll(_amemd8_const(&a[i]));
    b3_b2 = _hill(_amemd8_const(&b[i]));
    b1_b0 = _loll(_amemd8_const(&b[i]));

    /* Perform dot-products on pairs of elements, totaling the results in the accumulator. */
    sum_high += _dotp2(a3_a2, b3_b2);
    sum_low += _dotp2(a1_a0, b1_b0);
}
```

How do we see what kind of code this produces? As in the upcoming lab, you can open the .asm file generated by the tools when you use the –k option. The following page contains a sample of the code showing the very efficient, single-cycle loop produced using the preceding code.
Intrinsics

dotp.asm

L3: ; PIPED LOOP PROLOG
    LDDW .D1T1 *A6++,A5:A4       ; [20] (P) <0,0>
    || [ A0] BDEC .S1 L4,A0
    || LDDW .D2T2 *B6++,B5:B4       ; [20] (P) <0,0>
    || LDDW .D1T1 *A6++,A5:A4       ; [20] (P) <1,0>
    || [ A0] BDEC .S1 L4,A0
    || LDDW .D2T2 *B6++,B5:B4       ; [20] (P) <1,0>
    || LDDW .D1T1 *A6++,A5:A4       ; [20] (P) <2,0>
    || [ A0] BDEC .S1 L4,A0
    || LDDW .D2T2 *B6++,B5:B4       ; [20] (P) <2,0>
    || LDDW .D1T1 *A6++,A5:A4       ; [20] (P) <3,0>
    || [ A0] BDEC .S1 L4,A0
    || LDDW .D2T2 *B6++,B5:B4       ; [20] (P) <3,0>
    || MVK .S2 0x4,B0            ; init prolog collapse
    || LDDW .D1T1 *A6++,A5:A4       ; [20] (P) <4,0>
    || [ A0] BDEC .S1 L4,A0
    || LDDW .D2T2 *B6++,B5:B4       ; [20] (P) <4,0>

;** -----------------------------------------------------------------*

L4: ; PIPED LOOP KERNEL
    [ B0] SUB .S2 B0,1,B0          ; <0,9>
    || ![B0] ADD .L2 B8,B7,B7       ; [20] <0,9>
    || ![B0] ADD .L1 A7,A3,A3       ; [20] <0,9>
    || DOTP2 .M1X B5,A5,A7          ; [20] <4,5>
    || [ A0] BDEC .S1 L4,A0
    || LDDW .D1T1 *A6++,A5:A4       ; [20] <9,0>
    || LDDW .D2T2 *B6++,B5:B4       ; [20] <9,0>

;** -----------------------------------------------------------------*

L5: ; PIPED LOOP EPILOG
    ADD .D2 B8,B7,B6              ; [20] (E) <1,9>
    || ADD .D1 A7,A3,A3            ; [20] (E) <1,9>
    || DOTP2 .M1X B5,A5,A3          ; [20] (E) <5,5>
    || DOTP2 .M2X B4,A4,B7          ; [20] (E) <5,5>
    || ADD .D2 B8,B6,B6              ; [20] (E) <2,9>
    || ADD .D1 A7,A3,A3            ; [20] (E) <2,9>
    || DOTP2 .M1X B5,A5,A3          ; [20] (E) <6,5>
    || DOTP2 .M2X B4,A4,B4          ; [20] (E) <6,5>
    || ADD .D2 B8,B6,B6              ; [20] (E) <3,9>
    || ADD .D1 A7,A3,A3            ; [20] (E) <3,9>
    || DOTP2 .M2X B4,A4,B5          ; [20] (E) <7,5>
    || DOTP2 .M1X B5,A5,A3          ; [20] (E) <7,5>
    || ADD .D2 B8,B6,B6              ; [20] (E) <4,9>
    || ADD .D1 A7,A3,A6            ; [20] (E) <4,9>
    || DOTP2 .M2X B4,A4,B4          ; [20] (E) <8,5>
    || DOTP2 .M1X B5,A5,A3          ; [20] (E) <8,5>
    || ADD .D2 B7,B6,B5              ; [20] (E) <5,9>
    || ADD .D1 A3,A6,A4            ; [20] (E) <5,9>
    || DOTP2 .M1X B5,A5,A3          ; [20] (E) <9,5>
    || DOTP2 .M2X B4,A4,B5          ; [20] (E) <9,5>
    || ADD .D2 B4,B5,B4              ; [20] (E) <6,9>
    || ADD .D1 A3,A4,A4            ; [20] (E) <6,9>
    || ADD .D1 A3,A4,A4            ; [20] (E) <7,9>
    || ADD .D2 B5,B4,B5              ; [20] (E) <7,9>
    || MVC .S2 B9,CSR      ; interrupts on
    || ADD .D2 B4,B5,B4              ; [20] (E) <8,9>
    || ADD .D1 A3,A4,A4            ; [20] (E) <8,9>
    || ADD .D1 A3,A4,A3            ; [20] (E) <9,9>
    || ADD .D2 B5,B4,B7              ; [20] (E) <9,9>
    || ADD .D1X B7,A3,A4
Provide the Compiler with More Information (Pragmas)

Whenever your compiler has more information about your system, it can do a better job of optimizing your code.

<table>
<thead>
<tr>
<th>Provide Compiler with More Insight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Program Level Optimization: <code>-pm -op2 -o3</code></td>
</tr>
<tr>
<td>2. #pragma UNROLL(# of times to unroll);</td>
</tr>
<tr>
<td>3. #pragma MUST_ITERATE(min, max, %factor);</td>
</tr>
<tr>
<td>4. Scope (static keyword, -op2)</td>
</tr>
<tr>
<td>5. #pragma DATA_ALIGN(variable, 2^n alignment);</td>
</tr>
<tr>
<td>6. _nassert()</td>
</tr>
</tbody>
</table>

- Like `-pm`, #pragmas are an easy way to pass more information to the compiler
- The compiler uses this information to create “better” code
- #pragmas are ignored by other C compilers if they are not supported

The pragma statements can be included into your C programs to do exactly what they say.

1. Program Level Optimization

Was just discussed in the last topic, but we mention it again here since it is an important part of providing more insight to the compiler
2. UNROLL

The UNROLL pragma allows you to direct the compiler to unroll the loop and improve your code performance.

```c
#pragma UNROLL(2);
for(i = 0; i < count ; i++) {
    sum += a[i] * x[i];
}
```

- Tells the compiler to unroll the for() loop twice
- The compiler will generate extra code to handle the case that count is odd
- The #pragma must come right before the for() loop
- UNROLL(1) tells the compiler not to unroll a loop

3. MUST_ITERATE

MUST_ITERATE allows you to give the compiler information about the trip count that will always be used for a loop. We will look at MUST_ITERATE closer in the next chapter.

```c
#pragma UNROLL(2);
#pragma MUST_ITERATE(10, 100, 2);
for(i = 0; i < count ; i++) {
    sum += a[i] * x[i];
}
```

- Gives the compiler information about the trip (loop) count
  In the code above, we are promising that:
  count >= 10, count <= 100, and count % 2 == 0
- If you break your promise, you might break your code
- Allows the compiler to remove unnecessary code
- Modulus (%) factor allows for efficient loop unrolling
- The #pragma must come right before the for() loop
4. Static Keyword

(4. Static
(FIR Example))

```c
static int my_cfir(short *x, short *h, short *r, int nh, int nr)
{
    int i, j, sum;
    for (j = 0; j < nr; j++) {          Outer Loop
        sum = 0;
        for (i = 0; i < nh; i++) Inner Loop
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
    }
    return(1);
}
```

What is the theoretical maximum optimization of the FIR filter?

- **How many MAC's can be done per cycle?** (C64x) 4 MACs/cycle
- **How many loads/cycle?** (C64x) 1 LDNDWs/cycle = 4 shorts/cycle
- **Why do we need to use LDNDWs for our filter?** Loading only 4 shorts only means we can only support 2 MACs/cycle, thus we are I/O bound.
  
  Each filter pass uses a “sliding” window:
  
  Cycle 1 = x0, x1, x2, x3;
  Cycle 2 = x1, x2, x3, x4; etc.

  Thus, the algorithm requires data from unaligned boundaries.

How can we improve our rate to get back close to 4 MACs/cycle?

If I had to do it by hand, I’d unroll the inner/outer loops so that the “unaligned” data could be reused once it had been read into registers.

4. Scope

(Static
Keyword)

```c
static int my_cfir(short *x, short *h, short *r, int nh, int nr)
{
    int i, j, sum;
    for (j = 0; j < nr; j++) { Outer Loop
        sum = 0;
        for (i = 0; i < nh; i++) Inner Loop
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
    }
    return(1);
}
```

- The **static keyword indicates the function is only used within the file** (local), as opposed to being a global function
- When using –pm –op2, the compiler is able to determine this automatically
- Knowing entire scope of the function, the compiler can be more aggressive; in this case, it can now “unroll & jam” combining the inner/outer loops
- Here are some example results:

<table>
<thead>
<tr>
<th></th>
<th># cycles/loop</th>
<th>Results/loop</th>
<th>Results/Cycle</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>–o3 with</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>only –o3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>62471</td>
</tr>
<tr>
<td>–pm –op0</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>62471</td>
</tr>
<tr>
<td>–pm –op2</td>
<td>9</td>
<td>32</td>
<td>3.6</td>
<td>28035</td>
</tr>
<tr>
<td>static</td>
<td>9</td>
<td>32</td>
<td>3.6</td>
<td>28035</td>
</tr>
</tbody>
</table>
Both the -pm option and the Static keyword allowed the compiler to combine our inner/outer loops. How can this help?

**Outer Loop Unrolling**

5. **DATA_ALIGN**

DATA_ALIGN allows you to tell the compiler how to create variables. As we have seen earlier in this course, if variables are properly aligned, they can be accessed with packed-data instructions.

### 5. Data Align Pragma

```
#pragma DATA_ALIGN(a, 8);
short a[256] = {1, 2, 3, ...}
#pragma DATA_ALIGN(x, 8);
short x[256] = {256, 255, 254, ...}
#pragma UNROLL(2);
#pragma MUST_ITERATE(10, 100, 2);
for(i = 0; i < count ; i++) {
    sum += a[i] * x[i];
}
```

- Tell compiler to create variables on a $2^n$ boundary
- Allows use of (double) word-wide optimized loads/stores
6. Use _nassert()

6. _nassert()

```c
_nassert((ptr & 0x7) == 0);
```

- Generates no code, evaluated at compile time
- Hints to the optimizer as to what optimizations might be valid
- Tells the optimizer that the expression declared with the ‘assert’ function is true
- Above example declares that `ptr` is aligned on an 8-byte boundary (i.e. the lowest 3-bits of the address in `ptr` are 000b)

Why would we need this?

Like the pragmas, the _nassert() statement is used to pass information to the compiler. The above statement asserts to the compiler that when “label” is anded with 0x3 the result will be zero. In other words, this is how you can tell the compiler that the lower two address bits are zero in the address represented by “label”. Of course, before doing this, you should use the #pragma DATA_ALIGN to assure the definition of label is on a 4-byte boundary. Something along the lines of:

```c
#pragma DATA_ALIGN(myVar, 4)
short myVar();
...
_nassert(myVar & 0x3 == 0);
for (i = 40; i > 0; i++) {
    myVar(i) * a(i); ...
```

When using –pm, the compiler is effective at determining data alignments. That is, if it can see both the definition (`short myVar` with align pragma) and the usage (preceding the `for` loop), the compiler can usually figure out when to use things like word-wide optimization.

At times, though, the compiler is not omniscient enough to use these aggressive optimizations. For example, what if you were creating a library of DSP routines, that were to be linked as object code into other projects. In this case, the source code being optimized doesn’t even contain the data definitions – the data declaration will be done at a later time, in another project. This is a good time to use _nassert. If you can guarantee to the compiler that the data will be aligned at runtime (maybe, by using the #pragma DATA_ALIGN when declaring the variable to be passed to the library routine), then the compiler can go ahead and use the more aggressive optimizations.
**Using _nassert() for Object Libraries**

### Using Object Code Libraries

- **Why use an object code library (.lib)?**
  
  There are many reasons, but having a set of validated functions whose implementation doesn't change can be valuable

- **What drawback exists with object code libraries?**
  
  - Program level optimization (-pm) only works for source code
  - Library code, compiled independently, cannot benefit from many optimizations

### Using _nassert() within Library

- _nassert example tells the compiler that symbols (aptr, xptr) are aligned
- This assertion allows the optimizer to perform word-wide optimization, thus achieving the best performance

The Data Align #pragma works great for making sure that the variables are created on aligned boundaries. However, if the variables are going to be accessed by a pointer in another file, the compiler doesn't have visibility into the file that created them. So, it can't see that the variables are aligned, because the pointer could take on any value. The _nassert intrinsic helps us fix this issue.
Reduce user error with tuned object libraries

Sometimes libraries are built with two different versions of the same function. One named slightly different from the other, depending upon whether the data must be aligned or not.

Reduce User Error

When using #pragmas or _nassert() to improve performance...
offer both restrained and unrestrained versions of each function

```c
File.c

main()
{
    LIB_function();
    LIB_function_aligned();
}

MyLibrary.lib

int LIB_function()
int LIB_function_aligned()
int LIB_function_alignedGT10()
```
Use Optimized Libraries

TI provides several libraries of optimized code to help build a DSP system. The following slides provide some information on some of these libraries and how to use them.

**DSPLIB**

- Optimized **DSP Function Library** for C programmers
- These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical.
- Shorten your dev't time by using these routines to achieve execution speeds considerably faster than equivalent ANSI C code.
- Versions Available:
  - Float: C674x, C67x/C67x+
  - Fixed: C64x, C64x+/C674x
- The DSP library features:
  - C-callable
  - Hand-coded assembly-optimized
  - Tested against C model and existing run-time-support functions

<table>
<thead>
<tr>
<th>Adaptive filtering</th>
<th>Math</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP_firlms2</td>
<td>DSP_dotp_sq</td>
</tr>
<tr>
<td>Correlation</td>
<td>DSP_dotprod</td>
</tr>
<tr>
<td>DSP_autocor</td>
<td>DSP_maxval</td>
</tr>
<tr>
<td>FFT</td>
<td>DSP_maxidx</td>
</tr>
<tr>
<td>DSP_bitrev_cplx</td>
<td>DSP_minval</td>
</tr>
<tr>
<td>DSP_radix 2</td>
<td>DSP_mul32</td>
</tr>
<tr>
<td>DSP_rfft</td>
<td>DSP_neg32</td>
</tr>
<tr>
<td>DSP_fft</td>
<td>DSP_recip16</td>
</tr>
<tr>
<td>DSP_fft16x16r</td>
<td>DSP_vecsumsq</td>
</tr>
<tr>
<td>DSP_fft16x16t</td>
<td>DSP_w_vec</td>
</tr>
<tr>
<td>DSP_fft16x32</td>
<td>Matrix</td>
</tr>
<tr>
<td>DSP_fft32x32</td>
<td>DSP_mat_mul</td>
</tr>
<tr>
<td>DSP_fft32x32s</td>
<td>DSP_mat_trans</td>
</tr>
<tr>
<td>DSP_ifft16x32</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>DSP_ifft32x32</td>
<td>DSP_bexp</td>
</tr>
<tr>
<td>Filters &amp; convolution</td>
<td>DSP_blk_eswap16</td>
</tr>
<tr>
<td>DSP_fir_cplx</td>
<td>DSP_blk_eswap32</td>
</tr>
<tr>
<td>DSP_fir_gen</td>
<td>DSP_blk_eswap64</td>
</tr>
<tr>
<td>DSP_fir_r4</td>
<td>DSP_blk_move</td>
</tr>
<tr>
<td>DSP_fir_r8</td>
<td>DSP_floq15</td>
</tr>
<tr>
<td>DSP_fir_sym</td>
<td>DSP_minerror</td>
</tr>
<tr>
<td>DSP_iir</td>
<td>DSP_q15tof1</td>
</tr>
</tbody>
</table>

**IMGLIB**

- Optimized **Image Function Library** for C programmers using C62x/C67x and C64x devices
- The image library features:
  - C-callable
  - C and linear assembly src code
  - Tested against C model

<table>
<thead>
<tr>
<th>Compression / Decompression</th>
<th>Picture Filtering / Format Conversions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMG_d dct 8x8</td>
<td>IMG_conv 3x3</td>
</tr>
<tr>
<td>IMG_idct 8x8</td>
<td>IMG_corr 3x3</td>
</tr>
<tr>
<td>IMG_idct 8x8 12q4</td>
<td>IMG_corr_gen</td>
</tr>
<tr>
<td>IMG_mad 8x8</td>
<td>IMG_errdif bin</td>
</tr>
<tr>
<td>IMG_mad 16x16</td>
<td>IMG_median 3x3</td>
</tr>
<tr>
<td>IMG_mpeg2_vid_intra</td>
<td>IMG_pix_expand</td>
</tr>
<tr>
<td>IMG_mpeg2_vid_inter</td>
<td>IMG_pix_sat</td>
</tr>
<tr>
<td>IMG_quantize</td>
<td>IMG_ycc_demux_be16</td>
</tr>
<tr>
<td>IMG_ycc_demux_le16</td>
<td>IMG_ycc_demux_le16</td>
</tr>
<tr>
<td>IMG_ycc_cr422_rgb565</td>
<td>IMG_w ave horz</td>
</tr>
<tr>
<td>IMG_wave horz</td>
<td>IMG_boundary</td>
</tr>
<tr>
<td>IMG_wave vert</td>
<td>IMG此刻_3b</td>
</tr>
<tr>
<td>IMG_erode_bin</td>
<td>IMG_t rle2max</td>
</tr>
<tr>
<td>IMG_histogram</td>
<td>IMG_t rle2hr</td>
</tr>
<tr>
<td>IMG_perimeter</td>
<td>IMG_t rle2hr</td>
</tr>
<tr>
<td>IMG_sobel</td>
<td>IMG_t rle2hr</td>
</tr>
<tr>
<td>IMG_t rle2min</td>
<td>IMG_t rle2hr</td>
</tr>
<tr>
<td>IMG_t rle2thr</td>
<td>IMG_t rle2hr</td>
</tr>
</tbody>
</table>
**FastRTS (C62x/C64x)**

- Optimized **floating-point math** function library for C programmers enhances floating-point performance on C62x and C64x fixed-point devices
- The FastRTS library features:
  - C-callable
  - Hand-coded assembly-optimized
  - Tested against C model and existing run-time-support functions
- Download the library – search for: SPRC122
- FastRTS must be installed per directions in its Users Guide (SPRU653.PDF)

<table>
<thead>
<tr>
<th></th>
<th>Single Precision</th>
<th>Double Precision</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>addf</td>
<td>_addf</td>
<td>_addd</td>
<td>_cvtdf</td>
</tr>
<tr>
<td>divf</td>
<td>_divf</td>
<td>_divd</td>
<td>_cvtd</td>
</tr>
<tr>
<td>fixf</td>
<td>_fixf</td>
<td>_fixd</td>
<td></td>
</tr>
<tr>
<td>fixfli</td>
<td>_fixfli</td>
<td>_fixdli</td>
<td></td>
</tr>
<tr>
<td>fixfu</td>
<td>_fixfu</td>
<td>_fixdu</td>
<td></td>
</tr>
<tr>
<td>fixful</td>
<td>_fixful</td>
<td>_fix dul</td>
<td></td>
</tr>
<tr>
<td>fltf</td>
<td>_fltf</td>
<td>_fltid</td>
<td></td>
</tr>
<tr>
<td>fttif</td>
<td>_fttif</td>
<td>_fttud</td>
<td></td>
</tr>
<tr>
<td>fttuf</td>
<td>_fttuf</td>
<td>_fttuid</td>
<td></td>
</tr>
<tr>
<td>mpyf</td>
<td>_mpyf</td>
<td>mpyd</td>
<td></td>
</tr>
<tr>
<td>recipf</td>
<td>recip</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subf</td>
<td>_subf</td>
<td>_subd</td>
<td></td>
</tr>
</tbody>
</table>

**FastMath (C67x)**

- Optimized **floating-point math** function library for C programmers using TMS320C67x devices
- Includes all floating-point math routines currently in existing C6000 run-time-support libraries
- The FastRTS library features:
  - C-callable
  - Hand-coded assembly-optimized
  - Tested against C model and existing run-time-support functions
- Download the library – search for: SPRC060
- FastRTS must be installed per directions in its Users Guide (SPRU100a.PDF)

<table>
<thead>
<tr>
<th></th>
<th>Single Precision</th>
<th>Double Precision</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>atanf</td>
<td>atanf</td>
<td>atan</td>
<td></td>
</tr>
<tr>
<td>atan2f</td>
<td>atan2f</td>
<td>atan2</td>
<td></td>
</tr>
<tr>
<td>cosf</td>
<td>cosf</td>
<td>cos</td>
<td></td>
</tr>
<tr>
<td>expf</td>
<td>expf</td>
<td>exp</td>
<td></td>
</tr>
<tr>
<td>exp2f</td>
<td>exp2f</td>
<td>exp2</td>
<td></td>
</tr>
<tr>
<td>exp10f</td>
<td>exp10f</td>
<td>exp10</td>
<td></td>
</tr>
<tr>
<td>logf</td>
<td>logf</td>
<td>log</td>
<td></td>
</tr>
<tr>
<td>log2f</td>
<td>log2f</td>
<td>log2</td>
<td></td>
</tr>
<tr>
<td>log10f</td>
<td>log10f</td>
<td>log10</td>
<td></td>
</tr>
<tr>
<td>powf</td>
<td>powf</td>
<td>pow</td>
<td></td>
</tr>
<tr>
<td>recipf</td>
<td>recipf</td>
<td>recip</td>
<td></td>
</tr>
<tr>
<td>rsqrtf</td>
<td>rsqrtf</td>
<td>rsqrt</td>
<td></td>
</tr>
<tr>
<td>sinf</td>
<td>sinf</td>
<td>sin</td>
<td></td>
</tr>
</tbody>
</table>
Use Optimized Libraries

**VLIB**

- Optimized functions for **C64x & C64x+** DaVinci DSP (examples: DM642, DM6437)
- **Simulink** blocks to enable MathWorks model-based design
- Bit-exact version for testing on PC
- **VLIB 2.0 Function List**
  - Background subtraction
    - Exponentially and uniformly weighted mean
    - Exponentially and uniformly weighted variance
    - Mixture of Gaussians
  - Canny edge detection
    - Non-maxima suppression
  - Hough transform for lines
  - Integral image
  - Image pyramid
  - Legendre moments
  - 6 additional fcn's, incl: Bit mask packing/unpacking, 16-bit IIR filter, L1 distance

Now that you know about the libraries, here's where to find them and some information on how they are organized. Each library also has documentation that goes along with it.

**Location of Libraries**

*(in CCS v3.1)*

- DSP and IMG Libraries provided as source archive, and Little Endian C6000 obj library
- **Folder Structure:**
  - *bin* - supporting Windows executables
  - *lib* - library files (.lib) and source code (.src)
  - *include* - contains the library header files
  - *support* - miscellaneous supporting code
- **CCS Doc's folder contains:**
  - SPRU565A.pdf - DSP API User Guide
  - SPRU023A.pdf - Imaging API User Guide
  - SPRU100A.pdf – FastRTS Math API UG
- **Application Notes:**
  - SPRA885.pdf - DSPLIB App note
  - SPRA886.pdf - IMGLIB App note
**Summary – Coding Methodologies**

**Summary - Optimization Methodology**

- **There are 6 coding methods that can be used:**
  1. **Natural C code:** No effort.
  2. **Optimized C code:** Smallest effort. Relies totally on compiler to perform automatic code transformations.
  3. **Intrinsic C code:** More effort. However very flexible method to map out all the instructions and the algorithmic transformations that are needed for optimizing the algorithm. Many benefits from Linear Assembly, yet easier to use.
  4. **Linear Assembly:** Allows specific choice of assembly instructions, thus eliminating the abstraction between C and Assembly languages.
  5. **Partition Linear Assembly:** Allow further control by mapping register symbols to sides to remove scheduling issues. Un-partitioned Linear Assembly is more portable across compiler revisions.
  6. **Standard Hand-Coded Assembly:** Greatest effort. Should not be used. (Only required for interrupt vector table, but that can be built using the Config Tool.)

- **Quit with the first method that meets real time.**
- **Always save the code at each method level (esp. #1 and #4).**
  - These are most portable across compiler revisions.
  - Future compiler revisions often can improve upon “less instrumented” code.

**Six different flavors of the same function being optimized.**

1. **Natural C code or Committee code.** Text Book implementation of the algorithm to be optimized. Used to compare other flavors for speedup and for bit-exactness. This can also be viewed as the golden C code.

2. **Optimized C code** can use manual loop unrolling of inner and outer loops. It can also use compiler pragmas and _nasserts to the compiler to inform it about the alignment of various input and output arrays. This allows the compiler to perform automatic SIMD transformations, which works in some cases but not all.

3. **Intrinsic C code** allows for the use of all the instructions on the given architecture and can be used to express any assembly language code in a high level environment. The only limitation is circular addressing support. The compiler may not be able to perform memory alias disambiguation or partition instructions correctly between the two data paths of the architecture.

4. **Linear assembly** is a mapping of the intrinsic C code into assembly by directly using the instructions in an assembly language format. Assembly optimizer is invoked to act on these instructions and perform register allocation and scheduling.

5. **Partitioned linear assembly** performs partitioning of the instructions by appending a .1 or .2 in front of the unit. Load/Store operations can be partitioned as .DxTx to indicate which side the pointer comes from and to which side the loaded value lands. The use of .1x shows that the second operand comes from the opposite data path.
Both partitioned and linear assembly do not have any latencies that the programmer must take care of. The assembly optimizer figures out latencies and dependencies and then performs instruction scheduling.

6. Hand-coded assembly: Of course, there is always hand coded assembly where the user does instruction set selection, register allocation and the latencies of the instructions.

Example:

**MAD_8x8 Performance Summary for (64x32)**

- This is a performance tabulation example for the MAD algorithm discussed in Chapter 7.

<table>
<thead>
<tr>
<th></th>
<th>Natural C (CN)</th>
<th>Optimized C (CO)</th>
<th>C with Intrinsics</th>
<th>Linear Assembly</th>
<th>Partitioned Serial Assembly</th>
<th>Standard Hand Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycles</td>
<td>22254</td>
<td>22254</td>
<td>22254</td>
<td>20561</td>
<td>16454</td>
<td>16458</td>
</tr>
<tr>
<td>bytes</td>
<td>1280</td>
<td>1280</td>
<td>1248</td>
<td>1056</td>
<td>1280</td>
<td>800</td>
</tr>
</tbody>
</table>

* For CCS 2.x, results may be improved upon with later versions of the code generation tools

7.96 MADs/cycle
This page intentionally left blank.
Introduction

In Chapter 9, we discussed how to optimize your C code for maximum code performance, where performance is measured by execution time. Execution time is often the most important care about for a lot of developers. However, embedded system developers may also be concerned with memory footprint, or size of the code that they put into their systems. TI’s code generation tools give you some good options to find the sweet spot for code size and performance in your system.

Outline

◆ Volatile
◆ Optimizing for Code Size
◆ C64x+ Code Size Features
Chapter Topics

Volatile...................................................................................................................................................10-3
Volatile in Linear Assembly......................................................................................................................10-4
Optimizing for Code Size .........................................................................................................................10-5
File- and Function-Level Optimizations ................................................................................................10-6
C64x+ Code Size Features ........................................................................................................................10-8
(Optional) Compact Instructions Details ................................................................................................10-9
When highly optimizing compilers are used, it is not unusual for a program to fail to execute correctly. The optimizer analyzes data flow to avoid memory accesses whenever possible.

If C code reads memory locations that are modified outside the scope of C (such as a hardware register), the compiler may optimize these reads out. To prevent this, these memory accesses must be identified with the “volatile” keyword. The compiler does not optimize out any references to volatile variables.

For example, the `while` loop waits (polls) for a location to be read as non-zero (i.e. it waits while the location is equal to zero):

```c
int *ctrl;
while (*ctrl == 0);
```

In this example, `*ctrl` is a loop-invariant expression; i.e., the expression never changes during execution of the loop. In this case, the optimizer will reduce the loop to a single memory read — not what was intended by this “busy-waiting loop.” This kind of code is common in control-type applications. To prevent this busy-waiting loop from essentially being optimized out of existence, use the “volatile” keyword. Although it is an ANSI Standard C keyword, it is often only understood by programmers which have spent time with highly optimizing C compilers.

To prevent the optimizer from changing this expression, the declaration for `*ctrl` is changed to:

```c
volatile int *ctrl;
while (*ctrl == 0);
```
Volatile

Volatile and the Cache

The volatile keyword in C does have any affect on the cacheability of off-chip memory – or vice versa. If you are using the volatile keyword to tell the compiler not to optimize away references to external memory locations that could change outside the scope of C (such as hardware switches, FPGA status registers, etc), you most like will want to configure the Memory Attribute Registers (MAR) to prevent caching of ‘old’ data. Please see the cache chapter for more information regarding the MAR registers and how to use them.

Volatile in Linear Assembly

```assembly
.reg X, Y
.volatile st, ld
STW W, *X{st} ; volatile store
STW U, *V
LDW *Y{ld}, Z ; volatile load
```

- Use .volatile to designate memory references as volatile.
- Volatile loads and stores are not deleted or reordered with respect to other volatile loads and stores.
- Example: designate memory references as volatile
Optimizing for Code Size

If code size is more critical than performance, how do you let the compiler know that?

Minimizing Space Option (-ms)

- The table shows the basic strategy employed by compiler and Asm-Opt when using the -ms options:

<table>
<thead>
<tr>
<th>-ms level</th>
<th>Performance</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>-ms0</td>
<td>90</td>
<td>10</td>
</tr>
<tr>
<td>-ms1</td>
<td>60</td>
<td>40</td>
</tr>
<tr>
<td>-ms2</td>
<td>20</td>
<td>80</td>
</tr>
<tr>
<td>-ms3</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

- Use must use the optimizer (-o) with -ms for the greatest effect. The optimizer provides a great deal of information for code-size reduction, as well as increasing performance.

Additional Code Space Options

- Use program level optimization (-pm)
- Try -mh to reduce prolog/epilog code
- Use –oi0 to disable auto-inlining

- Inlining inserts a copy of a function into a C file rather than calling (i.e. branching) to it
- Auto-inlining is a compiler feature whereas small functions are automatically inlined
- Auto-inlining is enabled for small functions by –o3
- The –oisize sets the size of functions to be automatically inlined
  - size = function size * # of times inlined
  - Use –on1 or –on2 to report size
- Force function inlining with inline keyword
  - inline void func(void);
File- and Function-Level Optimizations

File Level Options

File and Function-Level Optimizations
The `FUNCTION_OPTIONS` pragma allows you to compile a specific function in a C or C++ file with additional command-line compiler options. The affected function will be compiled as if the specified list of options appeared on the command line after all other compiler options.

In C, the pragma is applied to the function specified. The syntax of the pragma in C is:

```c
#pragma FUNCTION_OPTIONS (func, "additional options");
```

In C++, the pragma is applied to the next function. The syntax of the pragma in C++ is:

```c
#pragma FUNCTION_OPTIONS("additional options");
```
C64x+ Code Size Features

- **SPLOOP: Software Pipelined Loop Buffer**
  - Loop buffer “builds” software pipelined loop
  - Only one iteration needed in source file

- **CALLP (Protected Call)**
  - Takes the place of 3-4 instructions
  - Not used unless –ms is selected since its “protection” causes a pipeline flush

- **MPY32 (32x32 multiply)**
  - Eliminates the need to run software routine from RunTime Support library

- **Compact Instructions**
  - 16-bit instruction versions of common instructions to reduce code size

**Best of all, compiler does all the work!**
### (Optional) Compact Instructions Details

#### 16-bit Compact Instructions

- **16-bit versions of common instructions added**
  - Subset of main ISA is usable as 16-bit opcode
  - Can freely mix 32- and 16-bit
  - Compiler supported, will trade off performance vs. code size

- **Significant code size reductions for “control” code**
  - **Target:** up to 30% reduction at high performance
  - Reduction in cache miss rate (due to code size reduction)

<table>
<thead>
<tr>
<th>Scenario A</th>
<th>Scenario B</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Compact ISA</td>
<td>With Compact ISA</td>
</tr>
<tr>
<td>1MB DSP Memory</td>
<td></td>
</tr>
<tr>
<td>500K Program</td>
<td>350K Program</td>
</tr>
<tr>
<td>500K Data</td>
<td>“New” Data Memory</td>
</tr>
<tr>
<td></td>
<td>650K Data</td>
</tr>
</tbody>
</table>

#### 16-bit and 32-bit Instructions

**32-bit Instruction**

<table>
<thead>
<tr>
<th>31</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>18</th>
<th>17</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>5</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>creg</td>
<td>z</td>
<td>dst</td>
<td>src2</td>
<td>src1</td>
<td>x</td>
<td>opcode</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>s</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**16-bit Instruction**

<table>
<thead>
<tr>
<th>15</th>
<th>13</th>
<th>11</th>
<th>9</th>
<th>7</th>
<th>6</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1</td>
<td>x</td>
<td>op</td>
<td>0</td>
<td>src2</td>
<td>dst</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Previous to the C64x+ all C6000 instructions were “32-bit” instructions**
- **C64x+ introduces “16-bit” versions of common instructions (these are also called “compact” instructions)**
- **Compact instructions save code space, but...**
  - Reduced register set (upper or lower)
  - Some are two operand (rather than three)
  - No predicate register (unconditional, except for BNOP)
  - Loads can be “protected” (eliminates NOP 4)
### Compact Packet Structures

#### Standard C6000 Fetch Packet

<table>
<thead>
<tr>
<th>Word</th>
<th>32-bit opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32-bit opcode</td>
</tr>
<tr>
<td>1</td>
<td>32-bit opcode</td>
</tr>
<tr>
<td>2</td>
<td>32-bit opcode</td>
</tr>
<tr>
<td>3</td>
<td>32-bit opcode</td>
</tr>
<tr>
<td>4</td>
<td>32-bit opcode</td>
</tr>
<tr>
<td>5</td>
<td>32-bit opcode</td>
</tr>
<tr>
<td>6</td>
<td>32-bit opcode</td>
</tr>
<tr>
<td>7</td>
<td>32-bit opcode</td>
</tr>
</tbody>
</table>

#### Header Based Fetch Packet

<table>
<thead>
<tr>
<th>Word</th>
<th>32-bit opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32-bit opcode</td>
</tr>
<tr>
<td>1</td>
<td>16-bit Opcode</td>
</tr>
<tr>
<td>2</td>
<td>32-bit opcode</td>
</tr>
<tr>
<td>3</td>
<td>16-bit Opcode</td>
</tr>
<tr>
<td>4</td>
<td>32-bit opcode</td>
</tr>
<tr>
<td>5</td>
<td>16-bit Opcode</td>
</tr>
<tr>
<td>6</td>
<td>16-bit Opcode</td>
</tr>
<tr>
<td>7</td>
<td>Header</td>
</tr>
</tbody>
</table>

#### Compact Header Word Format

- **Layout**: Which words are split into two 16-bit instructions
- **p-bits**: Which 16-bit instructions are executed as parallel instructions
- **Expansion**: Provides other information about instructions in the packet (branches, data size, saturation, protected load, and register set)

#### Example Code Size Reduction

- **48 bytes, 9 cycles on 64x**

#### Optimization

- **32 bytes, 9 cycles on 64x+**
- **50% Savings**
### Look of Compact Object Code

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Dissassembled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>e250</td>
<td>ADD.L1 A7, A4, A5</td>
</tr>
<tr>
<td>00000002</td>
<td>e251</td>
<td>ADD.L2 B7, B4, B5</td>
</tr>
<tr>
<td>00000004</td>
<td>04a3c1e1</td>
<td>ADD.S1 A30, A8, A9</td>
</tr>
<tr>
<td>00000008</td>
<td>a46b</td>
<td>BNOP.S2 label1, 5</td>
</tr>
<tr>
<td>0000000a</td>
<td>607c</td>
<td>LDW.D1T1 *A4[3], A7</td>
</tr>
<tr>
<td>0000000c</td>
<td>0a8c02c7</td>
<td>LDH.D2T2 *+B3[0], B21</td>
</tr>
<tr>
<td>0000010</td>
<td>06b38c81</td>
<td>MPY.M1 A28, A12, A13</td>
</tr>
<tr>
<td>0000014</td>
<td>06b38c82</td>
<td>MPY.M2 B28, B12, B13</td>
</tr>
<tr>
<td>0000018</td>
<td>02920ca1</td>
<td>SHL.S1 A4, 0x10, A5</td>
</tr>
<tr>
<td>000001c</td>
<td>e0a08033</td>
<td>.fphead n, l, W,</td>
</tr>
</tbody>
</table>

- Notice how 16-bit instructions are displayed in the Dissassembly Window inside CCS

### ‘C64x+ Compact Code Generation

- Compact 16-bit instructions cannot be specified in assembly language
- Instructions can be “tailored”, though, so that they will likely use 16-bit form
- Depending on –ms option, compiler adjusts:
  - Instruction selection
  - Instruction-dependent register allocation
  - Scales-back/enables other optimizations
- Disable with --no_compress

---

**Compressible**

- All instructions are 32-bit
- "Compressible" instructions have been converted to 16-bits
- What affect does this have?
Code-Savings example

Example – C64x vs. C64x+

Notes:
- Avg. of 29 applications
- Flat memory
Basic Memory Management

Introduction

Memory management involves:
- Defining system memory requirements
- Describing the available memory map to the linker
- Allocating code and data sections using the linker

These along with the C6000 memory architecture are covered in this chapter. The lab explores basic linking within CCS.

Defining memory requirements is very application specific and therefore, is outside the scope of this workshop. If you have question regarding this, please discuss these during a break with your instructor.

Outline

- C6000 Memory Architecture
  - What is a Memory Map?
  - Addressable Memory
  - C64x+ Memory Maps
  - C674x Devices
  - Older Generations (C671x, C672x, C64x)
- Section ? Memory Placement
- Using the Linker (.CMD)
- Lab Exercise
Chapter 11 Topics

Basic Memory Management........................................................................................................ 11-1

C6000 Memory Architecture........................................................................................................ 11-3
Adresseable Memory................................................................................................................... 11-4
C64x+ Devices .......................................................................................................................... 11-6
Older Generations (C671x, C672x, C64x) ............................................................................. 11-8

Section → Memory Placement.................................................................................................. 11-10
What is a Section? ..................................................................................................................... 11-10
Section Placement Exercise ...................................................................................................... 11-11
Compiler’s Section Names - Review ........................................................................................ 11-13
System Software Initialization Sections .................................................................................... 11-14

Using the Linker....................................................................................................................... 11-16
Overview of Linking ................................................................................................................ 11-16
How Do You Place Sections into Memory Regions? ................................................................. 11-17
  1. Creating a New Memory Objects (Using MEM) ................................................................. 11-18
  2. Placing Sections – MEM Manager Properties ................................................................ 11-19
  3. Running the Linker .......................................................................................................... 11-21
Sidebar: Linker Command File (CMD).................................................................................... 11-24
C6000 Memory Architecture

The C6000 CPU supports up to 4 Giga-Bytes (4GB) of external memory. This is accomplished using 32-bit wide addresses. While the CPU supports full 32-bit addressing, due to cost constraints, not all devices route the full 32-bits of address externally. Current devices support 52MB up to 1GB.

Another feature is that the C6000 supports byte-wide addressing. This means each address represents one byte of data. This is juxtaposed against many DSPs that only support word-wide addressing (16- or 32-bits per address). Byte wide addressing provides better memory density which is important in applications such as video and imaging (better usage, less waste).

Note: The early C6000 devices (C620x, C6701) used a slightly different memory architecture the rest of the family. This memory architecture is described in the optional topics at the end of this chapter. As for all C6000 devices, it is a good idea refer to each device's specific data sheet for exact memory configuration details.

Block Diagrams and Memory Maps

The memory map allows the user to see the logical layout of memory in a contiguous fashion. This helps the programmer plan where different parts of program and data will reside on the C6000.
Adressable Memory

In this chapter, we are mainly interested in addressable memory. That is, memory that has addresses that we can assign code and data to.

Many C6x devices consist of a single internal memory block (called L2), and four blocks of external memory. The four external blocks are called CE0 thru CE3 based upon the pins used to select each memory block.

If you are already familiar with the C6x memory scheme, you may notice that the memory blocks called L1P and L1D are missing. That is because on most devices (all devices except for those based on the C64x+ CPU) these are cache only, and thus provide no addresses to allocate code and data to. These memory blocks will be discussed in Chapter 15, along with a lot more detail on the internal memory and cache.
Sidebar: Chip Enable Lines Define External Memory Regions

We’re often asked: "What does it mean there are different regions (or blocks) of memory?"

The memory is broken out into blocks based upon the hardware pin signals. To make the system efficient, the address bus is reused when accessing each region; this reduces the number of pins required to go off-chip (lowers cost and power dissipation).

When an address is placed on the address bus by the EMIF, it must signal which block it is requesting data from. It does this by toggling a pin associated with the memory region it’s interested in. Each region has its own pin. CE0 is used for region 0; CE1 for region 1, and so forth. CE stands for Chip Enable. (Some other devices use the pin name CS for Chip Select, but the functionality is just the same.)
C64x+ Devices

C6437 Memory Map

<table>
<thead>
<tr>
<th>Device</th>
<th>Internal</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6437</td>
<td>L1P: 32 KB</td>
<td>DDR2: 256MB (x32b)</td>
</tr>
<tr>
<td></td>
<td>L1D: 80 KB</td>
<td>Async: 64MB (x8b)</td>
</tr>
<tr>
<td></td>
<td>L2: 128 KB</td>
<td></td>
</tr>
</tbody>
</table>

Device Specific Notes
- L1D size increased to help achieve real-time on high complexity video codecs
- L2 size kept smaller to minimize costs

C6455 Memory Map

<table>
<thead>
<tr>
<th>Device</th>
<th>Internal</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6455</td>
<td>L1P: 32 KB</td>
<td>DDR2: 512MB (x32b)</td>
</tr>
<tr>
<td></td>
<td>L1D: 32 KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L2: 2 MB</td>
<td>Async: 32MB (x64b)</td>
</tr>
</tbody>
</table>

Device Specific Notes
- L2 size increased to keep large sets of data on-chip
C6748 Device

C6748 Memory Map

<table>
<thead>
<tr>
<th>Device</th>
<th>Internal</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6748</td>
<td>L1P: 32 KB</td>
<td>DDR2: 512MB (x16b)</td>
</tr>
<tr>
<td></td>
<td>L1D: 32 KB</td>
<td>Async: 128MB (x16b)</td>
</tr>
<tr>
<td></td>
<td>L2: 256 KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L3: 128 KB</td>
<td></td>
</tr>
</tbody>
</table>

Device Specific Notes
◆ Level 3 (L3)
  • Datasheet calls it “128 KB RAM Memory”
  • Unified (i.e. Program or Data)
  • Faster than DDR2, slower than L2 RAM
Older Generations (C671x, C672x, C64x)

Details of C671x Addressable Memory

The C671x devices use the addressable memory block architecture shown previously (page 11-4). Some details include:

<table>
<thead>
<tr>
<th>Devices</th>
<th>L2</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6211 C6711</td>
<td>64 KB</td>
<td>512MB (32-bit wide)</td>
</tr>
<tr>
<td>C6712</td>
<td>64 KB</td>
<td>256MB (16-bit wide)</td>
</tr>
<tr>
<td>C6713</td>
<td>256 KB</td>
<td>512MB (32-bit wide)</td>
</tr>
</tbody>
</table>

Details of C672x Addressable Memory

The C672x processor has a unique EMIF which only contains two external addressable spaces. The address range at 0x80000000 allows for easy connection to single data rate SDRAM devices while the address range at 0x90000000 is intended for connection to FLASH and other asynchronous memories. There is also a 348 KB internal ROM.

<table>
<thead>
<tr>
<th>Devices</th>
<th>L2</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6727 C6726</td>
<td>256 KB</td>
<td>256 MB 64 MB</td>
</tr>
<tr>
<td>C6722</td>
<td>128 KB</td>
<td>64 MB (16-bit wide)</td>
</tr>
<tr>
<td>C6720</td>
<td>64 KB</td>
<td>64 MB (16-bit wide)</td>
</tr>
</tbody>
</table>
C64x Addressable Memory

The C64x devices add a couple important EMIF features, namely, many incorporate a 64-bit wide bus. Additionally, many C64x devices add a second, albeit smaller EMIF to increase throughput and allow designers to separate slow FLASH type memories from fast memory bus.

C64x Details

- **C64x Internal Memory**
  - Larger (256K or 1M bytes)
- **C64x External Memory**
  - Some have two EMIF’s

<table>
<thead>
<tr>
<th>Devices</th>
<th>L2</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6410</td>
<td>128KB</td>
<td>1GB</td>
</tr>
<tr>
<td>C6412</td>
<td>256KB</td>
<td>1GB</td>
</tr>
<tr>
<td>C6413</td>
<td>256KB</td>
<td>1GB</td>
</tr>
<tr>
<td>C6418</td>
<td>512KB</td>
<td>1GB</td>
</tr>
<tr>
<td>C6414/15</td>
<td>1MB</td>
<td>A: 1GB</td>
</tr>
<tr>
<td>DM640/16</td>
<td></td>
<td>B: 256MB</td>
</tr>
<tr>
<td>DM642/63</td>
<td>256KB</td>
<td>1GB</td>
</tr>
</tbody>
</table>

- Level 2 Internal Memory

For additional C64x devices, please refer to the DSP Product Selection Guide.
Section → Memory Placement

What is a Section?

Looking at the program component, you'll notice it contains code (algorithms) and multiple data sections (data structures).

The various parts of a program are called Sections. Breaking the program code and data into various sections provides flexibility since it allows you to place code sections in ROM and variable in RAM. The preceding diagram illustrates five sections.
**Section Placement Exercise**

Where would you anticipate these sections should be placed into memory? Try your hand at placing all three sections and tell us why you would locate them there.

<table>
<thead>
<tr>
<th>Section</th>
<th>Location</th>
<th>Why</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>C000_0000</td>
<td></td>
</tr>
<tr>
<td>.cinit</td>
<td>6000_0000</td>
<td></td>
</tr>
<tr>
<td>.bss</td>
<td>CS2</td>
<td></td>
</tr>
<tr>
<td>.stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.cio</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Hint:** Think about what type of memory each one should reside in – ROM or RAM.
Exercise Solution

Here's our solution. Many solutions are possible, here's ours:

<table>
<thead>
<tr>
<th>Section</th>
<th>Location</th>
<th>Why</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>FLASH</td>
<td>Must exist after reset</td>
</tr>
<tr>
<td>.cinit</td>
<td>FLASH</td>
<td>Must exist after reset</td>
</tr>
<tr>
<td>.bss</td>
<td>Internal</td>
<td>Must be in RAM memory</td>
</tr>
<tr>
<td>.stack</td>
<td>Internal</td>
<td>Must be in RAM memory</td>
</tr>
<tr>
<td>.cio</td>
<td>DDR2</td>
<td>Needs RAM, speed not critical</td>
</tr>
</tbody>
</table>

As discussed earlier, some sections need to exist before the CPU is released from reset. Specifically, .text must be available since it contains the code to be run after reset. The .cinit section will be used by the compiler to initialize all global and static variables. In both cases, we located our initialized sections into a “ROM” like memory; in this case, that was EPROM.

We chose to put the .bss and .stack sections into internal memory. Since internal memory provides faster access than external memory, this provides faster access to our global and local variables.

The .cio section contains the data buffers for the standard I/O functions. As these functions run rather slowly (as they do on any processor), we chose not to waste important on-chip memory for them.
Compiler’s Section Names - Review

These are the sections created by the C compiler.

<table>
<thead>
<tr>
<th>Section Name</th>
<th>Description</th>
<th>Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>Code</td>
<td>initialized</td>
</tr>
<tr>
<td>.switch</td>
<td>Tables for switch instructions</td>
<td>initialized</td>
</tr>
<tr>
<td>.const</td>
<td>Global and static string literals</td>
<td>initialized</td>
</tr>
<tr>
<td>.cinit</td>
<td>Initial values for global/static vars</td>
<td>initialized</td>
</tr>
<tr>
<td>.pinit</td>
<td>Initial values for C++ constructors</td>
<td>initialized</td>
</tr>
<tr>
<td>.bss</td>
<td>Global and static variables</td>
<td>uninitialized</td>
</tr>
<tr>
<td>.far</td>
<td>Aggregates (arrays &amp; structures)</td>
<td>uninitialized</td>
</tr>
<tr>
<td>.stack</td>
<td>Stack (local variables)</td>
<td>uninitialized</td>
</tr>
<tr>
<td>.sysmem</td>
<td>Memory for malloc fcns (heap)</td>
<td>uninitialized</td>
</tr>
<tr>
<td>.cio</td>
<td>Buffers for stdio functions</td>
<td>uninitialized</td>
</tr>
</tbody>
</table>

The compiler uses an additional section (.pinit) for C++ source code. Other than these, the TI C compiler will not generate any other section names. In the next chapter we discuss how you can create other section names, which is needed when you need to put specific functions or data into specific places in the memory-map.

Next, we will look at some of the sections created (or rather, we should say are a part of) the DSP/BIOS library of routines. Many of these are added to our system when linking in code generated by the Configuration Tool.

Sidebar: Initialized vs. Uninitialized Memory

Earlier in the chapter we used the terms RAM-like and ROM-like to describe types of memory. Read Only Memory (ROM) is usually hard-wired in some fashion, such that it retains its values after power has been removed and reapplied. Thus, we call this initialized memory since it retains its initial values, even after power cycling.

On the other hand, Random Access Memory (RAM) usually refers to memory that can be read and written to – like variables in our C code. As with variables, its value is uninitialized upon power up. Before being used, the memory must be set to some appropriate value (i.e. initialized to some value.)

In most cases, you will find the sections listed above requiring initialized memory (e.g. .text) in a ROM-like (initialized) memory. This is the simplest method of creating a system; and thus, was the solution to our earlier exercise.

More advanced systems might find other ways to handle initialized sections. For example, another host microprocessor (DSP or otherwise) could pre-load RAM memory after the system is powered-up. To the slave DSP, this pre-initialized RAM would appear as initialized memory – with valid information. The C6000 BIOS Integration Workshop discusses the bootload features of the C6000 family, and how this feature can be used to preload on-chip RAM before the CPU begins running.
System Software Initialization Sections

Software systems must get initialized. When writing in C code and including a TCF file in your project, the Software System Initialization is done for you.

Notice the two section names used by the tools to “hold” the reset vector and system initialization code. They can be placed into a specific memory region using the Config Tool. Later in the workshop we will cover other (hardware) system initialization topics.
Sidebar: What is Reset?

Reset is an external pin on all C6000 devices (in fact, on all microprocessors). This pin can stop all processing and return the device to a known (pre-described) state.

**What is Reset?**

- When the RESET pin on the processor is driven low, then high, the device is reset

  ![Reset Diagram]

- Upon reset four actions occur:
  1. All processing stops immediately
  2. Some registers set to pre-defined state
  3. Program Counter (PC) is set to zero
  4. Begin running code (from address 0)

Sidebar: What happens during Software System Init?

The compiler and Config Tool provide software needed to initialize their software environments. In many embedded systems this must be written by the user. Luckily, this has already been done.

### Software System Initialization

1. Initialize Pointers for compiler
   - stack
   - heap
   - global/static var's
2. Initialize global and static var's
3. Initialize DSP/BIOS objects
4. Call _main

```c
short m = 10;
short b = 2;
short y = 0;
main()
{
    short x = 0;
    scanf(x);
    malloc(y);
    y = m * x;
    y = y + b;
}
```
Using the Linker

Overview of Linking

The linker was briefly discussed earlier along with the rest of the TI DSP tool set. Its main purpose is to link together various object files. It combines like-named input sections from the various input object files and places each new output section at specific locations in memory. The linker can create two outputs, the executable (.out) file and a report which describes the results of linking (.map).
How Do You Place Sections into Memory Regions?

Now that we have defined these sections and where we want them to go, how do you create the memory areas that they are linked to and how do you actually link them there?

Linking code is a three step process:

1. Define memory objects (e.g. FLASH, SDRAM)
2. Place the sections into these memory areas?
3. Run the linker with “Build”
1. Creating a New Memory Objects (Using MEM)

First, to create a specific memory object, open up the .TCF file, right-click on the Memory Section Manager and select “Insert MEM”. Give this object a unique name and then specify its base and length. Once created, you can place sections into it (next step).

**Creating Memory Objects**

- MEM Manager allows you to create memory Objects & place sections
- To Create a New Memory Object:
  - Right-click on MEM and select Insert Mem
  - Fill in base/len, etc.

How do you place sections into these memory areas?

**Note:** The heap part of this dialog box is discussed in a later chapter.
2. Placing Sections – MEM Manager Properties

The configuration tool makes it easy to place sections. The predefined compiler sections that were described earlier each have their own drop-down menu to select one of the memory regions you defined (in step 1).

MEM Manager Properties

To Place a Section Into a Memory Object…

1. Right-click on MEM Section Manager and select Properties
2. Select the appropriate tab (e.g. Compiler)
3. Select the memory object for each section

<table>
<thead>
<tr>
<th>MEM - Memory Section Manager Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
</tr>
<tr>
<td>Test Section (text):</td>
</tr>
<tr>
<td>Switch Jump Tables (switch):</td>
</tr>
<tr>
<td>C Variables Section (.bss):</td>
</tr>
<tr>
<td>C Variables Section (.init):</td>
</tr>
<tr>
<td>Data Initialization Section (.init):</td>
</tr>
<tr>
<td>C Function Initialization Table (.init):</td>
</tr>
<tr>
<td>Constant Section (.const):</td>
</tr>
<tr>
<td>Data Section (.data):</td>
</tr>
<tr>
<td>Data Section (.bss):</td>
</tr>
</tbody>
</table>

What about the BIOS Sections?
Using the Linker

There are 3 tabbed pages of pre-defined section names:
(1) BIOS Data Sections
(2) BIOS Code Sections
(3) Compiler sections

We just looked at the 3rd page where the compiler's sections are allocated. If you look on the BIOS Code Sections page, you will find the system initialization sections discussed earlier in the chapter.

Placing BIOS Sections

- BIOS creates both Data and Code sections
- User needs to place these into appropriate memory region

What gets created after you make these selections?

We don't have time to describe all the BIOS-related sections. Please refer to the online help for a description of each or attend the 4-day workshop: *C6000 Embedded System Design Workshop using DSP/BIOS*.

At times you may need to define and place your own user-defined sections, this is discussed later in the next chapter.
3. Running the Linker

Creating the Linker Command File (via .TCF)

When you have finished creating memory objects and allocating sections into these memory them (which happens when you create and save the .TCF file), the DSP/BIOS Configuration Tool creates five files. One of the files is BIOS’s *cfg.cmd file — a linker command file.

This file contains two main parts, MEMORY and SECTIONS. (Though, if you open and examine it, it’s not quite as nicely laid out as shown above.)
Page left nearly blank ...
Running the Linker

The linker’s main purpose is to link together various object files. It combines like-named input sections from the various object files and places each new output section at specific locations in memory. In the process, it resolves (provides actual addresses for) all of the symbols described in your code.

The linker creates two outputs, the executable (.out) file and a report which describes the results of linking (.map).

Note: If the graphic above wasn’t clear enough, the linker is invoked automatically when you BUILD or REBUILD your project.
Sidebar: Linker Command File (CMD)

The linker is controlled by a linker command file (.CMD). Therefore, next to building the actual hardware, creating the linker command file is the crux of memory management.

The basic form of the linker command file has two parts:
- System Memory Description (MEMORY)
- Section Placement Description (SECTIONS)

Here’s an example of a linker command file:

```plaintext
MEMORY
{ FLASH: org = 0h, len = 400000h
  IRAM: org = 1400000h, len = 10000h
  DDR2: org = 2000000h, len = 1000000h
}
SECTIONS
{ .cinit :> FLASH
  .const :> FLASH
  .text :> FLASH
  .switch :> IRAM
  .bss :> IRAM
  .far :> DDR2
  .stack :> IRAM
  .cio :> DDR2
}
```

The first part of the linker command file describes the memory addresses available to the system. This includes internal memory, external memory, and any memory-mapped peripherals (internal or external).

This part of the linker command file allows you to bind (i.e. place) a section to a particular region of memory. In doing so, it resolves all relocatable addresses. In other words, all the labels (or symbols) you used in your code are given a specific address. We use labels, the processor uses addresses. (Note, we use label and symbol synonymously.)

If multiple sections are placed into the same memory region, they are placed following each other in the order listed in the linker command file.

**Note:** The C6000 is a byte addressable processor. That is, each address specifies the location of one byte. Therefore, when calculating the lengths of memories for the linker command file (CMD) or while using the Config Tool’s MEM manager you should specify all lengths in bytes.
Advanced Memory Management

Introduction

Advance memory management involves using memory efficiently. We will step through a number of options that can help you optimize your memory usage as well as your performance needs.

Outline

- Accessing Global Memory Efficiently
- Using Memory Efficiently
  - Keep it on-chip
  - Use multiple sections
  - Use local variables (stack)
  - Using dynamic memory (heap, BUF)
  - Overlay memory (load vs. run)
  - Use cache
Chapter 12 Topics

Advanced Memory Management

Accessing Global Memory Efficiently

Near/Far Data Accesses

Using Memory Efficiently

Keep it On-Chip

Using Multiple Sections

Custom Sections

What is the “.far” Section?

Link Custom Sections

Using Local Variables

Everything you wanted or didn’t want to know about the stack

Sidebar: How to PUSH and POP Registers

Setting the Stack Size

Using the Heap

Multiple Heaps

Using MEM_alloc

Using BUF

Memory Overlays

Implementing Overlays (code overlay example)

Memory Overlays Summary

Copy Tables

Sidebar: Linker Groups and Unions

Cache

Summary
Part of advance memory management includes accessing global memory efficiently.

Can you remember which instructions to use to load \( x \) into register A7?

It takes three instructions to load a global variable. But, there’s a special way to access \( x \) …
... by using the DP (global Data Pointer). DP is created by the compiler at runtime (during .sysinit, discussed in the last chapter) and is located in register B14. This value points to the top of all global/static variables, which you may remember are stored in the .bss section. Using the DP to access global variables is faster, because it only takes one instruction to access the value. Also, using the DP allows traceability of the compiler.

**Near Addressing using Global Data Pointer**

![Diagram of near addressing using global data pointer]

B14 – used by DP – is one of the only two registers that allow a 15-bit offset. This allows DP to quickly access up to 32K variables in .bss. On the other hand, you cannot access more than 32K variables in .bss. Therefore, if you have a number of large arrays, you may want to place some of them into another data section (which will be discussed next). Alternatively, you can define all data accesses to be far, that is, all data accesses should use MVKL/MVKH.

**Note:** If you have used previous TI DSPs (320C30, 320C50, etc.), please do not confuse the global pointer (DP) on the C6000 with the Data Page Register (also called DP) on previous processors. They are two different things.

- C6000’s global pointer is used as an optimization for quickly accessing global and static variables from the .bss section.

- Previous processors used the Data Page register to hold the upper address bits of a value being accessed via direct addressing. Due to the C6000’s RISC nature, it does not support direct addressing and therefore does not need the Data Page register capability.

It is unfortunate that these two items have the same DP abbreviation.
Updating the diagram from Chapter 2 regarding the C6000 compiler’s usage of CPU data registers, we have:
**Near/Far Data Accesses**

The compiler offers multiple addressing options that can be chosen to fit your needs. By default, the compiler accesses scalar data from .bss using near addressing; aggregate data (structures, unions, and arrays) are accessed from .far using far addressing.

Default addressing models can be changed using the --mem_model compiler option.

### Near/Far Data Accesses

#### How the compiler handles global and static data:

<table>
<thead>
<tr>
<th>Scalar Data</th>
<th>Aggregate Data (structs/arrays/unions)</th>
<th>Compiler Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>near</td>
<td>far</td>
<td>--mem_model:=far_aggregates   (default)</td>
</tr>
<tr>
<td>near</td>
<td>near</td>
<td>--mem_model:data=near</td>
</tr>
<tr>
<td>far</td>
<td>far</td>
<td>--mem_model:data=far</td>
</tr>
</tbody>
</table>

#### Override defaults with near/far keywords:

```c
short near myForceNearVar = 25;
short far myForceFarVar = 25;
```

The `near` and `far` keywords allow you to override the current memory model on a case by case basis. Thus, you can make all accesses `far`, except those you specify as `near`; or vice versa.
Using Memory Efficiently

Keep it On-Chip

One challenge for the system designer is to figure out where everything should be placed. Putting everything on-chip is the easiest way to maximize performance.

From earlier discussions in this chapter, remember that two sections hold most of our code and data. They are:

- .text - code and
- .bss - global and static variables.
Unfortunately, keeping everything on-chip is not always possible. Often code and data will require too much space and you are left with the decision of what should be kept on-chip and what can reside off-chip. Here are 5 other techniques to help you make the best use of on-chip memory and maximize performance.

**How to use Internal Memory Efficiently**

1. Keep it on-chip
2. Use multiple sections
3. Use local variables (stack)
4. Using dynamic memory (heap, BUF)
5. Overlay memory (load vs. run)
6. Use cache
Using Multiple Sections

If your code and data cannot all fit on-chip, create multiple sections.

If these sections are too big to fit on-chip, you will have to place them off-chip. But you may still want to put some critical functions and/or data on-chip.
Custom Sections

In order to use multiple sections, you’ll need a way to create them:

Making Custom Code Sections

- Create custom code section using

  ```
  #pragma CODE_SECTION (dotp, "critical");
  int dotp(a, x)
  ```

- Use the compiler’s --mo option
  - -mo creates a subsection for each function
  - Subsections are specified with ":"

  ```
  #pragma CODE_SECTION (dotp, ".text:_dotp");
  ```

To make a data section ...
Making Custom Data Sections

◆ Make custom named data section

```c
#pragma DATA_SECTION(x, "myVar");
#pragma DATA_SECTION(y, "myVar");
int x[32];
short y;
```

You will have to create new sections to keep critical code and data on-chip and other less critical (or very large) code and data sections off-chip.

**Hint:** Here is a little rule of thumb: “Create a new section for any code or data that must be placed in a specific memory location.”
What is the “.far” Section?

Rather than type in the whole DATA_SECTION pragma, if all you want to do is create a second data section, you can use the far keyword. Shown below are three different ways to create a variable m in the .far section.

Special Data Section: “.far”

- .far is a pre-defined section name
- Accessed with far addressing (mvkl || mvkh)
- Add variable to .far using:
  1. DATA_SECTION pragma
     ```
     #pragma DATA_SECTION(m, "far")
     short m;
     ```
  2. Far compiler option
     ```
     --mem_model:data=far
     ```
  3. Far keyword:
     ```
     far short m;
     ```

“Far” data is put into .far rather than .bss. Good default option because, rather than filling up .bss with aggregate data (which is usually accessed by loading a single pointer then using ptr manipulation), all of .bss is left for scalar data.

How do we link our own sections?

No matter how you create additional data sections, they will always be accessed using far addressing (MVKL/MVKH). Only .bss is ever accessed with the near addressing optimization (global Data Pointer).
Link Custom Sections

Recall that the CCS Memory Manager provided drop down boxes to aid with placing the compiler and Bios created sections. Unfortunately, there isn’t a way for TI to know what section names you might create, thus there are no drop-down boxes for custom section placement.

Rather, you must create your own linker command file, as shown below.

A few points:

1. Second, using the SECTIONS descriptor, list all the custom sections you have created and direct them into a MEM object. Each line “reads”:

   ```
   myVar :     >   SDRAM
   critical:  >   IRAM
   .text:_dotp:> IRAM
   ```

   To learn more about the SECTIONS directive, or linking in general, please refer to TMS320C6000 Assembly Language Users Guide (SPRU186).

2. You should not specify a section in both the Configuration Tool and your own linker command file.

3. You shouldn’t use the same label for a section name as you did for a label in your code. In other words, don’t put variable y into section “y”.

4. Specifying link order

If you have more than one linker command file, how do you specify the order they are executed?

If you are concerned that you might forget a custom-named section (or a team member might create one without telling you), the --w linker option can warn you of unspecified sections:
Using Local Variables

Whenever a new function is encountered, its local variables are automatically created on the software stack. Upon exiting the function, they are deleted from the stack. While most folks today call them “local” variables, they often used to be called “auto” variables. (A fitting name in that they are automatically allocated and deallocated from memory as they’re needed.)

Linking the software stack (.stack) into on-chip memory – and using local variables – can be an excellent way to increase on-chip memory efficiency … and performance.
**Everything you wanted or didn’t want to know about the stack**

Why learn about the stack? It is important to learn about the stack so you can trace what the compiler is doing, write assembly ISRs (Interrupt Service Routines), and because engineers want to know or think they need to know about the stack. So, here it goes!

The C/C++ compiler uses a stack to:
- Save function return addresses
- Allocate local variables
- Pass arguments to functions
- Save temporary results

The run-time stack grows from the high addresses to the low addresses. The compiler uses the B15 register to manage this stack. B15 is the stack pointer (SP), which points to the next unused location on the stack.

The linker sets the stack size to a default of 1024 bytes. You can change the stack size at link time by using the --stack option with the linker command. The actual length and location of the stack is determined at link time. Your link command file can determine where the .stack section will reside. The stack pointer is initialized at system initialization.

If arguments are passed to a function, they are placed in registers or on the stack. Up to the first 10 arguments are passed in even number registers alternating between A registers and B registers starting with A4, B4, A6, B6, and so on. If the arguments are longs, doubles, or long doubles, they are placed in register pairs A5:A4, B5:B4, A7:A6, and so on.
Using Memory Efficiently

Any remaining arguments are place on the stack. The stack pointer (SP) points to the next free location. This is where the eleventh argument and so on would be placed. Arguments place on the stack must be aligned to a value appropriate for their size. An argument that is not declared in a prototype and whose size is less than the size of int is passed as an int. An argument that is a float is passed as double if it has no prototype declared. A structure argument is passes as the address of the structure. It is up to the called function to make a local copy.

**Sidebar: How to PUSH and POP Registers**

*How would you PUSH “A1” to the stack?*

```
STW A1, *SP--[1]
```

*How about POPing A1?*

```
LDW *++SP[1], A1
```

**Using the Stack in Asm**

Rather than affecting PUSH/POP instructions, the compiler moves the stack pointer to reserve an entire frame (i.e. space needed for its operations), then fills it in.

**Using the Stack in Assembly**

```
; PUSH nine registers -- “A0” thru “A8”
SP .equ B15
STW A0, *SP--[10] ;
STW A1, *+SP[9]
STW A2, *+SP[8]
STW A3, *+SP[7]
STW A4, *+SP[6]
STW A5, *+SP[5]
STW A6, *+SP[4]
STW A7, *+SP[3]
STW A8, *+SP[2]
```

The compiler always keeps the stack pointer aligned. Until recently, it was always aligned to a 64-bit (i.e. double) boundary. With the advent of 128-bit datatypes for vector operations, the compiler will force alignment to 128-bits if any of those instructions are specified.
Setting the Stack Size

Setting Stack Size (.stack)

Stack Size (stack): 0x400
Using the Heap

When the term *dynamic memory* is used, though, most users are referring to the heap.

In addition to using a stack, C compilers provide another block of memory that can be user-allocated during program execution (i.e. at runtime). It is sometimes called System Memory (.sysmem), or more commonly, the *heap*.

For example ...

---

**Dynamic Memory Usage (Heap)**

- **Internal SRAM**
- **External Memory**

**Using Memory Efficiently**

3. **Local Variables**
   - If stack is located on-chip, all functions can use it

4. **Use the Heap**
   - Common memory reuse within C language
   - A Heap (i.e. system memory) allocates, then frees chunks of memory from a common system block
Here is an example using dynamic memory; in fact, it provides a good comparison between using traditional static variable definitions and their dynamic counterparts.

```
#define SIZE 32
int x[SIZE]; /*allocate*/
int a[SIZE];
x={…}; /*initialize*/
a={…};
filter(…); /*execute*/
```

```
#define SIZE 32
int x=malloc(SIZE);
int a=malloc(SIZE);
x={…};
a={…};
filter(…);
free(a);
free(x);
```

malloc() is a standard C language function that allocates space from the heap and returns an address to that space.

The big advantage of dynamic allocation is that you can free it, then re-use that memory for something else later in your program. This is not possible using static allocations of memory (where the linker allocates memory once-and-for-all during program build).
**Multiple Heaps**

Assuming you have infinite memory (like most introduction to C classes assume), one heap should be enough. In the real world, though, you may want more than one. For example, what if you want both an off-chip and an on-chip heap?

Just as we discussed earlier with *Multiple Sections* for code and data, multiple heaps allows you to target critical elements on-chip, while less critical (or larger ones) can be allocated off-chip.
While standard C compilers do not provide multiple heap capability, TI’s DSP/BIOS tools do. When creating MEM objects, you have the option to create a heap in that memory space. Just indicate you want a heap (with a checkmark) and set the size. From henceforth, you can refer to this specific heap by its MEM object name.

Alternatively, if you don’t want to use the MEM object name to refer to a heap you can define a separate identification label.
Using Memory Efficiently

Using MEM_alloc

Q: If standard C doesn’t provide multi-heap capabilities, how would the standard C functions like malloc() know which heap to use?

A: They can’t know.

Solution: Use the DSP/BIOS MEM_alloc() function as opposed to malloc().

<table>
<thead>
<tr>
<th>Standard C syntax</th>
<th>Using MEM functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define SIZE 32</td>
<td>#define SIZE 32</td>
</tr>
<tr>
<td>x = malloc(SIZE);</td>
<td>x = MEM_alloc(IRAM, SIZE, ALIGN);</td>
</tr>
<tr>
<td>a = malloc(SIZE);</td>
<td>a = MEM_alloc(SDRAM, SIZE, ALIGN);</td>
</tr>
<tr>
<td>x = {...};</td>
<td>x = {...};</td>
</tr>
<tr>
<td>a = {...};</td>
<td>a = {...};</td>
</tr>
<tr>
<td>filter(...);</td>
<td>filter(...);</td>
</tr>
<tr>
<td>free(a);</td>
<td>MEM_free(SDRAM, a, SIZE);</td>
</tr>
<tr>
<td>free(x);</td>
<td>MEM_free(IRAM, x, SIZE);</td>
</tr>
</tbody>
</table>

You can pick a specific heap

As you can see, there is also MEM_free() to replace free(). Additional substations can be found in the DSP/BIOS library.
Using BUF

While using dynamic memory via the heap is advantageous from a memory reuse perspective, it does have its drawbacks.

Heap drawbacks:

- Allocation calls (i.e. malloc) are non-deterministic. That is, each time they are called they make take longer or shorter to complete.
- The allocation functions are non-reentrant. For example, if malloc() is called while a malloc() is already running (say, it was called in a hardware interrupt service routine), the system may break.
- Heap allocations are prone to memory fragmentation if many malloc's and free's are called.

BUF solves these problems by letting users create pools of buffers that can then be allocated, used, and set free.

- **Buffer pools** contain a specified number of equal size buffers
- Any number of pools can be created
- Buffers are allocated from a pool and freed back when no longer needed
- Buffers can be shared between applications
- Buffer pool API are faster and smaller than malloc-type operations
- In addition, BUF_alloc and BUF_free are deterministic (unlike malloc)
- BUF API have no reentrancy or fragmentation issues
Creating of Buffer Pool (BUF)

Creating a BUF
1. right click on BUF mgr
2. select “insert BUF”
3. right click on new BUF
4. select “properties”
5. indicate desired
   • Memory segment
   • Number of buffers
   • Size of buffers
   • Alignment of buffers
   • Gray boxes indicate effective pool and buffer sizes
Memory Overlays

Another traditional method of maximizing use of on-chip memory is to *overlay* code and data. (You could even substitute the term *overlap* for *overlay.* ) While each exists on its own externally, they run from the same overlayed locations, internally.

With overlays, each code or data item must reside in its own starting location. The TI tools call this its *load* location, because this is what is downloaded to the system (when using the CCS Load Program menu item, or when you download to an EPROM via an EPROM programmer).

During program execution, your code must copy the overlayed data or code elements into their *run* location. This is where the program expects the information to reside when it is used (i.e. when the overlayed function is called, or the overlayed data elements are accessed). The linker resolves all your code/data labels (i.e. symbols) to the *run*time addresses.

How do you implement overlays, follow these 3 steps …
Implementing Overlays (code overlay example)

1. Create a section for each item you want to overlay.

For example, if you wanted two functions to be overlayed, create them with their own sections.

```c
#pragma CODE_SECTION(fir, "FIR");
int fir(short *a, ...)
```

```c
#pragma CODE_SECTION(iir, "myIIR");
int iir(short *a, ...)
```

We arbitrarily chose the section names .fir and myIIR.

---

How can we get them to run from the same location?

Where will they be originally loaded into memory?

The key is in the linker command file ...
2. Create your own linker command file (as discussed earlier for Multiple Sections).

   Earlier we put something like this into our SECTIONS part of the linker command file.

   [.bss :> IRAM]

   This could be re-written as:

   [.bss: load = IRAM, run = IRAM]

In the case of our overlayed functions, though, we don’t want them to be loaded-to and run-
from the same locations in memory, therefore, we might try something like:

   [.fir: load = EPROM, run = IRAM]
   myIIR: load = EPROM, run = IRAM

   In this case, they are both loaded into EPROM and Run from IRAM.
Using Memory Efficiently

Load vs. Run Addresses

- Simply specify different addresses for load and run
- You must make sure they get copied (using the memcopy or the DMA)

Back to our original problem, what if we want them to run from the same address?

The problem is that the linker assigns different run addresses for both functions. But, we wanted them to share (i.e. overlap) their run addresses. How can we make this happen?
Using Memory Efficiently

Use the linker’s UNION command. The union concept is similar to that of creating union types in the C language. In our case, we want to tell the linker to put the run addresses of the two functions in union.

```cpp
UNION run= IRAM
{
    .fir:  load = EPROM, table(_fir_copy_table)
    myIIR: load = EPROM, table(_iir_copy_table)
}
```

This then, allocates separate load addresses for each function, while providing a single run address for both functions.

**Note:** To set separate load and run addresses for pre-defined BIOS and Compiler sections, there is an additional tabbed page in the CCS Config Tools Memory Section Manager dialog.
3. **Last, but not least, you must copy the code from its original location to its runtime location.** Before you run each function you must force the code (or data, in a data overlay) to be copied from its load addresses to its run addresses. When using the Copy Table feature of the linker, copying code from its original location is quite easy.

```c
#include <cpy_tbl.h>
extern far COPY_TABLE fir_copy_table;
extern far COPY_TABLE iir_copy_table;
extern void fir(void);
extern void iir(void);
main()
{  copy_in(&fir_copy_table);
    fir();
    ...
  copy_in(& iir_copy_table);
    iir();
    ...
}
```

The `copy_in()` function is a simple wrapper around the compiler’s `mem_copy()` function. It reads the table description created by the “table” feature of the linker and uses it to perform a `mem_copy()`.

From a performance standpoint, though, you are better off using the DMA or EDMA hardware peripherals. These hardware peripherals can be easily used to copy these tables by using the `DAT_copy()` function from TI's Chip Support Library (CSL).

**Memory Overlays Summary**

- **First**, create a `section` for each function
- **In your own linker cmd file**:  
  - `load`: where the fnx resides at reset  
  - `run`: tells linker its runtime location  
  - `UNION` forces both functions to be runtime linked to the same memory addresses (i.e. overlayed)  
- **You** must move it with CPU or DMA

```c
myLnk.CMD
SECTIONS
{  .bss: > IRAM  /*load & run*/
    UNION    run = IRAM
    {       .FIR : load = EPROM
            myIIR: load = EPROM
    }
```
**Copy Tables**

**Creating Copy Tables**

An easy way to generate the addresses required for overlays is to use copy tables. This is done using the table directive in the linker command file:

```
.FIR : load = EPROM, table(_fir_copy_table)
```

Using the `table` command tells the linker to create a copy table and record. Each record contains a description required for a memory copy:

- Source address ("load address")
- Destination address ("run address")
- Size of information to copy

These records are combined into a **Copy Table**, meaning that you might want to have multiple records of information all moved at the same time. For example, when you want to do an overlay, maybe you want to move a set of data and the code to process that data at the same time. This might cause you to have two Copy Records in a single Copy Table.

```c
typedef struct copy_record
{  unsigned int load_addr;
   unsigned int run_addr;
   unsigned int size;
} COPY_RECORD;

typedef struct copy_table
{  unsigned short rec_size;
   unsigned short num_recs;
   COPY_RECORD recs[2];
} COPY_TABLE;
```

**How do we use a Copy Table?**
Using Copy Tables

The Runtime Support Library contains a copy_in() function that makes easy use of the Copy_Table.

```c
#include <cpy_tbl.h>
extern far COPY_TABLE fir_copy_table;
extern far COPY_TABLE iir_copy_table;
extern void fir(void);
extern void iir(void);
main()
{  copy_in(&fir_copy_table);
   fir();
   ...
   copy_in(&iir_copy_table);
   iir();
   ...
}
```

As described in the graphic above, copy_in() provides a simple wrapper around mem_copy() -- which is a commonly known and used function from standard C compiler support libraries. Given the copy table address, copy_in() reads each record in the copy table and calls mem_copy().

While copy_in() is very convenient, it is not an efficient way to move blocks of memory on the C6000 family. Rather, using the on-chip DMA peripheral is much more efficient. The Chip Support Library provides a simple wrapper for the DMAs on the C6000 devices called DAT. In fact, the DAT_copy() function -- similar to the mem_copy() function, but using the DMA -- is an excellent way to move copy records.
Overlays can be very useful, but they’re also tedious to setup. Isn’t there an easier way to get the advantages of overlays? …
Cache

Data and program caching provides the benefits of memory overlays, without all the hassles.

Since the C6711 has both data and program cache hardware, this is the easiest method of overlaying memory (and hence, most commonly used).

Rather than discuss cache in detail here, the next chapter is dedicated to this topic.
Summary

You may notice the order in the summary is a bit different from that which we just discussed the topics. While introducing them to you, we wanted to build the concepts piece-by-piece. In real life, though, as you design your system you will probably want to employ them in the following order.

Summary: Using Memory Efficiently

- You may want to work through your memory allocations in the following order:
  1. Keep it all on-chip
  2. Use Cache (more in Ch 13)
  3. Use local variables (stack on-chip)
  4. Using dynamic memory (heap, BUF)
  5. Make your own sections (pragma’s)
  6. Overlay memory (load vs. run)

- While this tradeoff is highly application dependent, this is a good place to start

For example,

1. If you can get everything on-chip, you’re done.

2. If it won’t all fit, you might try enabling the cache. If your system meets its real-time deadlines, you’re now done.

3. In most cases, you’ve probably already used local variables whenever possible. So this one is probably a ‘given’.

4. If you’ve enabled the cache and still need to tweak the system for performance, you might try to using dynamic memory
   … or one of the remaining options.

The advantage to the top 4 methods is that they can all be done from within your C code. The remaining two require a custom linker command file (or modification of your .cmd file). (Not difficult, but one more thing to manage.)
Introduction

As the performance of DSPs increase, the ability to put large, fast memories on-chip decreases. Current silicon technology has the ability to dramatically increase the speed of DSP cores, but the speed of the memories needed to provide single-cycle access for data and instructions to these cores are limited in size. In order to keep DSP performance high while reducing cost, large, flat memory models are being abandoned in favor of caching architectures. Caching memory architectures allow small, fast memories to be used in conjunction with larger, slower memories and a cache controller that moves data and instructions closer to the core as they are needed. Most C6000 devices provide a two-level cache architecture that is flexible and powerful. We'll look at how to configure the cache and use it effectively in a system.

Outline

- Why Cache?
- Cache Basics
- Cache Example (Direct-Mapped)
- L1 Program (Linesize, Cache/RAM)
- L1 Data (Two-Way Cache)
- L2 Memory
- Cache Coherency
- Additional Memory/Cache Topics
- Lab 13
- Optional Topics
Chapter Topics

Why Cache? ...........................................................................................................................................13-3

Cache vs. RAM ........................................................................................................................................13-5

Cache Fundamentals ...............................................................................................................................13-7

Direct-Mapped Cache ..............................................................................................................................13-10

Direct-Mapped Cache Example .............................................................................................................13-11

Three Types of Misses ...........................................................................................................................13-19

Cache Tuning Tool .................................................................................................................................13-19

Internal Memory Hierarchy .....................................................................................................................13-20

L1 Program Cache (L1P) ...........................................................................................................................13-23

Cache Term: Linesize ...............................................................................................................................13-24

Cache vs. Addressable RAM ..................................................................................................................13-26

Cache Freeze .........................................................................................................................................13-26

L1 Data Cache (L1D) .................................................................................................................................13-27

A Way Better Cache .................................................................................................................................13-28

Cache Sets ...............................................................................................................................................13-29

L1 Data (L1D) Cache Summary .............................................................................................................13-31

L2 Memory .............................................................................................................................................13-32

L2 Configuration ......................................................................................................................................13-33

Why both RAM and Cache? ...................................................................................................................13-36

Data Cache Coherency ............................................................................................................................13-38

Example Problem ..................................................................................................................................13-38

Solution 1: Using Cache Flush & Clean ..................................................................................................13-42

Solution 2: Use L2 Memory ....................................................................................................................13-45

Cache Coherency Summary......................................................................................................................13-46

Cache Line Alignment .............................................................................................................................13-47

“Turn Off” the Cache (MAR) ....................................................................................................................13-48

Additional Memory/Cache Topics ..........................................................................................................13-51

'C64x Memory Banks ..............................................................................................................................13-51

Cache Optimization ..............................................................................................................................13-53

Cache Aware Linking ..............................................................................................................................13-54

Cache Terminology Summary ..................................................................................................................13-54
**Why Cache?**

In order to understand why the C6000 family of DSPs uses cache, let's consider a common problem. Take, for example, the last time you went to a crowded event like the symphony, a sporting event, or the ballet, any kind of event where a lot of people want to get to one place at the same time. How do you handle parking? You can only have so many parking spots close to the event. Since there are only so many of them, they demand a high price. They offer close, fast access to the event, but they are expensive and limited.

Your other option is the parking garage. It has plenty of spaces and it's not very expensive, but it is a ten minute walk and you are all dressed up and running late. It's probably even raining. Don't you wish you had another choice for parking?

You do! A valet service gives the same access as the close parking for just a little more cost than the parking garage. So, you arrive on time (and dry) and you still have money left over to buy some goodies.
Cache is the valet service of DSPs. Memory that is close to the processor and fast can only be so big. You can attach plenty of external memory, but it is slower. Cache helps solve this problem by keeping what you need close to the processor. It makes the close parking spaces look like the big parking garage around the corner.

One of the often overlooked advantages of cache is that it is automatic. Data that is requested by the CPU is moved automatically from slower memories to faster memories where it can be accessed quickly.
Cache vs. RAM

Using Internal Program as RAM

DSPs achieve their highest performance when running code from on-chip program RAM. If your program will fit into the on-chip program RAM, use the DMA or the Boot-Loader to copy it there during system initialization. This method of using the DMA or a Boot-Loader is powerful, but it requires the system designer to set everything up manually.

If your entire system code cannot fit on chip but individual, critical routines will fit, place them into the on-chip program RAM as needed using the DMA. Again, this method is manual and can become complex very quickly as the system changes and new routines are added.

In the example above, the system has three functions (func1, func2, and func3) that will fit in the on-chip program memory located at 0x0. The system designer can set up a DMA transfer from 0x8000 to 0x0 for the length of all three functions. Then, when the functions are executed they will run from quick on-chip memory.

Unfortunately, the details of setting up the DMA-copy are left to the designer. Several of these details change every time the system/code is modified (i.e. addresses, section lengths, etc.).

Worse yet, if the code grows beyond the size of the on-chip program memory, the designer will have to make some tough choices about what to execute internally, and which to leave running from external memory. Either that, or implement a more complicated system which includes overlays.
Using Cache

The cache feature of the ‘C6000 allows the designer to store code in large off-chip memories, while executing code loops from fast on-chip memory … automatically.

That is, the cache moves burden of memory management from the designer to the cache controller – which is built into the device.

Let’s start with Basic Concepts of a Cache ...

Notice that Cache, unlike the normal memory, does not have an address. The instructions that are stored in cache are associated with addresses in the memory map. Over the next few pages we further describe the term associated along with how cache works, in general.
Cache Fundamentals

As stated earlier, locations in cache memory do not have their own addresses. These locations are associated with other memory locations. You may think of it like cache locations “shadowing” addressable memory locations (usually a larger, slower-access memory).

As part of its function, cache hardware and memory must have an organizational method to keep track of what addressable memory locations it contains.

Blocks, Lines, Index

One way to think about how a direct-mapped cache works is to think of the entire memory map as blocks. These blocks are the same size as the cache. The cache block is further broken into lines. A line is the smallest element (location) that can be specified in a cache. Finally, we number each line in the cache. This is often called an index, or more obviously, line-number.

In the example above, the cache has 16 lines. Therefore, the entire memory map (or at least the part that can be cached) is broken up into 16 line blocks. The first line of each block is associated with the first line in cache; the second line of each block is associated with the second line of cache, continuing out to the 16th line. If the first line of cache is occupied by information from the first block and the DSP accesses the same line from the second block, the information in the cache will be overwritten because the two addresses reside at the same line.
Cache Tag

When values from memory are copied into a line or more of cache, how can we keep track of which block they are from?

The cache controller uses the address of an instruction to decide which line in cache it is associated with, and which block it came from. This effectively breaks the address into two pieces, the index and the tag. The index determines which line of cache an instruction will reside at in cache (and the lower order bits of the address represent it). The tag is the higher order bits of the address, and it determines which block the cache line is associated with in the memory map.

While a single tag will allow the cache to discern which block of memory is being “shadowed”, it requires all lines of the cache to be associated with the same block of memory. As caches become larger, as is the case with the C6000, you may want different lines to be associated with different blocks of memory. For this reason, each line has an associated tag.

- A **Tag** value keeps track of which block is associated with a cache block
- **Each line has its own tag** -- thus, the whole cache block won't be erased when lines from different memory blocks need to be cached simultaneously

How do we know a cache line is valid (or not)?
Valid Bits

Just because a cache can hold, say, 4K bytes, that doesn’t mean that all of its lines will always have valid data. Caches provide a separate valid bit for each line. When data is brought into the cache, the valid bit is set.

When a CPU load instruction reads data from an address, the cache is examined to see if the valid, specified address exists in the cache. That is, at the index specified by the address, does the correct tag value exist and is it marked valid?

Note: Given a 4K byte cache, do the bits associated with the cache management (tag, valid, etc.) use up part of the 4K bytes? The answer is No. When a 4K byte cache is specified, we are indicating the amount of usable memory.
Direct-Mapped Cache

A Direct-Mapped cache is a type of cache that associates each one of its lines with a line from each of the blocks in the memory map. So, only one line of information from any given block can be live in cache at a given time.

- Direct-Mapped Cache associates an address within each block with one cache line
- Thus ... there will be only one unique cache index for any address in the memory-map
- Only one block can have information in a cache line at any given time

Another way to think about this is, “For any given memory location, it will map into one, and-only-one, line in the cache.”
Direct-Mapped Cache Example

In the example below, we have a 16-line cache. How many bits are needed to address 16 lines? The answer of course is four, so this is the number of bits that we have as the index. If we have 16-bit addresses, and the lowest 4-bits are used for the index, this leaves 12-bits for the tag. The tag is used to determine from which 16-line block of memory the index came.

Let’s examine an arbitrary direct-mapped cache example:
- A 16-line, direct-mapped cache requires a 4-bit index.
- If our example µP used 16-bit addresses, this leaves us with a 12-bit tag.

The best way to understand how a cache works is by studying an example. The example below illustrates how a direct-mapped cache with 16-bit addresses operates on a small piece of code. We will use this example to understand basic cache operation and define several terms that are applicable to caches.

Arbitrary Direct-Mapped Cache Example

- The following example uses:
  - 16-line cache
  - 16-bit addresses, and
  - Stores one 32-bit instruction per line
- C6000 cache’s have different cache and line sizes than this example
- It is only intended as a simple cache example to reinforce cache concepts
Note: The following cache example does not illustrate the exact operation of a 'C6000 cache. The example has been simplified to allow us to focus on the basic operation of a direct-mapped cache. The operation of a 'C6000 cache follows the same basic principles.

Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0003h</td>
<td>L1</td>
</tr>
<tr>
<td>0004h</td>
<td>MPY</td>
</tr>
<tr>
<td>0005h</td>
<td>ADD</td>
</tr>
<tr>
<td>0006h</td>
<td>B L2</td>
</tr>
<tr>
<td>0026h</td>
<td>L2</td>
</tr>
<tr>
<td>0027h</td>
<td>ADD</td>
</tr>
<tr>
<td>0028h</td>
<td>SUB</td>
</tr>
<tr>
<td></td>
<td>cnt</td>
</tr>
<tr>
<td></td>
<td>[!cnt] B</td>
</tr>
<tr>
<td></td>
<td>L1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
The first time instructions are accessed the cache is cold. A cold cache doesn't have anything in it. When the DSP accesses the first instruction of our example code, the LDH, the cache controller uses the index, 3, to check the contents of the cache. The cache controller includes a valid bit for each line of cache. As you can see below, the valid bit for line 3 is not set. Therefore, the LDH instruction causes a cache miss. More specifically, this is called a compulsory miss. The instruction has to be fetched from memory at its address, 0x0003. This operation will cause a delay until the instruction is brought in from memory.

### Direct Mapped Cache Example

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Index</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

- **Address**: 0003h, 0004h, 0005h, 0006h, 0026h, 0027h, 0028h
- **Code**: LDH, MPY, ADD, SUB, cnt

**Compulsory Miss**
When the LDH instruction is brought in from memory, it is given to the core and added to the cache at the same time. This operation minimizes the delay to the core. When the instruction is added to the cache, it is added to the appropriate index line, the tag is updated, and the valid bit is set.

The following three instructions are added to the cache in the same manner. When they have all been accessed, the cache will look like this:

Notice that the branch instruction is the last instruction that was transferred by the cache controller. A branch by definition can take the DSP to a new location in memory. The branch in this case takes us to the label \(L2\), which is located at 0x0026.
When the CPU fetches the ADD instruction, it checks the cache to see if it currently resides there. The cache controller checks the index, 6, and finds that there is something valid in cache at this index. Unfortunately, the tag is not correct, so the add instruction must be fetched from memory at its address.

Since this is a direct-mapped cache, the ADD instruction will overwrite whatever is in cache at its index. So, in our example, the ADD will overwrite the B instruction since they share the same index, 6.
The DSP executes the instructions after the ADD, the SUB and the B. Since they are not valid in cache, they will cause cache misses.

When the branch executes, it will take the DSP to a new location in memory. The branch in this case takes the DSP to the address of the symbol \texttt{L1}, which is 0x0003. This is the address of the original LDH instruction from above.
When the DSP accesses the LDH instruction this time, it is found to be in cache. Therefore, it is given to the core without accessing memory, which removes any memory delays. This operation is called a *cache hit*.

A few observations can be made at this point. Instructions are added to cache only by accessing them. If they are only used once, the cache does not offer any benefit. However, it doesn't cause any additional delays. This type of cache has the biggest benefit for looped code, or code that is accessed over and over again. Fortunately, this is the most common type of code in DSP programming.

Notice also what seems to be happening at line 6. Each time the code runs, line 6 is overwritten twice. This behavior is called thrashing the cache. The cache misses that occur when you are thrashing the cache are called conflict misses. Why is it happening? Is it reducing the performance of the code?

Thrashing occurs when multiple elements that are executed at the same time live at the same line in the cache. Since it causes more memory accesses, it dramatically reduces the performance of the code. How can we remove thrashing from our code?
The thrashing problem is caused by the fact that the ADD and the B share the same index in memory. If they had different indexes, they would not thrash the cache. So, a simple fix to this problem is to make sure that the second piece of code (ADD, SUB, and B) doesn't share any indexes with the first chunk of code. A simple fix is to move the second chunk down by one line so that its indexes start at 7 instead of 6.

This relocation can be done several different ways. The simplest is probably to make the two sections contiguous in memory. Code that is contiguous and smaller than the size of the cache will not thrash because none of the indexes will overlap. Since code is placed in the same memory section a lot of the time, it will not thrash. Given the possibility of thrashing, caution should be exercised when creating different code sections in a cache based system.
Three Types of Misses

The types of misses that a cache encounters can be summarized into three different types.

- **Compulsory**
  - Miss when first accessing an new address

- **Conflict**
  - Line is evicted upon access of an address whose index is already cached
  - Solutions:
    - Change memory layout
    - Allow more lines for each index

- **Capacity** (we didn’t see this in our example)
  - Line is evicted before it can be re-used because capacity of the cache is exhausted
  - Solution: Increase cache size

Cache Tuning Tool

The CacheTune tool within CCS helps visualize different types of cache misses.

![CacheTune Diagram](image)
Internal Memory Hierarchy

As mentioned earlier in the workshop, the double-level memory devices provide three chunks of internal memory. Level 1 memories (being closest to the CPU) are provided as cache for both program (L1P) and data (L1D), respectively.

The third memory chunk is called L2 memory. The processor will look for an address in L1 memories first; if not found L2 memory is examined next. L2 memory may be addressable RAM or cache – its configurability will be discussed shortly.

Finally, on these DSPs, all external memory is considered Level three memory since it is the third location examined in the memory access hierarchy. Of course, this makes sense since external accesses are slower than internal accesses.
Memory Hierarchies

- A Memory Hierarchy organizes memory into different levels:
  - Higher Levels are closer to the CPU
  - Lower Levels are further away
- CPU requests are sent from higher levels to lower levels
- The higher levels are designed to keep information that the CPU needs based on:
  - Temporal Locality – most recently accessed
  - Spatial Locality – closest in memory
- Middle levels can buffer between small-fast memory and large-slow memory

Memory Hierarchy maximizes the effectiveness of its on and off chip memories. Small, fast memories are used close to CPU for performance – and large, slow memories off-chip for bulk storage. The cache controller is optimized to keep instructions and data that the core needs in the faster memories automatically with minimal effect on the system design. Large off-chip memories can be used to store large buffers without having to pay for larger memories on-chip, which can be expensive.

The L1P and L1D are the highest order memories in the hierarchy. As you move further away from these memories, performance decreases. CPU requests are first sent to these fast memories, then to slower memories lower in the hierarchy. The highest orders are designed to store the information that the CPU needs based on temporal and spatial locality. Intermediate levels can be inserted between the highest order (L1P and L1D) and the lowest order (external memory) to serve as a larger buffer that further increases performance of the memory system. Again, L2 is a middle hierarchical layer that helps the cache controller keep the items that the CPU will need next closer to the L1 memories.

Here is a simple flow chart of the decision process that the cache controller uses to fulfill CPU requests.
Double-Level Memory – Cache Logic

While each device’s cache logic may differ slightly from this, it should provide a good sense of the logic between the various cache levels.
The C6000 devices have a direct-mapped internal program cache called L1P which is 4K bytes large. The L1 cache is always enabled.

The various devices have different L1P sizes.

- **Zero-waitstate Program Memory**
- **Direct-Mapped Cache**
  - Works exceptionally well for DSP code (which tends to have many loops)
  - Can be placed to minimize thrashing

All L1P memories provide zero waitstate access.
The C6713 L1P has 4KB of cache broken into cache lines that store 16 instructions. So, the linesize of the L1P is 16 instructions. What do we mean by linesize …

**Cache Term: Linesize**

Our earlier direct-mapped cache example only stored one instruction per line; conversely the C6711 L1P cache line can hold 16 instructions. In essence, linesize specifies the number of addressable memory locations per line of cache.

Increasing the linesize does not change the basic concepts of cache. The cache is still organized with: blocks, lines, tags, and valid-bits. And cache accesses still result in hits and misses. What changes, though, is how much information is brought into cache when a miss occurs.

Let’s look at a simple linesize comparison. In this case, let’s look at a line that caches one byte of external memory …

![New Term: Linesize Diagram]

In our earlier cache example, the size was:
- **Size:** 16 bytes
- **Linesize:** 1 byte
- **# Of indexes:** 16

How else could it be configured?
As opposed to a linesize of one byte, here’s a linesize of two bytes:

**New Term: Linesize**

<table>
<thead>
<tr>
<th>Index</th>
<th>Cache</th>
<th>External Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0x8000</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0x8010</td>
</tr>
<tr>
<td>0x7</td>
<td>0xE</td>
<td>0x8020</td>
</tr>
</tbody>
</table>

In our earlier cache example, the size was:
- Size: 16 bytes
- Linesize: 1 byte
- # Of indexes: 16

We have now changed it to:
- Size: 16 bytes
- Linesize: 2 bytes
- # Of indexes: 8

What’s the advantage of greater line size? **Speed!** When cache retrieves one item, it gets another at the same time.

Notice that the block size is consistent in both examples. Of course, when the linesize is doubled, then number of indexes is cut in half.

Increasing the linesize often may increase the performance of a system. If you are accessing information sequentially (especially common when accessing code and arrays), while the first access to a line may take the extra time required to access the addressable memory, each subsequent access to the cache line will occur at the fast cache speeds.

Coming back to the L1P, when a miss occurs, not only do you get one 32-bit instruction, but the cache also brings in the next 15 instructions. Thus, if your code execute sequentially, on the first pass through your code loops, you will only receive one delay every 16 instructions rather than a delay for every instruction.

A direct mapped cache is very effective for program code where a sequence of instructions is executed one after the other. This effect is maximized for looped code, where the same instructions are executed over and over again. So a direct-mapped cache works well when a single element (instruction) is being accessed at a given time and the next element is contiguous in memory.

Will a direct mapped cache work well for data? We will see, in a couple of minutes.
Cache vs. Addressable RAM

C64x+ L1P – Cache vs. Addressable RAM

- Can be configured as Cache or Addressable RAM
- Five cache sizes are available: 0K, 4K, 8K, 16K, 32K
- Allows critical loops to be put into L1P, while still affording room for cache memory

Cache Freeze

Cache Freeze (C64x+)

- Freezing cache prevents data that is currently cached from being evicted
- Cache Freeze
  - Responds to read and write hits normally
  - No updating of cache on miss
  - Freeze supported on C64x+ L2/L1P/L1D
- Commonly used with Interrupt Service Routines so that one-use code does not replace realtime algo code
- Other cache modes: Normal, Bypass
- BCACHE: BIOS Cache management module

```c
// Cache Mode Management
typedef enum {
    BCACHE_NORMAL, 
    BCACHE_FREEZE, 
    BCACHE_BYPASS 
} BCACHE_Mode;

typedef enum {
    BCACHE_L1D, 
    BCACHE_L1P, 
    BCACHE_L2 
} BCACHE_Level;

Mode = BCACHE_getMode(level) // rtn state of specified cache
oldMode = BCACHE_setMode(level, mode) // set state of specified cache
```
L1 Data Cache (L1D)

The aspects that make a direct-mapped cache effective for code make it less useful for data. For example, the CPU only accesses one instruction at a time, but one instruction may access several pieces of data. Unlike code, these data elements may or may not be contiguous. If we consider a simple sum of products, the buffers themselves may be contiguous, but the individual elements are probably not. In order to avoid organizing the data so that each element is contiguous, which is difficult and confusing, a different kind of cache is needed.

![Caching Data Diagram]

- One instruction may access multiple data elements:
  ```c
  for( i = 0; i < 4; i++ ) {
      sum += x[i] * y[i];
  }
  ```

- What would happen if x and y ended up at the following addresses?
  - x = 0x8000
  - y = 0x9000

- Increasing the associativity of the cache will reduce this problem

If the addresses of X and Y both began at the start of a cache block, then they would end up overwriteing each other in the cache, which is called thrashing. x₀ would go into index 0, and then y₀ would overwrite it. x₁ would be placed in index 1, and then y₁ would overwrite it. And so on.
A Way Better Cache

Since multiple data elements may be accessed by one instruction, the associativity of the cache needs to be increased. Increasing the associativity allows items from the same line of multiple blocks to live in cache at the same time. Splitting the cache in half doubles its associativity. Take the C6713 L1P as an example of a single, 4Kbyte direct-mapped cache. Splitting it in half yields two blocks of 2Kbytes each – which is how the L1D cache is configured. These two blocks are called cache ways. Each way has half the number of lines of the original block, but each way can store the associated line from a block. So, two cache ways means that the same line from memory can be stored in each of the two cache ways.
# Cache Sets

All of the lines from the different cache ways that store the same line from memory form a **set**. For example, in a 2-way cache, the first line from each way stores the first line from each of the N blocks in memory. These two lines form a set, which is the group of lines that store the same indexes from memory. This type of cache is called a set associative-cache. So, if you have 2 cache ways, you have a 2-way set-associative cache.

**What is a Set?**

- **The lines from each way** that map to the same index form a set.
- **Set of index zeroes**, i.e. **Set 0**
- **The number of lines per set** defines the cache as an N-way set-associative cache
- **With 2 ways, there are now 2 unique cache locations for each memory address**

Another way to look at this is from the address point of view. In a direct-mapped cache, each index only appears once. In an N-way set-associative cache, each index appears N times. So, N items from the same index (with the same lower address bits) can reside in the cache at the same time. In reality, a direct-mapped cache can be thought of as a 1-way set-associative cache.

## Advantage of Multiple Cache Sets

The main reason to increase the associativity of the cache, which increases the complexity of the cache controller, is to decrease the burden on the user. Without associativity, the user has to make sure that the data elements that are being accessed are contiguous. Otherwise, the cache would thrash. Consider the sum of produces example below. If the x[] and y[] arrays start at the beginning of two different blocks in memory, then each instruction will thrash. First, x[i] is brought into the cache with index 0. Then, y[i] is brought in with the same index, forcing x[i] to be overwritten. If x[] is every used again, this would dramatically decrease the performance of the cache.

Take the same example as shown with two cache ways. Now, x[i] and y[i] each have their own location in the cache, and the thrashing is eliminated. The programmer does not have to worry about where the data elements ended up in their system because the associativity allows more flexibility.
Replacing a Set (LRU)

What happens in our 2-way cache when both lines of a set have valid data and a new value with the same index (i.e. line number) needs to be cached?

The cache controller uses a **Least Recently Used (LRU)** algorithm to decide which cache way line to overwrite when a cache miss occurs. With this algorithm, the most recently *accessed* data is always stays in the cache. Note that this may or may not be the "oldest" item in the cache, rather the most recently “used”. In a 2-way set-associative cache, this algorithm can be implemented with a bit per line. The LRU algorithm maximizes the effect of temporal locality, which caches depend upon to maximize performance.
L1 Data (L1D) Cache Summary

The L1D is a 2-way set-associative data cache. On the C671x devices, it is 4K bytes large with a 32-byte linesize. It gets larger on the C64x and C64x+ based devices.

<table>
<thead>
<tr>
<th>Device</th>
<th>Scheme</th>
<th>Size</th>
<th>Linesize</th>
<th>New Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>C62x/C67x</td>
<td>2-Way Set Assoc.</td>
<td>4K bytes</td>
<td>32 bytes</td>
<td>N/A</td>
</tr>
<tr>
<td>C64x</td>
<td>2-Way Set Assoc.</td>
<td>16K bytes</td>
<td>64 bytes</td>
<td>N/A</td>
</tr>
<tr>
<td>C64x+</td>
<td>2-Way Set Assoc.</td>
<td>C6455: 32K</td>
<td>64 bytes</td>
<td>Cache/RAM, Cache Freeze, Memory Protection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DM64xx: 80K</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- All L1D memories provide zero waitstate access
- Cache/RAM configuration and Cache Freeze work similar to L1P
- L1 caches are 'Read Allocate', thus only updated on memory read misses

The C64x+ based devices, similar to the L1P, supports configuration of the L1D as Cache or RAM, Cache Freeze, and Memory Protection.
L2 Memory

The Level 2 memory (L2) is a middle hierarchical layer that helps the cache controller keep the items that the CPU will need next closer to the L1 memories. It is larger than the L1D/L1P memories to help store larger arrays/functions and keep them closer to the CPU than external memory. It is a unified memory, meaning that it can store both code and data.
L2 Configuration

C671x L2 Configuration

The L2 memory is configurable to allow for a mix of RAM blocks and cache ways. The 64KB is divided into four chunks, each of which can either be RAM memory or a cache way. This allows the designer to set some on-chip memory aside for dedicated buffers, and to use the other memory as cache ways.

C671x – L2 Memory Configuration

- Four 16KB blocks – Configure each as cache or addressable RAM
- Each additional cache block provides another cache way
- L2 is unified memory – can hold program or data
- C6713 Still has 4 configurable 16KB RAM/cache blocks, the remaining 192KB is always RAM

The L2 can be changed during run time. So, a designer could choose to change a RAM block to cache or vice versa. Before making a switch from RAM to cache, the user should make sure the any information needed by the system that is currently in the RAM block is copied somewhere else. This copy can be done with the DMA to minimize the overhead on the CPU. Before switching a cache way to RAM, the cache should be free of any dirty data. Dirty data is data that has been written by the CPU but may not have been copied out to memory.
**L2 Memory**

**C64x L2 Configuration**

- **Configuration**
  - When cache is enabled, it’s always 4-Way
  - This differs from C671x

- **Linesize**
  - Linesize = 128 bytes
  - Same linesize as C671x

- **Performance**
  - L2 → L1P
    - 1-8 Cycles
  - L2 → L1D
    - L2 SRAM hit: 6 cycles
    - L2 Cache hit: 8 cycles
    - Pipelined: 2 cycles

**C64x+ L2 Configuration**

- **Configuration**
  - 2MB on C6455
  - When enabled, it’s always 4-Way (same as C64x)

- **Linesize**
  - Linesize = 128 bytes
  - Same linesize as C671x & C64x

- **Performance**
  - L2 → L1P
    - 1-8 Cycles
  - L2 → L1D
    - L2 SRAM hit: 12.5 cycles
    - L2 Cache hit: 14.5 cycles
    - Pipelined: 4 cycles
    - When required, minimize latency by using L1D RAM

*Using the Config Tool...*
C66x LL2 Configuration

C66x – LL2 Memory Configuration

- Local Level 2 (LL2)
- Configuration
  - When enabled, it's always 4-Way (same as C64x+)
- Linesize
  - Linesize = 128 bytes
  - Same linesize as C64x+
- Performance
  - L2 → L1P
    - ... tbd ...
  - L2 → L1D
    - ... tbd ...
    - When required, minimize latency by using L1D RAM

Using the Config Tool...

Configuring L2 using the Configuration Tool

The L2 can be configured at initialization using the configuration tool.

If cache is so great, why allow cache or RAM?
Why both RAM and Cache?

Why would a designer choose to configure the L1 or L2 memory as RAM instead of cache? Consider a system that uses the EDMA to transfer data from a serial port. If there is no internal memory, this data has to be written into external memory. Then, when the CPU accesses the data, it will be brought in to L2 (and/or L1) by the cache controller. Does this seem inefficient?

If L1 or L2 didn’t have addressable RAM?

- Requires external storage of peripheral data
- Both EDMA and CPU must tie up EMIF to store and retrieve data

If you use the DMA to read from on-chip peripherals – such as the McBSP – you might prefer to use part of the L2 memory as memory-mapped RAM. This setup allows you to store incoming data on-chip, rather than having to move it to off-chip, cache it on-chip, and then move it back off-chip to send it out to the external world.
The configurability of the L2 memory (and L1 on the C64x+) as RAM or cache allows designers to maximize the efficiency of their system.

**C6000 - Flexible & Efficient**

- Configure L2 (on all devices) or L1 (C64x+) as cache and/or mapped-RAM
- Allows peripheral data or critical code and data storage on-chip

**L2 Performance Summary**

**Cache Performance Summary**

<table>
<thead>
<tr>
<th>Device</th>
<th>L1P</th>
<th>L1D</th>
<th>L2 Performance</th>
</tr>
</thead>
</table>
| C62x/C67x| Zero Waitstate Cache | Zero Waitstate Cache | L2 ? L1P: 16 instr in 5 cycles  
L2 ? L1D: 32 bytes in 4 cycles |
| C64x     | Zero Waitstate Cache | Zero Waitstate Cache | L2 ? L1P: 8 instr in 1-8 cycles  
L2 ? L1D: 64 bytes in:  
L2 SRAM: 6 cycles  
L2 Cache: 8 cycles  
Pipelined: 2 cycles |
| C64x+ C674x | Zero Waitstate Cache/RAM | Zero Waitstate Cache/RAM | L2 ? L1P: 8 instr in 1-8 cycles  
L2 ? L1D: 64 bytes in:  
L2 SRAM: 12.5 cycles  
L2 Cache: 14.5 cycles  
Pipelined: 4 cycles |
Data Cache Coherency

One issue that can arise with caching architectures is called coherency. The basic idea behind coherency is that the information in the cache should be the same as the information that is stored at the memory address for that information. As long as the CPU is the only piece of the system that modifies information, and the system does not use self-modifying code, coherency will always be maintained. Ignoring the self-modifying code issue, is there anything else in the system that modifies memory?

Example Problem

Let's look at an example that will highlight coherency issues and provide some solutions.

![Coherency Example: Description](diagram)

- For this example, L2 is set up as cache
- Example's Data Flow:
  - EDMA fills RcvBuf
  - CPU reads RcvBuf, processes data, and writes to XmtBuf
  - EDMA moves data from XmtBuf (e.g. to a D/A converter)

In this example, the coherency between the L1, L2, and external memories is considered. This example only deals with data.

An important consideration in 'C6x11 based systems is the effect of the EDMA. The EDMA can modify (read/write) information. The CPU does not know about the EDMA modifying memory locations. The CPU and the DMA can be viewed as two co-processors (which is what they really are) that are aware of each other, but don't know exactly what the other is doing.

Look at the diagram below. This system is supposed to receive buffers from the EDMA, process them, and send them out via the EDMA. When the EDMA finishes receiving a buffer, it interrupts the CPU to transfer ownership of the buffer from the EDMA to the CPU.
In order to process the buffers, the CPU first has to read them. The first time the buffer is accessed, it is not in either of the caches, L1 or L2. When the buffer is read, the data is brought in to both of the caches. At this point, all three of the buffers (L1, L2, and External) are coherent.

When the CPU is finished processing the buffer, it writes the results to a transmit buffer. This buffer is located out in external memory. When the buffer is written, since it does not currently
reside in L1D, a write miss occurs. This write miss causes the transmit buffer to be written to the next lower level of memory, L2 in this case. The reason for this is that L1D does NOT allocate space for write misses. Usually DSPs do a lot more reading than they do writing, so the effect of this is to allow more read misses to live in cache.

The net effect is that the transmit buffer gets written to L2.

---

**Where Does the CPU Write To?**

- After processing, the CPU writes to XmtBuf
- Write misses to L1D are written directly to the next level of memory (L2)
- Thus, the write does **not** go directly to external memory
- **Cache line Allocated:**
  - L1D on **Read only**
  - L2 on **Read or Write**
Remember that the EDMA is going to be used to send the buffer out to the real world. So, where does it start reading the buffer from? That's right, external memory. Don't forget that caches do not have addresses. The EDMA requires an address for the source and destination of the transfer. The EDMA can't transfer from cache, so the buffer has to get from cache to external memory at the correct time.

Since the cached value which was written by the CPU is different from the value stored in external memory, the cache is said to be incoherent.

If coherency is not maintained (by sending the new cache values out to external memory), then the EDMA will send whatever is at the address that it was told to use. The best case is that this memory has been initialized with something that won't cause the system to break. The worst case is that the EDMA sends garbage data that may disrupt the rest of the system. Either way, the system is not doing what we wanted it to do.
Solution 1: Using Cache Flush & Clean

A solution to this problem is to tell the cache controller to send out anything that it has stored at the address of the transmit buffer. This can be done with a cache writeback operation. A cache writeback sends anything that is in cache out to its address in external memory. Does a writeback need to send all of the data? No, it only needs to send the information that has been modified by the CPU, which is referred to as dirty. In the case of the transmit buffer, all of the information was written by the CPU, so it is all dirty and it will all be sent to external memory by a writeback.

So, when the CPU is finished with the data, performing a writeback of the entire buffer will force the information out to its real address so that the EDMA can read it. Another way to think of a writeback is a copy of dirty data from cache to its memory location.

Solution 1: Flush & Clear the Cache

- When the CPU is finished with the data (and has written it to XmtBuf in L2), it can be sent to ext. memory with a cache writeback
- A writeback is a copy operation from cache to memory, writing back the modified (i.e. dirty) memory locations – all writebacks operate on full cache lines
- Use BIOS BCACHE to force a writeback:

  BIOS: BCACHE_wb (XmtBuf, BUFFSIZE, CACHE_NOWAIT);

Before looking at the next solution, there’s one other coherency issue...
Now that we know how to get the transmit buffers to their memory addresses to solve the coherency issue, let's consider another case on the read side. What happens if the EDMA writes new data to the receive buffer. The CPU needs to process this new data and send it out, just like before. However, this situation is different because the addresses for the receive buffer are already in the cache. So, when the CPU reads the buffer, it will read the cached values (i.e. the old values) and not the new values that the EDMA just wrote.

- EDMA writes a new RcvBuf buffer to ext. memory
- When the CPU reads RcvBuf a cache hit occurs since the buffer (with old data) is still valid in cache
- Thus, the CPU reads the old data instead of the new
In order to solve this problem, we need to force the CPU to read the external memory instead of the cache. This can be done with a cache invalidate. An invalidate invalidates all of the lines by setting the valid bit of each line of cache to 0 or false.

Another Coherency Solution

To get the new data, you must first invalidate the old data before trying to read the new data (clears cache line’s valid bits)

Again, cache operations (writeback, invalidate) operate on cache lines

BIOS BCACHE also provide an invalidate option:

```
BIOS: BCACHE_inv (RcvBuf, BUFFSIZE, CACHE_WAIT);
```

What about internal memory?

The C621x/C671x processors only have a writeback-invalidate operation on L2. They cannot do an invalidate by itself. A couple of things need to be considered before performing the cache writeback-invalidate. Since the writeback-invalidate performs a writeback of the data on L2, any modified or dirty data will be sent out to external memory. So, the writeback-invalidate must be done while the CPU owns the buffer. Otherwise, the old modified values could overwrite the new values from the EDMA. Also, a writeback-invalidate should only be performed after the CPU has finished modifying the buffer. If the writeback-invalidate is performed before the CPU is finished with the data, it will be brought back in, negating the effect of the writeback-invalidate.
**Solution 2: Use L2 Memory**

A second solution to the coherency issues is to let the device handle them for you. Start by linking the buffers into addressable L2 memory rather than external memory. The EDMA can then transfer in and out of these buffers without any coherency issues. What about coherency issues between L1 and L2? The cache controller handles all coherency issues between L1 and L2.

![Diagram showing Solution 2: Keep Buffers in L2](image)

- Configure some of L2 as RAM
- Locate buffers in this RAM space
- Coherency issues do **not** exist between L1D and L2

To summarize Cache Coherency...

This solution may be the simplest and best for the designer. It is a powerful solution, especially when considering that the EDMA could be transferring from another peripheral, the McBSP. In this case, it is best to have the EDMA transfer to on-chip buffers so that they don't have to be brought back in again by the cache controller as we discussed earlier. Add this to the fact that all coherency issues are taken care of for you, and this makes for a powerful, efficient solution.
Cache Coherency Summary

The tables below list the different situations that may cause coherency issues and their possible solutions:

### BCACHE Functions Summary

<table>
<thead>
<tr>
<th></th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Invalidate</td>
<td>BCACHE_inv(blockPtr, byteCnt, wait)</td>
</tr>
<tr>
<td></td>
<td>BCACHE_invL1pAll()</td>
</tr>
<tr>
<td>Cache Writeback</td>
<td>BCACHE_wb(blockPtr, byteCnt, wait)</td>
</tr>
<tr>
<td></td>
<td>BCACHE_wbAll()</td>
</tr>
<tr>
<td>Invalidate &amp; Writeback</td>
<td>BCACHE_wbInv(blockPtr, byteCnt, wait)</td>
</tr>
<tr>
<td></td>
<td>BCACHE_wbInvAll()</td>
</tr>
<tr>
<td>Sync waiting for Cache</td>
<td>BCACHE_wait()</td>
</tr>
</tbody>
</table>

- blockPtr: start address of range to be invalidated
- byteCnt: number of bytes to be invalidated
- Wait: 1 = wait until operation is completed

### Coherence Summary

**Internal (L1/L2) Cache Coherency is Maintained**
- Coherence between L1D and L2 is maintained by cache controller
- No BCACHE operations needed for data stored in L1D or L2 RAM
- L2 coherence operations implicitly operate upon L1, as well

**Simple Rules for Error Free Cache**
- **Before** the DSP begins reading a shared external INPUT buffer, it should first **BLOCK Invalidate** the buffer
- **After** the DSP finishes writing to a shared external OUTPUT buffer, it should initiate an L2 **BLOCK WRITEBACK**
**Cache Line Alignment**

**Problem:** How can I invalidate (or writeback) just the buffer?

- **In this case, you can’t**

**Definition:** False Addresses are ‘neighbor’ data in the cache line, but outside the buffer range

**Why Bad:** Writing data to buffer marks the line ‘dirty’, which will cause entire line to be written to external memory, thus:

- External neighbor memory could be overwritten with old data

---

### FIR Cache Align Example

```c
#define BUF 128
#pragma DATA_ALIGN (in, BUF)
short in[2][20*BUF];
```

---

// Definitions - BUFFERS, History/Data/Hole (+ some math !)
//---------------------------------------------------------------

// Declare processing buffer structure. For use with a C-coded FIR routine,
// we need a delay line on the receive buffer to keep track of the history of
// samples that are not processed, then copy them to the next incoming buffer
// in order to process them.
//
// Each set of Rcv buffers + the history buffer are
// declared separately in order to align each on a L2 line size boundary
// (128 bytes). ORDER-1 is an odd number, so this will create holes in
// memory. To be explicit, we have allocated space for the holes created by the
// 128-byte alignment. This has to be comprehended by the EDMA channel sorting.

#define L2_LINESIZE (128)
#define HIST_SIZE (ORDER-1)
#define DATA_SIZE (BUFFSIZE/2)

// HOLE_SIZE is based on buffer sizes and alignment. This calculation is done in
// shorts (16-bit) - hence the L2_LINESIZE/2. That is because the "hole" is def’d
// as a type int16_t. Effectively, the equation is 64 - [(HIST+DATA) & (63)].
// This is a poorman’s modulo calculation.
#define HOLE_SIZE (L2_LINESIZE/2-((HIST_SIZE+DATA_SIZE) & (L2_LINESIZE/2-1)))

// BIDX_RCV_SIZE used in channel sorting to bump RCV pointer from L to R samples
#define BIDX_RCV_SIZE (HIST_SIZE + DATA_SIZE + HOLE_SIZE)

// BIDX_XMT_SIZE used in channel sorting to bump XMT pointer from L ro R samples
// This is different than BIDX_RCV_SIZE because XMT has no history or holes
#define BIDX_XMT_SIZE (DATA_SIZE)
“Turn Off” the Cache (MAR)

As stated earlier in the chapter, the L1 cache cannot be turned-off. While this is true, alternatively a region of memory can be made non-cacheable. A memory access that must go all the way to the original memory location is called a long-distance access.

Using the Memory Attribute Registers (MAR), one can force the CPU to do a long-distance access to memory every time a read or write is performed. The L1 and/or L2 cache is not used for these long-distance accesses.

Why would you want to prevent some memory addresses from being cached? Often there are values found in off-chip, memory-mapped registers that must be read anew each time they are accessed. One example of this might be a system that references a hardware status register found in a field programmable gate array (FPGA). Another example where this might be useful is a FIFO out in external memory, where the same memory address is read repeatedly, but a different value is accessed for each read.

While MARs may also provide a solution to coherency issues, this is not a recommended solution because long-distance accesses can be extremely slow. If accesses infrequently, this decreased speed may not be an issue, but if used for real-time data access the decreased performance may keep the system from operating correctly anyway, coherency issues or not.
The Memory Attribute Registers allow the designer to turn cacheability on and off for a given address range. Each MAR controls the cacheability of 16MB of external memory.

### Memory Attribute Regs (MAR)

- Use MAR registers to enable/disable caching of external ranges
- Useful when external data is modified outside the scope of the CPU
- You can specify MAR values in Config Tool

#### C671x:
- 16 MARs
- 4 per CE space
- Each handles 16MB

#### C64x/C64x+:
- Each handles 16MB
- 256/224 MARs
- 16 per CE space
  (on current C64x, some are rsvd)

These registers can be used to control the caching of different ranges by setting the appropriate bit to 1 for cache enabled and 0 for cache disabled. These registers can also be setup using the configuration tool.

### Setting MARs in TCF (C67x)

<table>
<thead>
<tr>
<th>MAR 0</th>
<th>MAR 1</th>
<th>MAR 2</th>
<th>MAR 3</th>
<th>...</th>
<th>MAR 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>...</td>
<td>00000000</td>
</tr>
</tbody>
</table>

**MAR bit values:**
- 0 = Not cached
- 1 = Cached

Looking at the C64x+ GUI...
Data Cache Coherency

Setting MARs in TCF (C64x)

Configure MAR via GCONF (C6748)

If L1P cache is on, it caches everything
- On the C64x, if the MAR bits marked a range of memory as non-cacheable, L1P would not cache that memory
- On the C64x+, everything is cacheable by the L1P controller regardless of the state of the MAR bits
**Additional Memory/Cache Topics**

**'C64x Memory Banks**

The 'C64x also uses a memory banking scheme to organize L1. Each bank is 32 bits wide, containing four byte addresses. Eight banks are interleaved so that the addresses move from 1 bank to the next. The basic rule is that you can access each bank once per cycle, but if you try to access a bank twice in a given cycle you will encounter a memory bank stall. So, when creating arrays that you plan to access with parallel load instructions, you need to make sure that the arrays start in different banks. The DATA_MEM_BANK() pragma helps you create the arrays so that they start in different memory banks.

![C64x/C64x+ L1D Memory Banks Diagram]

- Only one access allowed per bank per cycle
- Use DATA_MEM_BANK to make sure that arrays that will be accessed in parallel start in different banks
  
  `#pragma DATA_MEM_BANK(var, 0 or 2 or 4 or 6)`
Sometimes variables need to be aligned to account for the way that memory is organized. The DATA_MEM_BANK is a specialized data align type #pragma that does exactly this.

### DATA_MEM_BANK Example

- Only one L1D access per bank per cycle
- Use DATA_MEM_BANK pragma to begin paired arrays in different banks
- *Note*: sequential data are *not* down a bank, instead they are along a horizontal line across banks, then onto the next horizontal line
- Only even banks (0, 2, 4, 6) can be specified

```c
#pragma DATA_MEM_BANK(a, 4);
short a[256];

#pragma DATA_MEM_BANK(x, 0);
short x[256];

for(i = 0; i < count ; i++) {
    sum += a[i] * x[i];
}
```

Unlike some of the other pragmas discussed in this chapter, the DATA_ALIGN pragma does not have to be used directly before the definition of the variable it aligns. Most users, though, prefer to keep them together to ease in code maintenance.
Cache Optimization

Here are some great ideas for how to optimize cache.

- Optimize for Level 1
- Multiple Ways and wider lines maximize efficiency
  - *we did this for you!*
- Main Goal - *maximize line reuse before eviction*
  - Algorithms can be optimized for cache
- “Touch Loops” can help with compulsory misses
- Up to 4 write misses can happen sequentially, but the next read or write will stall
- Be smart about data output by one function then read by another (touch it first)

Each one of these subjects deserves to be treated with enough material to fill a chapter in a book. In fact, a book has been written to cover these subjects.

Updated Cache Documentation

- **Cache Reference**
  - More comprehensive description of C6000 cache
  - Revised terminology for cache coherence operations

- **Cache User's Guide**
  - Cache Basics
  - Using C6000 Cache
  - Optimization for Cache Performance

SPRU609: C621x/C671x
SPRU610: C64x
SPRU871: C64x+
SPRUFK5: C674
SPRUGW0: C66x
SPRU656: C62x/C64x/C67
SPRU862: C64x+
SPRUG82: C674x
SPRUGY8: C66x
Cache Aware Linking

Goal
Re-arrange functions to reduce L1P conflict misses

How it works
CGT v7.0 contains a new cache layout tool (clt6x). It takes dynamic profile info to create a preferred function ordering linker command file that guides the placement of function subsections

More Info
http://processors.wiki.ti.com/index.php/Program_Cache_Layout

Procedure
1. Profile code for L1P cache misses - don’t solve a problem that doesn’t exist
2. Instrument your app by building with compiler option (--gen_profile_info)
3. Run instrumented app to generate profile data (.ppd)
4. Decode profile data file (.prf)
5. Generate WCG data (.csv) for each source file
6. Generate linker command file (.cmd file)
7. Re-build of the app with optimized function ordering

Cache Terminology Summary

Cache – General Terminology

- **Associativity**: The # of places a piece of data can map to inside the cache.
- **Coherence**: assuring that the most recent data gets written back from a cache when there is different data in the levels of memory
- **Dirty**: When an allocated cache line gets changed/updated by the CPU (*file)
- **Read-allocate cache**: only allocates space in the cache during a read miss. C64x+ L1 cache is read-allocate only.
- **Write-allocate cache**: only allocates space in the cache during a write miss.
- **Read-write-allocate cache**: allocates space in the cache for a read miss or a write miss. C64x+ L2 cache is read-write allocate.
- **Write-through cache**: updates to cache lines will go to ALL levels of memory such that a line is never “dirty” (less efficient than WB cache – more DDR xfrs).
- **Write-back cache**: updates occur only in the cache. The line is marked as “dirty” and if it is evicted, updates are pushed out to lower levels of memory. All C64x+ cache is write-back’.
Authoring a xDAIS/xDM Algorithm

Introduction

This chapter looks at algorithms from the inside out; how you write a xDAIS algorithm. It begins with the general description of xDAIS and how it is used, then examines the interface standard by focusing on the creation/usage/deletion of an algorithm and how its API deals with memory resource allocations.

Learning Objectives

Outline

◆ Introduction
  • What is xDAIS (and VISA)?
  • Software Objects
  • Master Thread Example
  • Intro to Codec Engine Framework (i.e. VISA)
◆ Algorithm Lifecycle
◆ Frameworks
◆ Algorithm Classes
◆ Making An Algorithm
◆ Appendix
Chapter Topics

Authoring a xDAIS/xDM Algorithm ..............................................................................................................13-1

Introduction ..................................................................................................................................................13-3
What is xDAIS (and VISA)? .....................................................................................................................13-3
Software Objects ........................................................................................................................................13-4
Master Thread Example...............................................................................................................................13-6
Intro to Codec Engine Framework (i.e. VISA).............................................................................................13-6

Algorithm Lifecycle.......................................................................................................................................13-9
Create...........................................................................................................................................................13-10
Process.........................................................................................................................................................13-14
Delete.........................................................................................................................................................13-16

Frameworks ..................................................................................................................................................13-17
Algorithm Classes ......................................................................................................................................13-19
xDM/VISA Classes ....................................................................................................................................13-19
Universal Class .........................................................................................................................................13-20
Design Your Own – Extending the Universal Class ..................................................................................13-21

Making an Algorithm..................................................................................................................................13-22
Rules of xDAIS ..........................................................................................................................................13-22
Using the Algorithm Wizard .....................................................................................................................13-24

(Optional) Algorithms on ARM, DSP and ARM+DSP .................................................................................13-28
Remote Procedure Calls ............................................................................................................................13-29
Step-by-Step Remote Procedure Call ........................................................................................................13-30

Appendix .....................................................................................................................................................13-33
Reference Info .............................................................................................................................................13-33
Extending xDM ..........................................................................................................................................13-34
(Optional) xDAIS Data Structures .............................................................................................................13-35
(Optional) Multi-Instance Ability ...............................................................................................................13-39
(Optional) xDAIS : Static vs Dynamic ......................................................................................................13-40
Introduction

What is xDAIS (and VISA)?

- Componentize algorithms for:
  - Plug-n-play *ease of use*
  - Single, *standardized interface* to use/learn
  - Enables use of common frameworks

- Express DSP Algorithm Interface Standard (xDAIS):
  - Similar to *C++ class* for algorithms
  - Provides a *time-tested*, real-time protocol

**Algo’s as System Plug-In**

- Input
- Output
- Memory
- Your Application
- Algo
- Algo
- Algo

**Call with VISA : Author with xDAIS**

- Input
- Output
- Memory
- Your Application
- Algo
- Algo
- Algo

**Acronyms:**
- **xDAIS** – set of functions algorithm author writes (xDM – Extensions to xDAIS)
- **VISA** – complimentary set of functions used by application programmer
Software Objects

Examples of software objects:
- C++ classes
- xDAIS (or xDM) algorithms

What does a software object contain?
- Thinking of C++ classes:
  - Attributes:
    - Class object
    - Creation (i.e. construction) parameters
  - Methods
    - Constructor
    - Destructor
    - "processing" method(s)

What does it mean to create an algorithm?

Comparing Objects: C++ / xDAIS

class algo{
public:
    // methods
    int method1(int param);
    int method2(int param);
    // attributes
    int attr1;
    int attr2;
}

typedef struct {
    // methods
    int (*method1) (int param);
    int (*method2) (int param);
    // attributes
    int attr1;
    int attr2;
} algo;

- xDAIS (and xDM) provide a C++-like object, implemented in C
- Because C does not support classes, structs are used
- Because structs do not support methods, function pointers are used

Like C++ classes, you must create an instance of an algorithm...
Comparing Methods: C++ / xDM

Create Instance: (C++ Constructor)

C++ algo::algo(algo_params params)
VISA VIDENC_create(VIDENC_params params)

Process:

C++ algo::myMethod(…)
VISA VIDENC_process(…)

Delete Instance: (C++ Destructor)

C++ algo::~algo()
VISA VIDENC_delete()

Note: With VISA, the framework (i.e. Codec Engine library) allocates resources on algorithm creation, as opposed to C++ constructors, which allocate their own resources.

Algorithm Creation

Traditionally algorithms have simply used resources without being granted them by a central source

Benefits of Central Resource Manager:

1. Avoid resource conflict during system integration
2. Facilitates resource sharing (i.e. scratch memory or DMA) between algorithms
3. Consistent error handling when dynamic allocations have insufficient resources
Master Thread Example

Intro to Codec Engine Framework (i.e. VISA)

Codec Engine Framework Benefits

- Multiple algorithm channels (instances)
- Dynamic (run-time) algorithm instantiation
- Plug-and-play for algorithms of the same class (inheritance)
- Sharing of memory and DMA channel resources
- Algorithm interoperability with any CE-based Framework
- Same API, no new learning curve for all algorithm users
- Provided by TI!

Many of these benefits are a direct result of the object-oriented structure of the codec engine.
VISA API (Application Programming Interface)

- Complexities of Signal Processing Layer (SPL) are abstracted into four functions:
  _create_ _delete_ _process_ _control_

- **Create**: creates an instance of an algo that is, it malloc's the required memory and initializes the algorithm

- **Process**: invokes the algorithm calls the algorithms processing function passing descriptors for in and out buffers

- **Control**: used to change algo settings algorithm developers can provide user controllable parameters

- **Delete**: deletes an instance of an algo opposite of “create”, this deletes the memory set aside for a specific instance of an algorithm

VISA – Eleven Classes

- Complexities of Signal Processing Layer (SPL) are abstracted into four functions:
  _create_ _delete_ _process_ _control_

- VISA = 4 processing domains:
  Video Imaging Speech Audio

- Separate API set for encode and decode thus, a total of 11 API classes:
  V VIDENC VIDDEC
  I IMGENC IMGDEC
  S SPHENC SPHDEC
  A AUDENC AUDDEC
  Other VIDANALYTICS VIDTRANSCODE
  Universal

Codec Engine: VISA API

- Complexities of Signal Processing Layer (SPL) are abstracted into four functions:
  _create_ _delete_ _process_ _control_

- VISA = 4 processing domains:
  Video Imaging Speech Audio

- Separate API set for encode and decode thus, a total of 11 API classes:
  Video, Imaging, Speech, Audio

- Universal (generic algorithm)

- TI’s CODEC engine (CE) provides abstraction between VISA and algorithms
- Only one application interface (VISA API) – it doesn’t matter if the algo runs on the ARM and/or DSP
- TI provides many encoder/decoder algorithms – as well as our 3rd parties
- Use your own IP (intellectual property) by creating your own Universal algo’s

Reducing dozens of functions to 4

Filling out the Master Thread ...
**Master Thread Key Activities**

```c
// Create Phase
idevfd = open("/dev/xxx", O_RDONLY);
oflfd = open("./fname", O_WRONLY);
ioctl(idev fd, CMD, &args);
myCE = Engine_open("vcr", myCEAttrs);
myVE = VIDENC_create(myCE, "videnc", params);
while( doRecordVideo == 1 ) {
    read(idevfd, &rd, sizeof(rd));
    VIDENC_process(myVE, ...);
    VIDENC_control(myVE, ...);
    write(oflfd, &wd, sizeof(wd));
}
close(idevfd);
close(oflfd);
VIDENC_delete(myVE);
Engine_close(myCE);
```

**Execute phase**
- Read/swap buffer with Input device
- Run algo with new buffer
- Optional: perform VISA algo ctrl
- Pass results to Output device

**Delete phase**
- Return IO devices back to OS
- Algo RAM back to heap
- Close VISA framework

Note: the above pseudo-code does not show double buffering, often essential in Realtime systems!

**VISA Function Details**

```c
Engine_Handle myCE;
AUDENC_Handle myAE;
AUDENC_Params params;
AUDENC_DynParams dynParams;
AUDENC_Status status;
CERuntime_init();
myCE = Engine_open("myEngine", NULL);
myAE = AUDENC_create (myCE, "aEncoder", &params);
stat = AUDENC_process(myAE, &inBuf, &OutBuf, &inArgs, &outArgs);
stat = AUDENC_control(myAE, XDM_GETSTATUS, &dynParams, &status);
AUDENC_delete(myAE);
Engine_close (myCE);
```

- **Engine** and **Codec** string names are declared during the engine config file
- The config file (.cfg) specifies which algorithm packages (i.e. libraries) should be built into your application

**Pick your algo’s using .CFG file**

```javascript
/* Specify your operating system (OS abstraction layer) */
var osal = xdc.useModule('ti.sdo.ce.osal.Global');
osal.runtimeEnv = osal.LINUX;

/* Specify which algo’s you want to build into your program */
var vidDec = xdc.useModule('ti.codecs.video.VIDENC');
var audDec = xdc.useModule('ti.codecs.audio.AUDENC');

/* Add the Codec Engine library module to your program */
var Engine = xdc.useModule('ti.sdo.ce.Engine');

/* Create engine named "myEngine" and add these algo’s to it */
var myEng = Engine.create("myEngine", [
    {name: "vEncoder", mod: vidDec, local: true},
    {name: "aEncoder", mod: audDec, local: true},
  ]);```

**Application/Engine Configuration File (.cfg javascript file)**
Algorithm Lifecycle

### Algorithm Instance Lifecycle

<table>
<thead>
<tr>
<th>Algorithm Lifecycle</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create (&quot;Constructor&quot;)</td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td>doFilter</td>
</tr>
<tr>
<td>Delete (&quot;Destructor&quot;)</td>
<td></td>
</tr>
</tbody>
</table>

- Codec Engine only uses the Dynamic features of xDAIS

---

### Algorithm Instance Lifecycle

<table>
<thead>
<tr>
<th>Algorithm Lifecycle</th>
<th>Dynamic</th>
</tr>
</thead>
</table>
| Create ("Constructor") | algNumAlloc
                 algAlloc
                 algInit |
| Process             | doFilter |
| Delete ("Destructor") | algFree |

- Codec Engine only uses the Dynamic features of xDAIS

To understand these "alg" functions, let's look at how they are used...
iAlg Functions Summary

- **Create Functions**
  - `algNumAlloc` - Tells application (i.e. CODEC engine) how many blocks of memory are required; it usually just returns a number
  - `algAlloc` - Describes properties of each required block of memory (size, alignment, location, scratch/persistent)
  - `algInit` - Algorithm is initialized with specified parameters and memory

- **Execute Functions**
  - `algActivate` - Prepare scratch memory for use; called prior to using algorithms process function (e.g. prep history for filter algo)
  - `algDeactivate` - Store scratch data to persistent memory subsequent to algo's process function
  - `algMoved` - Used if application relocates an algorithm's memory

- **Delete Function**
  - `algFree` - Algorithm returns descriptions of memory blocks it was given, so that the application can free them

Create

Instance Creation - start

1. Here’s the way I want you to perform…
   ```c
   Params = malloc(x);
   *Params= PARAMS;
   ```
Algorithm Parameters (Params)

- How can you adapt an algorithm to meet your needs?
  
  **Vendor specifies “params” structure to allow user to set creation parameters.**
  *These are commonly used by the algorithm to specify resource needs and/or they are used for initialization.*

- For example, what parameters might you need for a FIR filter?

  A filter called IFIR might have:

  ```c
  typedef struct IFIR_Params {
      Int size;          // size of params
      XDAS_Int16 firLen;
      XDAS_Int16 blockSize;
  } IFIR_Params;
  ```

Instance Creation - start

1. Here’s the way I want you to perform…
   ```c
   Params = malloc(x);
   *Params= PARAMS;
   ```

2. How many blocks of memory will you need to do this for me?

3. I’ll need “N” blocks of memory.
   (N may be based upon a params value)

4. I’ll make a place where you can tell me about your memory needs…
   ```c
   MemTab = malloc(5*N)
   ```
XDAIS Components: Memory Table

- What prevents an algorithm from “taking” too much (critical) memory?
  - Algorithms cannot allocate memory.
  - Each block of memory required by algorithm is detailed in a Memory Table (memtab), then allocated by the Application.

- MemTab:

  - **MemTab**
    - Size
    - Alignment
    - Space
    - Attributes
    - Base Addr
  
  - **Space**: Internal / External memory
  
  - **Attributes**: Scratch or Persistent memory (discussed later)

  - **Base**: Starting address for block of memory

XDAIS Components: Memory Table

- What prevents an algorithm from “taking” too much (critical) memory?
  - Algorithms cannot allocate memory.
  - Each block of memory required by algorithm is detailed in a Memory Table (memtab), then allocated by the Application.

- MemTab example:

  - **Application**
    - Based on the four memory details in MemTab,
    - Application allocates each memory block, and then
    - Provides base address to MemTab

  - **Algorithm**
    - Algo provides info for each block of memory it needs,
    - Except base address …
Algorithm Lifecycle

Instance Creation - finish

Codec Engine
(Application Framework)

5. Tell me about your memory requirements...

6. I'll enter my needs for each of the N blocks of memory, given these parameters, into the MemTab...

7. I'll go get/assign the memory you need...

for (i=0; i<=N; i++)
mem = malloc(size);

8. Prepare the new instance to run!

9. Initialize vars in my instance object using Params & Base’s

10. Delete MemTab

Now I can run the "processing" functions of the algo.

Example: Params & InstObj

1. Creation Params

typedef struct IFIR_Params {
    XDAS_Uint32 size;
    XDAS_Int16 firLen;
    XDAS_Int16 blockSize;
} IFIR_Params;

typedef struct IFIR_Obj {
    IFIR_Fxns *fxns;
    XDAS_Int16 firLen;
    XDAS_Int16 blockSize;
    XDAS_Int16 *blockPtr;
    XDAS_Int16 *historyPtr;
    type myGlobVar1;
    type myGlobVar2;
    type etc ...
} IFIR_Obj;

2. memTab

Now that the algo's created, how do we run it?
Algorithm Lifecycle

Process

Instance Execution

Codec Engine
(Application Framework)

1. Get ready to run. Scratch memory is yours now.

algActivate()

Algorithm

2. Prepare scratch memory, as required, from persistent memory

Scratch vs Persistent Memory

- **Scratch**: used by algorithm during execution only
- **Persistent**: used to store state information during instance lifespan

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Scratch</th>
<th>Per.A</th>
<th>Per.B</th>
<th>Per.C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total RAM</td>
<td>Scratch</td>
<td>Per.A</td>
<td>Scratch</td>
<td>Per.B</td>
</tr>
</tbody>
</table>

Okay for speed-optimized systems, but not where memory efficiency is a priority ...

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Scratch</th>
<th>Per.A</th>
<th>Per.B</th>
<th>Per.C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scratch A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Scratch B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Scratch C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total RAM</td>
<td>Scratch</td>
<td>Per.A</td>
<td>Per.B</td>
<td>Per.C</td>
</tr>
</tbody>
</table>

Usually a **Limited** Resource (e.g. Internal RAM)
Often an **Abundant** Resource (e.g. External RAM)

Looking at our filter example...
Example of Benefit of Scratch Memory

Example:
- Let's say we will process 1K block of data at a time
- For 32-tap filter, 32 samples must be saved from one process call to the next

<table>
<thead>
<tr>
<th># Chans</th>
<th>Overlay / Scratch</th>
<th>Use Scratch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>1032</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>1064</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10,000</td>
<td>1320</td>
</tr>
</tbody>
</table>

- Without using scratch (i.e. overlay) memory, 10 channels of our block filter would take ten times the memory
- If sharing the block between channels, memory usage drops considerably
  Only 32 RAM/channel persistent buffer to hold history vs. 1000 RAM/channel

Instance Execution

Codec Engine (Application Framework)

1. Get ready to run. Scratch memory is yours now.

2. Prepare scratch memory, as required, from persistent memory

3. Run the algorithm ...

4. Perform algorithm - freely using all memory resources assigned to algo

5. I need the scratch block back from you now...

6. Save scratch elements to persistent memory as desired

And how do we delete an algo instance?
Delete

Instance Deletion

If I no longer need the algorithm:
1. I’ll make a memTab again, or reuse the prior one
   \*memTab = malloc(5*N)
2. What memory resources were you assigned?
3. Re-fill memTab using \textit{algAlloc} and base addresses stored in the instance object
4. free all persistent memories recovered from algorithm

Codec Engine
(Application Framework)

MemTab

size
alignment
space
attrs
\*base

Algorithm

InstObj
Param1
Param2
...Base1
Base2
...

algFree()
If all algorithms must use these ‘create’ functions, couldn’t we simplify our application code?

**Framework Create Function**

<table>
<thead>
<tr>
<th>Create Functions</th>
<th>Codec Engine Framework (VISA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( algNumAlloc () )</td>
<td>( VIDENC_create() )</td>
</tr>
<tr>
<td>( algAlloc () )</td>
<td></td>
</tr>
<tr>
<td>( algInit () )</td>
<td></td>
</tr>
</tbody>
</table>

- These functions are common for all xDAIS/xDM algo’s
- One create function can instantiate any XDM algo
- History of algorithm creation functions from TI:
  - ALG create is simplistic example create function provided with the xDAIS library
  - ALGREF library provided in Reference Frameworks
  - DSKT2 library is used by the Codec Engine and Bridge Frameworks
  - Codec Engine (CE) provides create functions defined in xDM (or xDM-like) algos
CODEC engine is provided by TI
You need only be concerned with VISA or xDM
Algorithm Classes

xDAIS Limitations

- xDAIS defines methods for managing algo heap memory: algCreate, algDelete, algMoved
- xDAIS also defines methods for preparation/preservation of scratch memory: algActivate, algDeactivate
- Does not define the API, args, return type of the processing method
- Does not define the commands or structures of the control method
- Does not define creation or control structures

- Reason: xDAIS did not want to stifle options of algo author
- and Yields unlimited number of potential algo interfaces

- For DaVinci technology, defining the API for key media types would greatly improve
  - Usability
  - Modifiability
  - System design

- As such, the digital media extensions for xDAIS “xDAIS-DM” or “xDM” has been created to address the above concerns in DaVinci technology
- Reduces unlimited possibilities to 4 encoder/decoder sets!

xDAIS Classes

Eleven xDM Classes

- Video
  - VIDENC (encode)
  - VIDDEC (decode)
  - VIDANALYTICS (analysis)
  - VIDTRANSCODE (transcode)
- Imaging
  - IMGENC (encode)
  - IMGDEC (decode)
- Speech
  - SPHEN (encode)
  - SPHDEC (decode)
- Audio
  - AUDENC (encode)
  - AUDDEC (decode)
- Universal (custom algorithm)

- Create your own VISA compliant algorithm by inheriting the Universal class
- Then, use your algorithm with the Codec Engine, just like any other xDM algo
Frameworks

**xDM Benefits**

- **Enable plug + play** ability for multimedia codecs across implementations/vendors/systems
- Uniform across domains...video, imaging, audio, speech
- Flexibility - allows extension for custom / vendor-specific functionality
- **Low overhead**
  - Insulate application from component-level changes
  - Hardware changes should not impact software (EDMA2.0 to 3.0,...)
  - PnP ...enable ease of replacement for versions, vendors
- Framework Agnostic
  - Integrate component into any framework
- Enable early and parallel development by *publishing* the API: create code faster
  - System level development in parallel to component level development
  - Reduce integration time for system developers
- Published and Stable API
  - TI, 3rd Parties, and Customers
  - Support Backward Compatibility

**Universal Class**

**Universal Algorithm : Methods**

```c
UNIVERSAL_create ( myCE, "aEncoder", &IUNIVERSAL_Params );
UNIVERSAL_process ( IUNIVERSAL_Handle handle,
  XDM_BufDesc *inBufs,
  XDM_BufDesc *outBufs,
  XDM_BufDesc *inOutBufs,
  IUNIVERSAL_InArgs *inArgs,
  IUNIVERSAL_OutArgs *outArgs );
UNIVERSAL_control ( IUNIVERSAL_Handle handle,
  IUNIVERSAL_Cmd id,
  IUNIVERSAL_DynamicParams *params,
  IUNIVERSAL_Status *status );
UNIVERSAL_delete ( IUNIVERSAL_Handle handle );
```

- **Create each of the required data structures:**
  - Params, InArgs, OutArgs, DynParams, Status
- **Structure names must begin with "I" and "my algo's name"**
  (which you can read as "interface" to "my algorithm")
- **Your algo's structures must inherit the IUNIVERSAL datatypes**
### Universal Algorithm : Data

```c
typedef struct IUNIVERSAL_Params {
    XDAS_Int32 size;
} IUNIVERSAL_Params;
```

// ========================================================

```c
typedef struct IUNIVERSAL_OutArgs {
    XDAS_Int32 size;
    XDAS_Int32 extendedError;
} IUNIVERSAL_OutArgs;
```

- Universal interface defined in xDM part of xDAIS spec
  `/xdais_6_23/packages/ti/xdais/dm/iuniversal.h`
  Which inherits:
  `/xdais_6_23/packages/ti/xdais/dm/xdm.h`
  Which then inherits:
  `/xdais_6_23/packages/ti/xdais/ialg.h`

### Design Your Own – Extending the Universal Class

#### Creating a Universal Algorithm : Data

```c
typedef struct IMYALG_Params {
    IUNIVERSAL_Params base; // IUNIVERSAL_Params.size
    XDAS_Int32 param1;
    XDAS_Int32 param2;
} IMYALG_Params;
```

// ========================================================

```c
typedef struct IMYALG_OutArgs {
    IUNIVERSAL_OutArgs base; // IUNIVERSAL_OutArgs.size
    // IUNIVERSAL_OutArgs.extendedError
    XDAS_Int32 outArgs1;
} IMYALG_OutArgs;
```

- Create each of the required data structures:
  Params, InArgs, OutArgs, DynParams, Status
- Structure names must begin with “I” and “my algo’s name”
  (which you can read as “interface” to “my algorithm”)
- Your algo’s structures must inherit the IUNIVERSAL datatypes
Making an Algorithm

Rules of xDAIS

**Application / Component Advantages**

Dividing software between components and system integration provides optimal reuse partitioning, allowing:

- **System Integrator (SI):** to have full control of *system resources*.
- **Algorithm Author:** to write components that can be used in any kind of system.

**What are “system resources”?**

- **CPU Cycles**
- **RAM (internal, external) : Data Space**
- **DMA hardware**
  - Physical channels
  - PaRAMs
  - TCCs

*How does the system integrator manage the usage of these resources?*

**Resource Management : CPU Loading**

- All xDAIS algorithms **run only when called**, so no cycles are taken by algos without being first called by SI (application) code.
- Algos **do not define their own priority**, thus SI’s can give each algo any priority desired – usually by calling it from a BIOS task (TSK).
- xDAIS algos are required to **publish their cycle loading** in their documentation, so SI’s know the load to expect from them.
- Algo documentation also must define the worst case latency the algo might impose on the system.
Resource Management: RAM Allocation

- **Algos never ‘take’ memory** directly
  - Algos tell system its needs (\texttt{algNumAlloc()}, \texttt{algAlloc()})
  - SI determines what memory to give/lend to algo (\texttt{MEM_alloc()})
  - SI tells algo what memories it may use (\texttt{algInit()})
- **Algos may request internal or external RAM, but must function with either**
  - Allows SI more control of system resources
  - SI should note algo cycle performance can/will be affected
- **Algo authors can request memory as ‘scratch’ or ‘persistent’**
  - \textbf{Persistent}: ownership of resource must persist during life of algo
  - \textbf{Scratch}: ownership or resource required only when algo is running

Resource Management: Scratch Memory

- **SI can assign a permanent resource to a Scratch request**
  - Easy - requires no management of sharing of temporary/scratch resources
  - Requires more memory in total to satisfy numerous concurrent algos
- **SI must assure that each scratch is only lent to one algo at a time** (\texttt{algActivate()}, \texttt{algDeactivate()})
- **No preemption amongst algos sharing a common scratch is permitted**
  - Best: share scratch only between equal priority threads – preemption is implicitly impossible
  - \textit{Tip: limit number of thread priorities} used to save on number of scratch pools required
  - Other scratch sharing methods possible, but this is method used by C/E
- **Scratch management can yield great benefits**
  - More usage of highly prized internal RAM
  - Smaller total RAM budget
  - Reduced cost, size, and power when less RAM is specified
Using the Algorithm Wizard

Files created ...
Files Created by genCodecPkg

Example of code created in mixer.c ...

Code Created: Functions

```c
/*
 * ======== MIXER_TTO_IMIXER ========
 * This structure defines TTO's implementation of the IUNIVERSAL
 * interface for the MIXER_TTO module.
 */
IUNIVERSAL_Fxns MIXER_TTO_IMIXER = {
    {IALGFXNS},
    MIXER_TTO_process,
    MIXER_TTO_control,
};
```

- The required iAlg functions (as discussed)
- plus functions defined by iUniversal class
- _process()
- _control()
Making an Algorithm

**Code Created: algAlloc()**

```c
Int MIXER_TTO_alloc( const IALG_Params *algParams, 
                     IALG_Fxns **pf, IALG_MemRec memTab[])
{
   /* Request memory for my object */
   memTab[0].size      = sizeof(MIXER_TTO_Obj);
   memTab[0].alignment = 0;
   memTab[0].space     = IALG_EXTERNAL;
   memTab[0].attrs     = IALG_PERSIST;

   return (1);
}
```

**Code Created: algInit**

```c
Int MIXER_TTO_initObj( IALG_Handle handle, const IALG_MemRec memTab[], 
                       IALG_Handle parent, const IALG_Params *algParams )
{
   const IMIXER_Params *params = (IMIXER_Params *)algParams;
   
   /* Typically, your algorithm will store instance-specific details 
      in the object handle.  If you want to do this, uncomment the 
      following line and the 'obj' var will point at your instance object. 
   */
   //    MIXER_TTO_Obj *obj = (MIXER_TTO_Obj *)handle;

   
   /*
   * If no creation params were provided, use our algorithm-specific ones.
   * Note that these default values _should_ be documented in your algorithm
   * documentation so users know what to expect.
   */
   if ( params == NULL ) {
      params = &IMIXER_PARAMS;
   }

   /* Store any instance-specific details here, using the 'obj' var above */
   return (IALG_EOK);
}
```
Making an Algorithm

**Code Created : process()**

```c
XDAS_Int32 MIXER_TTO_process ( IUNIVERSAL_Handle h,
XDML_BufDesc *inBufs, XDML_BufDesc *outBufs,
XDML_BufDesc *inOutBufs,
IUNIVERSAL_InArgs *universalInArgs,
IUNIVERSAL_OutArgs *universalOutArgs)
{
    XDAS_Int32 numInBytes, i;
    XDAS_Int16 *pIn0, *pIn1, *pOut, gain0, gain1;

    /* Local casted variables to ease operating on our extended */
    IMIXER_InArgs *inArgs = (IMIXER_InArgs *)universalInArgs;
    IMIXER_OutArgs *outArgs = (IMIXER_OutArgs *)universalOutArgs;

    /* Note that the rest of this function will be algorithm-specific */
    /* the initial generated implementation, this process() function */
    /* copies the first inBuf to the first outBuf. But you should */
    /* this to suit your algorithm's needs. */

    /* report how we accessed the input buffers */
    /* report how we accessed the output buffer */
    return (IUNIVERSAL_EOK);
}
```
You can run algorithms on DSP-only processors in the same manner shown above for ARM-only systems. The only thing that would change would be the use of DSP/BIOS device drivers.
Remote Procedure Calls

Local Call of Algo From Application

Remote Procedure Call “RPC”

- The CODEC engine abstracts remote calls
- Stub functions marshall (i.e. gather together) the required arguments
- Skeletons unpack args, call the algo on the remote processor
TI’s ARM+DSP Software Framework Benefits

Application Code
- No changes for local vs remote algo
- Serves as ‘master thread’
- Controls all other components in system

IO Drivers
- No change to peripherals
- No change to application code
- Drivers accessible to Linux community
- Data does not pass through app on way to algo – no extra layer of overhead
- Buffers are in shared memory, equally accessible to DSP

Algorithm / DSP Task
- No change to algo code to run on DSP
- No change to algo code
- DSP Task is a pure ‘data transducer’:
  - no direct control over peripherals
  - ‘slave’ to app code control
  - not the ‘master’ of the application
- Algo inside TSK to provide priority, context
- Algo can use ACPY, DMAN to bring buffer data from shared mem to local RAM

Interprocessor Communication
- CODEC engine abstracts all IPC details
- App/algo unaware of location of algo
- Infrastructure provided by TI for DaVinci technology

Step-by-Step Remote Procedure Call

**CODEC Engine: CERuntime_init()**

```
ARM (w Linux)   User Space
idevfd = open(...);
ofilefd = open(...);
ioctl(...);
CERuntime_init();
myCE = Engine_open(...);
myVE = VIDENC_create(...);

while( doSvc ){ // execute
    read(idevfd, ...);
    VIDENC_control(myVE, ...);
    VIDENC_process(myVE, ...);
    write(ofilefd, ...);
}
close(idevfd);
close(ofilefd);
VIDENC_delete(myVE);
Engine_close(myCE);
```

CERuntime_init():
- Create-phase activity
- Creates the CODEC engine thread
- Only needs to be done once in a system

13 - 30 DaVinci/OMAP System Design Workshop - Authoring a xDAIS/xDM Algorithm
CODEC Engine: Engine_open()

**ARM (w/ Linux)**
- User Space
  - devfd = open(...); // create
  - ofld = open(...);
  - ioctl(...);
  - CERuntime_init();
  - myCE = Engine_open(...);
  - myVE = VIDENC_create(...);

**Kernel Space**
- Input Driver
- Output Driver
- I Buf
- O Buf
- CE RMS

**DSP = BIOS**
- MEM_free
- MEM_alloc
- algInit
- algNumAlloc
- algFree

**Engine_open()**:
- Downloads image to DSP
- Releases DSP from reset
- DSP image initialization creates CE RMS

CODEC Engine: VIDENC_create()

**ARM (w/ Linux)**
- User Space
  - devfd = open(...); // create
  - ofld = open(...);
  - ioctl(...);
  - CERuntime_init();
  - myCE = Engine_open(...);
  - myVE = VIDENC_create(...);

**Kernel Space**
- Input Driver
- Output Driver
- I Buf
- O Buf
- CE RMS

**DSP = BIOS**
- MEM_free
- MEM_alloc
- algInit
- algNumAlloc
- algFree

**VIDENC_create()**:
- Signals CE RMS to create algo instance
- CE RMS creates TSK as algo’s context
- Skeleton unpacks args from IPC

CODEC Engine: VIDENC_control()

**ARM (w/ Linux)**
- User Space
  - devfd = open(...); // create
  - ofld = open(...);
  - ioctl(...);
  - CERuntime_init();
  - myCE = Engine_open(...);
  - myVE = VIDENC_create(...);

**Kernel Space**
- Input Driver
- Output Driver
- I Buf
- O Buf
- CE RMS

**DSP = BIOS**
- MEM_free
- MEM_alloc
- algInit
- algNumAlloc
- algFree

**VIDENC_control()**:
- Signals myVE TSK via RPC
- myVE TSK calls DSP VIDENC_control
CODEC Engine: VIDENC_process()

- RPC to myVE TSK: context for process()
- Drivers give buffers to algo via Shared Mem
- App is signaling center; not data buf owner

CODEC Engine: VIDENC_delete()

- Signals CE RMS to delete algo instance
- CE RMS also deletes algo TSK

CODEC Engine: Engine_close()

- Places DSP back in reset
Appendix

Reference Info

References

- Codec Engine Algorithm Creator User’s Guide
  SPRUED6  Texas Instruments

- Codec Engine Server Integrator’s Guide
  SPRUED5  Texas Instruments

- x dctools_1_21/doc directory in DVEVM 1.1 software
  Documentation on XDC Configuration Kit and BOM

- Using adapters to run xDAIS algorithms in the Codec Engine
  SPRAAE7  Texas Instruments

Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>Codec Engine</td>
<td>DaVinci framework for instantiating and using remote or local codecs</td>
</tr>
<tr>
<td>DMAN</td>
<td>Dma MANager module. Manages DMA resource allocation</td>
</tr>
<tr>
<td>DSKT2</td>
<td>Dsp SocKeT module, rev. 2. Manages DSP memory allocation</td>
</tr>
<tr>
<td>DSP Link</td>
<td>Physical Transport Layer for Inter-processor Communication</td>
</tr>
<tr>
<td>Engine</td>
<td>CE framework layer for managing local and remote function calls</td>
</tr>
<tr>
<td>EPSI API</td>
<td>Easy Peripheral Software Interface API. Interface to system drivers.</td>
</tr>
<tr>
<td>OSAL</td>
<td>Operating System Abstraction Layer</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>Server</td>
<td>Remote Thread that Services Create/Delete RPC’s from the Engine</td>
</tr>
<tr>
<td>Skeleton</td>
<td>Remote Thread that Services Process/Control RPC’s for Codecs</td>
</tr>
<tr>
<td>Stub</td>
<td>Function that Marshalls RPC Arguments for Transport over DSP Link</td>
</tr>
<tr>
<td>VISA API</td>
<td>Functions to interface to xDM-compliant codecs using CE framework</td>
</tr>
<tr>
<td>xDAIS</td>
<td>eXpress DSP Algorithm Interface Standard. Used to instantiate algos</td>
</tr>
<tr>
<td>xDM</td>
<td>Interface that extends xDAIS, adding process and control functionality</td>
</tr>
</tbody>
</table>
Extending xDM

**Easily Switch xDM Components**

- All audio class decoders (eg: MP3 & AAC) provide the identical API
- Plug and Play: App using the IAUDDEC_Structures can call all audio decoders
- Any algorithm specific arguments must be set to default values internally by the vendor (insulating the application from need to specify these parameters)
- Specific functionality can be invoked by the app using extended data structures
- To summarize:
  - Most authors can use the default settings of the extended features provided by vendors
  - "Power users" can (optionally) obtain further tuning via an algos extended structures

```
Extending xDM – AAC DynamicParams Ex.

typedef struct IAUDDEC_DynamicParams {
    XDAS_Int32 size; /* size of this structure */
    XDAS_Int32 outputFormat; /* To set interleaved/Block format. */
} IAUDDEC_DynamicParams;

typedef struct IAACDEC_DynamicParams {
    IAUDDEC_DynamicParams auddec_dynamicparams;
    Int DownSampleSbr;
} IAACDEC_DynamicParams;

AAC Control function code – Using the extended structure

XDAS_Int32 AACDEC_T1_control(IAAUDDEC_Handle AAChandle, IAUDDEC_Cmd id, IAUDDEC_DynamicParams *params, IAUDDEC_Status *sPtr)
{
    IAACDEC_DynamicParams *dyparams = (IAACDEC_DynamicParams *)params;
    ...
    case IAACDEC_SETPARAMS:
        if(sizeof(IAACDEC_DynamicParams)==dyparams->auddec_dynamicparams.size)
            handle->downsamplerSBR=dyparams->DownSampleSbr;
        else
            handle->downsamplerSBR=0;

Comparing AAC and MP3...
```
**AAC and MP3 Extended Data Structures**

```c
typedef struct IAACDEC_Params {
    IAUDDEC_Params auddec_params;
} IAACDEC_Params;

typedef struct IAAACDEC_DynamicParams {
    IAUDDEC_DynamicParams auddec_dynamicparams;
    Int DownSampleSbr; // AAC specific
} IAAACDEC_DynamicParams;

typedef struct IAACDEC_InArgs {
    IAUDDEC_InArgs auddec_inArgs;
} IAACDEC_InArgs;

typedef struct IAACDEC_OutArgs{
    IAUDDEC_OutArgs auddec_outArgs;
} IAACDEC_OutArgs;

typedef struct IMP3DEC_Params {
    IAUDDEC_Params auddec_params;
} IMP3DEC_Params;

typedef struct IMP3DEC_DynamicParams {
    IAUDDEC_DynamicParams auddec_dynamicparams;
} IMP3DEC_DynamicParams;

typedef struct IMP3DEC_InArgs {
    IAUDDEC_InArgs auddec_inArgs;
    XDAS_Int32 offset;
} IMP3DEC_InArgs;

typedef struct IMP3DEC_OutArgs{
    IAUDDEC_OutArgs auddec_outArgs;
    XDAS_Int32 layer; // MP3 specific layer info
    XDAS_Int32 crcErrCnt;
} IMP3DEC_OutArgs;
```

---

**(Optional) xDAIS Data Structures**

**The Param Structure**

**Purpose**: To allow the application to specify to the algorithm the desired modes for any options the algorithm allows, e.g. size of arrays, length of buffers, Q of filter, etc…

- `sizeof()`: Defined by **Algorithm** (in header file)
- `filterType`, `filterOrder`, `bufferSize`: Allocated by **Application**
- `...`: Written to by **Application**
- `...`: Read from by **Algorithm**
### Param Structures Defined in IMOD.H

// IFIR_Params - structure defines instance creation parameters

typedef struct IFIR_Params {
    Int size;          /* 1st field of all params structures */
    XDAS_Int16 firLen;
    XDAS_Int16 blockSize;
    XDAS_Int16 * coeffPtr;
} IFIR_Params;

// IFIR_Status - structure defines R/W params on instance

typedef struct IFIR_Status {
    Int size;          /* 1st field of all status structures */
    XDAS_Int16 blockSize;
    XDAS_Int16 * coeffPtr;
} IFIR_Status;

### IFIR_Params : IFIR.C

```c
#include <std.h>
#include "ifir.h"

IFIR_Params IFIR_PARAMS = {
    sizeof(IFIR_Params),
    32,
    1024,
    0,
};
```

- User may replace provided IFIR.C defaults with their preferred defaults
- After defaults are set, Params can be modified for instance specific behavior

```c
#include "ifir.h"

IFIR_Params IFIR_params;
IFIR_params = IFIR_PARAMS;
IFIR_params.firLen = 64;  // Override length parameter
IFIR_params.blockSize = 1000;  // Override block size parameter
```

### The MemTab Structure

**Purpose**: Interface where the algorithm can define its memory needs and the application can specify the base addresses of each block of memory granted to the algorithm.

<table>
<thead>
<tr>
<th>Memory Need</th>
<th>Defined by</th>
<th>Allocated by</th>
<th>Written to by</th>
<th>Read from by</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>IALG Spec &amp; Algorithm</td>
<td>Application</td>
<td>Algorithm (4/5) &amp; Application (base addr)</td>
<td>Application (4/5) &amp; Algorithm (base addr)</td>
</tr>
<tr>
<td>alignment</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>space</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>attrs + base</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Defined by: IALG Spec & Algorithm (rtn value of algNumAlloc)
- Allocated by: Application 5*algNumAlloc()
- Written to by: Algorithm (4/5) & Application (base addr)
- Read from by: Application (4/5) & Algorithm (base addr)
**The Instance Object Structure**

**Purpose:** To allow the application to specify to the algorithm the desired modes for any options the algorithm allows, eg: size of arrays, length of buffers, Q of filter, etc…

- *fxns
- filterLen
- blockSize
- *coeffs
- *workBuf
- ...

**Defined by:** Algorithm

**Allocated by:** Application

- via memRec() description

**Written to by:** Algorithm

**Read from by:** Algorithm

(private structure!)

---

**The vTab Concept and Usage**

```c
#include <ialg.h>
typedef struct IFIR_Fxns {
    IALG_Fxns ialg;  /* IFIR extends IALG */
    Void (*filter)(IFIR_Handle handle, XDAS_Int8 in[], XDAS_Int8 out[]);
} IFIR_Fxns;
```

```
typedef struct IALG_Fxns {
    Void *implementationId;
    Void (*algActivate)(...);
    Int     (*algAlloc)(...);
    Int     (*algControl)(...);
    Void   (*algDeactivate)(...);
    Int     (*algFree)(...);
    Int     (*algInit)(...);
    Void   (*algMoved)(...);
    Int     (*algNumAlloc)(...);
} IALG_Fxns;
```

**vTab Structure**

```c
typedef struct IALG_Fxns {
    Void *implementationId;
    Void (*algActivate)(...);
    Int     (*algAlloc)(...);
    Int     (*algControl)(...);
    Void   (*algDeactivate)(...);
    Int     (*algFree)(...);
    Int     (*algInit)(...);
    Void   (*algMoved)(...);
    Int     (*algNumAlloc)(...);
} IALG_Fxns;
```
Pragmas - For Linker Control of Code Sections

```c
#pragma CODE_SECTION(FIR_TTO_activate, " .text:algActivate")
#pragma CODE_SECTION(FIR_TTO_alloc, " .text:algAlloc")
#pragma CODE_SECTION(FIR_TTO_control, " .text:algControl")
#pragma CODE_SECTION(FIR_TTO_deactivate," .text:algDeactivate")
#pragma CODE_SECTION(FIR_TTO_free, " .text:algFree")
#pragma CODE_SECTION(FIR_TTO_initObj, " .text:algInit")
#pragma CODE_SECTION(FIR_TTO_moved, " .text:algMoved")
#pragma CODE_SECTION(FIR_TTO_numAlloc, " .text:algNumAlloc")
#pragma CODE_SECTION(FIR_TTO_filter, " .text:filter")
```

Linker Control of Code Sections

- Users can define, with any degree of specificity, where particular algorithmic components will be placed in memory

<table>
<thead>
<tr>
<th>Component</th>
<th>Memory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:algActivate</td>
<td>IRAM</td>
</tr>
<tr>
<td>.text:algDeactivate</td>
<td>IRAM</td>
</tr>
<tr>
<td>.text:filter</td>
<td>IRAM</td>
</tr>
<tr>
<td>.text</td>
<td>SDRAM</td>
</tr>
</tbody>
</table>

- Components not used may be discarded via the "NOLOAD" option

<table>
<thead>
<tr>
<th>Component</th>
<th>Memory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:algActivate</td>
<td>IRAM</td>
</tr>
<tr>
<td>.text:algDeactivate</td>
<td>IRAM</td>
</tr>
<tr>
<td>.text:filter</td>
<td>IRAM</td>
</tr>
<tr>
<td>.text</td>
<td>SDRAM, type = NOLOAD</td>
</tr>
<tr>
<td>.text:algAlloc</td>
<td>SDRAM, type = NOLOAD</td>
</tr>
<tr>
<td>.text:algControl</td>
<td>SDRAM, type = NOLOAD</td>
</tr>
<tr>
<td>.text:algFree</td>
<td>SDRAM, type = NOLOAD</td>
</tr>
<tr>
<td>.text:algNumAlloc</td>
<td>SDRAM, type = NOLOAD</td>
</tr>
<tr>
<td>.text</td>
<td>SDRAM</td>
</tr>
</tbody>
</table>
(Optional) Multi-Instance Ability

**XDAIS Instance**

![Diagram of XDAIS Instance]

**Multiple Instances of an Algorithm**

![Diagram of Multiple Instances of an Algorithm]

Allocate, Activate as many instances as desired

Uniquely named handles allow control of individual instances of the same algorithm

All instance objects point to the same vtab

Coefficient array can be shared

Scratch can be separate or common as desired
Appendix

(Optional) xDAIS : Static vs Dynamic

<table>
<thead>
<tr>
<th>Algorithm Lifecycle</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>SINE_init</td>
<td>algNumAlloc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>algAlloc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>algInit (aka sineInit)</td>
</tr>
<tr>
<td>Process</td>
<td>SINE_value</td>
<td>SINE_value</td>
</tr>
<tr>
<td></td>
<td>SINE_blockFill</td>
<td>SINE_blockFill</td>
</tr>
<tr>
<td>Delete</td>
<td>- none -</td>
<td>algFree</td>
</tr>
</tbody>
</table>

- Static usage requires programmer to read algo datasheet and assign memory manually.
- Codec Engine only uses the Dynamic features of xDAIS.

To understand these "alg" functions, let's look at how they are used ...

### Dynamic (top) vs Static (bottom)

1. \( n = \text{fxns} \rightarrow \text{alg}.\text{algNumAlloc}(); \) //Determine number of buffers required
   \( \text{memTab} = (\text{IALG}_\text{MemRec} *)\text{malloc}(n*\text{sizeof}((\text{IALG}_\text{MemRec} )); \) //Build the memTab
   \( n = \text{fxns} \rightarrow \text{alg}.\text{algAlloc}((\text{IALG}_\text{Params} *)\text{params}, \&\text{fxnsPtr}, \text{memTab}); \) //Inquire buffer needs from alg

2. for \( (i = 0; i < n; i++) \) {
   \( \text{memTab[i].base} = (\text{Void} *)\text{memalign}((\text{memTab[i].alignment}, \text{memTab[i].size}); \}

3. \( \text{alg} = (\text{IALG}\_\text{Handle})\text{memTab[0].base}; \)
   \( \text{alg} \rightarrow \text{fxns} = \&\text{fxns} \rightarrow \text{alg}; \)
   //Set up handle and *fxns pointer

4. \( \text{fxns} \rightarrow \text{alg}.\text{algInit}(\text{alg}, \text{memTab}, \text{NULL}, (\text{IALG}_\text{Params} *)\text{params}); \) // initialize instance object

1. \( \text{IALG}_\text{MemRec} \text{memTab[1]}; \)
   // Create table of memory requirements
   // Reserve memory for instance object

2. \( \text{memTab[0].base} = \text{buffer0}; \)
   // with 1st element pointing to object itself

3. \( \text{ISINE}\_\text{Handle sineHandle}; \)
   \( \text{sineHandle} = \text{memTab[0].base}; \)
   \( \text{sineHandle} \rightarrow \text{fxns} = \&\text{SINE}_\text{TTO}_\text{ISINE}; \)
   // Create handle to InstObj
   // Setup handle to InstObj
   // Set pointer to algo functions

4. \( \text{sineHandle} \rightarrow \text{fxns} \rightarrow \text{alg}.\text{algInit}((\text{IALG}\_\text{Handle})\text{sineHandle}, \text{memTab}, \text{NULL}, (\text{IALG}_\text{Params} *)\&\text{sineParams}); \)
Optimized Code and Interrupts

Introduction

Typically, DSP systems compute algorithms very quickly. But sometimes, this system might be controlled by events outside the DSP system itself. Because of these outside events, it is important to write code that can be interrupted, and can handle the system’s demands without corrupting the DSP algorithms.

Outline

- Interrupts Overview
- Writing Interruptible Code
- Using Hardware Interrupts
- New for C64x+
Chapter Topics

Optimized Code and Interrupts ................................................................. 15-1

What is an Interrupt .............................................................................. 15-3

How Interrupts Work ............................................................................ 15-4

Writing Interruptible Code ................................................................... 15-6
Branches ................................................................................................. 15-6
Single Register Assignment ................................................................... 15-8
Where is Multiple-Assignment Used? .................................................. 15-10
Solutions to Multiple-Assignment and Interrupts ................................. 15-11
Code Generation Tools – Interruptibility Options ............................... 15-12
What is Your Interrupt Threshold? ....................................................... 15-12
Compiler Option – Set Interrupt Threshold Level (-mi) ......................... 15-12
MUST_ITERATE Pragma ......................................................................... 15-14
SPLOOP Buffer (C64x+) ....................................................................... 15-16

Using Hardware Interrupts ................................................................... 15-18
Interrupt Functions and the HWI Object ............................................. 15-18
Interrupt Functions ............................................................................... 15-18
Configuring a HWI Object .................................................................. 15-19
Two Ways to Create Interrupt Service Routines in C ......................... 15-20
HWI Dispatcher .................................................................................... 15-20
Interrupt Keyword ................................................................................ 15-22
ISR Summary ......................................................................................... 15-23
Additional Interrupt Information ......................................................... 15-24
Summary of Interrupt Flow ................................................................... 15-24
Additional Notes .................................................................................. 15-25

New For C64x+ Based Devices .............................................................. 15-26
Interrupts ............................................................................................... 15-26
Saving State (TSR, ITSR, NTSR Registers) ........................................... 15-27
Disabling/Restoring Interrupts (DINT, RINT Instructions) .................. 15-29
Interrupt Controller ............................................................................. 15-30
C64x+ Interrupts - Additional Notes ................................................... 15-35
Summary of Interrupt Controller Registers ......................................... 15-36
Exceptions ............................................................................................ 15-37
Four Exception Types .......................................................................... 15-37
How Exceptions Work ......................................................................... 15-43
Exception - Additional Notes ............................................................... 15-44
Exception Related Registers ............................................................... 15-45
What is an Interrupt

**What is an Interrupt?**

- An interrupt stops the current CPU process, which allows the CPU to attend to a higher priority event.

- Interrupts are external to the CPU
  - On-chip – timers, serial ports, & DMAs
  - Off-chip – analog-to-digital converters, host controllers, etc.

DSP are known for computing algorithms very quickly, but sometimes certain events in the system can have a higher priority than the algorithms already executing on the DSP. When this happens, it is necessary to change, or interrupt, the process currently executing on the DSP. The C6000 provides hardware on-chip that allows this to occur automatically.
How Interrupts Work

How Interrupts Work

How do Interrupts Work?

1. An interrupt occurs
   - DMA
   - HPI
   - Timers
   - Ext pins
   - Etc.

2. Sets a flag in a register

3. CPU acknowledges interrupt and ...
   - Stops what it is doing
   - Turn off interrupts globally
   - Clears flag in register
   - Saves return-to location
   - Determines which interrupt
   - Calls ISR

4. ISR (Interrupt Service Routine)
   - Saves context of system*
   - Runs your interrupt code (ISR)
   - Restores context of system*
   - Continues where left off*

* Must be done in user code, unless you choose to use the DSP/BiOS HWI dispatcher

Receiving Interrupts (with BIOS Functions)

Interrupt Enable Reg (IER) turns on individual ints
C64_enableIER(mask);
C64_disableIER(mask);

Interrupt Flag Reg (IFR) bit set when int occurs

Global Interrupt Enable (GIE) bit in Control Status Reg (CSR) enables all IER enabled interrupts
HWI_enable();
HWI_disable();
HWI_restore();
## Interrupt Related Registers and Associated CSL2 Functions

<table>
<thead>
<tr>
<th>Register/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ_set</td>
<td>(sets ISR bit which sets corresponding IFR bit)</td>
</tr>
<tr>
<td>IRQ_clear</td>
<td>(sets ICR bit which clears corresponding IFR bit)</td>
</tr>
<tr>
<td>IRQ_config</td>
<td></td>
</tr>
<tr>
<td>IRQ_test</td>
<td></td>
</tr>
<tr>
<td>IRQ_enable</td>
<td></td>
</tr>
<tr>
<td>IRQ_disable</td>
<td></td>
</tr>
<tr>
<td>IRQ_restore</td>
<td></td>
</tr>
<tr>
<td>ISR (Interrupt Set Register)</td>
<td></td>
</tr>
<tr>
<td>ICR (Interrupt Clear Register)</td>
<td></td>
</tr>
<tr>
<td>IFR (Interrupt Flag Register)</td>
<td></td>
</tr>
<tr>
<td>IER (Interrupt Enable Register)</td>
<td></td>
</tr>
<tr>
<td>IRP (Interrupt Return Pointer)</td>
<td></td>
</tr>
<tr>
<td>IRP (Non-maskable Int. Return Ptr.)</td>
<td></td>
</tr>
<tr>
<td>ISTP (Interrupt Service Table Ptr.)</td>
<td></td>
</tr>
</tbody>
</table>

*Chip Support Library (CSL 2.x): Interrupt functions*

*IRQ_setVecs or Use Config Tool*
Writing Interruptible Code

To write interruptible code, there are several rules that must be taken into consideration. For instance, can the code be stopped and then started again and produce the same results? Here are some topics to consider so that code can be interrupted.

- Pending branches or interruptible loops
- Single assignment registers vs. multiple assignment registers
- Compiler generated code that can or cannot be interrupted

Branches

Because the delay slots of all branch operations are protected from interrupts in hardware, all interrupts remain pending as long as the CPU has a pending branch. Since the branch instruction on the C6000 has 5 delay slots, loops smaller than 6 cycles always have a pending branch. For this reason, all loops smaller than 6 cycles are uninterruptible.

```
loop1:
  ldw
  mvc
  ext
  [] b loop1
  nop
  mpy
  stw
  add
  shl
```

**Non-Interruptible Branch**

- Branch operations take 6 cycles.
- Loops smaller than 6 cycles always have pending branches.
- Branches and their delay slots are not interruptible.
- Similar limitations found with most zero-overhead looping instructions (eg. RPTS) on other DSPs.
There are four ways to handle the non-interruptible Branch instruction.

**Interrupt Solutions for Branch**

1. **Unroll the loop so the loop is 6 cycles or more**
   - Not only guarantees interruptibility
   - High Performance
   - But, may increase the code size

2. **Slow down the loop so it’s 6 cycles (or more)**
   - Guarantees interruptibility
   - But, may degrade performance
   - Worst case, NOPs may be needed to force six-cycle loop

3. **Use nested loops**
   - Uses fast non-interruptible inner loop
   - Run inner-loop so that it doesn't block interrupt threshold
   - May slightly increase code size and timing

4. **Poll for interrupts in loop**
   - Difficult to implement
   - Along with polling code, a special exit epilog and return prolog must be created

Solution 1 not only guarantees interruptibility, but also maintains higher performance. Unfortunately, as we’ve seen earlier, the code size may increase slightly. Additionally, loop unrolling places constraints on the loop counter. If the code must be unrolled to six cycles, then either the number of iterations must be a factor of 6, or else the “odd” case must be handled – which is a bit more work (and code).

Solution 2 always works, but is least desirable due to lower performance. If all else fails, though, the compiler will use this option to maintain accurate results.

Solution 3 allows you to use the best performance loop (without unrolling), but you limit the number of times it will run consecutively. After some number of loop iterations (where the loop iterations time length of loop is less than your interrupt threshold), the inner-loop would end, thus allowing a short period for interrupts to be responded to. The outer loop, where interrupts are not blocked could be written such that it works as part of the inner loop’s epilog/prolog.

Solution 4 is probably the most difficult to code. Also, you need enough functional unit resources to read and evaluate the IFR (interrupt flag register) within the loop. If an interrupt is detected as part of your polling, the code would need to branch to a special routine that would:

- Gracefully exit from the software pipelined code (an epilog of sorts)
- Service the interrupt
- Then return to software pipelined routine via a prolog-like routine

While these are straight-forward solutions to the interruptibility of branch instructions, this is only part of the issue. How values are assigned to registers in super-scalar RISC processors, such as the C6000, must also be dealt with …
Single Register Assignment

Register allocation on the C6000 can be classified as either single assignment or multiple assignment. Single assignment code is interruptible; multiple assignment is not interruptible.

Single assignment means that the register is used for only one value throughout that routine or piece of code. Comparatively, multiple assignment means that the register is assigned more than one value during a piece of code. Let’s look at some examples of this.

What is “Single Assignment”

Single assignment requires that no registers are read which have pending results.

You can see that register A1 contains the same value in both places. We learned earlier in the workshop that NOPs (or other instructions) should be used to force the MPY to wait until the LDW result has shown up in A1.

In other words, a register (such as A1) isn’t used again until its pending value has been written to it. This is the definition of single-assignment.
What if the previous code was written in a slightly different way:

```
MA:  ADD .S1  A7,A8,A0
     LDW .D1  *A0,A1
     MPY .M1  A1,A2,A3
```

In this case, leaving out the NOPs changes the meaning of the code, doesn’t it? In this case, does the MPY instruction use the *old* or *new* value of A1?

Of course, MPY uses the *old* value, since the *new* value being loaded from *A0* won’t be written into A1 for another few cycles. It isn’t that this is “bad” code, rather, it’s valid code that makes very good use of the register A1. In fact, you might say that it is very efficient code, since A1 virtually contains two values – the *old* one and a *new* one. (Sometimes the new value is called an *in-flight* value.)

This is the definition of **multiple-assignment** (i.e. not single-assignment) - when a register is assigned two different values. While they actually are separated in time, there are still two values assigned simultaneously (both the *old* and *new*).

**Hint:** You might even think of multiple-assignment to registers as sort of time-division multiplexing (TDM) of the registers.

Multiple-assignment is shown in the following figure, but with a twist …

![Diagram showing single and multiple assignment](image)

```
Multiple Assignment:
MA:  ADD .S1  A7,A8,A0
     LDW .D1  *A0,A1
     MPY .M1  A1,A2,A3
     NOP
     SHR .S1  A3,15,A3
     ADD .L1  A3,A4,A4
```

The twist … what happens if your system were to service an interrupt between the LDW to A1 and the MPY with A1? In this case, MPY would end up using the *new* value of A1, rather than the intended *old* value.
In summary, in-flight operations cause code to be uninterruptible due to unpredictability. This unpredictability means that in order to ensure correct operation, multiple-assignment code should not be interrupted.

**Where is Multiple-Assignment Used?**

We have already written quite a bit of multiple-assignment code in this workshop. (And so has the compiler.) Multiple-assignment is commonly found when software pipelining. Look at the example below:

In this case, not only is this code juggling old/new values, A1 is virtually holding five values. While extraordinarily efficient, there are obvious interruptability issues.
Solutions to Multiple-Assignment and Interrupts

What can we do to protect multiple-assignment code from being corrupted by an interrupt?

One solution entails turning off interrupts around high-speed loops. (This is the compiler’s default solution; the next topic discusses how to control the compiler’s interrupt behavior.) This solution assumes that interrupts can be disabled during tight loops. Based upon the speed of the C6000’s CPU, this isn’t a bad assumption. Of course, there are many systems where this might be hazardous; systems where there can be one (or more) extremely fast real-time constraints – or algorithms with very large loops.

Multiple Assignment: Two Solutions

1. Turn off interrupts during the loop
2. Use single-assignment:
   + Single-assignment register usage won’t break interrupts
   + Maintains high performance
   - Uses more registers
   - May require loop unrolling (thus, larger code size)

Multiple Assignment:

<table>
<thead>
<tr>
<th>MA:</th>
<th>LDW .D1 *A0,A1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LDW .D1 *A0,A1</td>
</tr>
<tr>
<td></td>
<td>LDW .D1 *A0,A1</td>
</tr>
<tr>
<td></td>
<td>LDW .D1 *A0,A1</td>
</tr>
<tr>
<td></td>
<td>MPY .M1 A1,A2,A3</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Single Assignment:

<table>
<thead>
<tr>
<th>SA:</th>
<th>LDW .D1 *A0,A1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LDW .D1 *A0,A2</td>
</tr>
<tr>
<td></td>
<td>LDW .D1 *A0,A3</td>
</tr>
<tr>
<td></td>
<td>LDW .D1 *A0,A4</td>
</tr>
<tr>
<td></td>
<td>LDW .D1 *A0,A5</td>
</tr>
<tr>
<td></td>
<td>MPY .M1 A1,A6,A7</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

An alternative solution requires the programmer to adhere to single-assignment, which avoids these interruptibility issues. Using more registers allows you (or the compiler) to create interruptible, high-performance loops. The cost of this solution may be code-size, register pressure (running out of available registers), and flexibility.

While discussing the concepts of these various solutions provides good background knowledge, the real question is:

How can we get the tools to help us create high-speed, interruptible code?
Code Generation Tools – Interruptibility Options

The C6000 compiler provides a means of tuning the compiler to your systems particular interruptibility specifications. More specifically, we will discuss one compiler option and two #pragma statements that you can use to define your systems requirements.

What is Your Interrupt Threshold?

<table>
<thead>
<tr>
<th>Interrupt Rate</th>
<th>3µs</th>
</tr>
</thead>
</table>

What is Your Threshold?

◆ If you need high-speed loops and interruptibility ...
◆ How often does your system need to be able to recognize an interrupt?
  • 1 second
  • 1 millisecond
  • 1 microsecond?
◆ In other words, what is your interruptibility threshold?

Compiler Option – Set Interrupt Threshold Level (-mi)

The compiler provides the –mi option which lets you tell the compiler how many (instruction) cycles can go by before your system has to be allowed interrupt visibility. For example, by setting the –mi option to a 100 cycle threshold:

- mi 100

You have indicated to the compiler (and Assembly Optimizer) that your system’s interrupt threshold is 100 (instruction) cycles. In other words, the compiler should not build any uninterruptible loops larger than 100 cycles in length.

Note: When you think of it, though, 100 cycles is pretty fast real-time demand. For the original C6000 device (C6201), it’s only .5 µsec. Many systems can live with a much larger threshold, which provides the compiler quite a bit more flexibility.

You access the compiler options to set the interrupt threshold level as you would any other compiler option: Select: **Project → Build Options**. Then choose the **Advanced** category on the
Compiler tab. Check the Interrupt Threshold box and fill-in the appropriate threshold value in the box to the right.

**Compiler’s Interrupt Threshold (-mi)**

- With `-mi`, you to tell the compiler what cycle period is required between interrupts, `-mi <threshold>`
- If the interrupt threshold number will not be exceeded, within a loop the compiler may disable interrupts & use multiple-assignments to a reg.
- If compiler cannot determine loop count, it assumes the threshold is exceeded and generates an interruptible loop (albeit, maybe a slower loop)
- To control this on a function (vs. project) level, use:
  ```c
  #pragma FUNC_INTERRUPT_THRESHOLD(func, threshold);
  ```

The `-mi` option applies to the entire project. Sometimes there is a need to override this project-level specification on a function-by-function basis. Alternatively, you may chose to selectively apply a threshold value for only certain functions in your system, leaving the compiler the ability to efficiently turn-off interrupts when required for the most efficient coding of high-speed loops.

In either case, the #pragma FUNC_INTERRUPT_THRESHOLD shown in the preceding figure allows selective threshold values to be specified.

Here are some additional details regarding the compiler’s interpretation of the threshold value.

**-mi Details**

- `-mi 0`
  - Compiler’s code is not interruptible
  - User must guarantee no interrupts will occur
- `-mi 1`
  - Compiler uses single assignment and never produces a loop less than 6 cycles
- `-mi 1000` (or any number > 1)
  - Tells the compiler your system must be able to see interrupts every 1000 cycles
- **When not using `-mi` (compiler’s default)**
  - Compiler will software pipeline (when using `-o2` or `-o3`)
  - Interrupts are disabled for s/w pipelined loops

**Notes:**
- Be aware that the compiler is unaware of issues such as memory wait-states, etc.
- Using `-mi`, the compiler only counts instruction cycles
**MUST_ITERATE Pragma**

While the ability to specify a threshold via –mi or #pragma is incredibly useful, it is only part of the solution. Here’s an example that isn’t fully solved by –mi:

*What happens if a loop counter is set based upon a runtime input? That is, what events in your system determine the actual loop-count?*

*In other words, if the compiler doesn’t know how many times a loop will be executed it doesn’t really matter if you have specified –mi100.*

The compiler is pretty good at tracking down loop count values … if it’s determined within the scope of C code. In cases where the loop count is determined outside the scope of the compiler, you can assist the compiler by giving it additional information. The key to this information passing scheme is another #pragma.

---

**MUST_ITERATE**

🔹 **How does the compiler know how many cycles a loop takes to iterate?**

1. Most of the time the compiler can deduce this by doing full program optimization (-pm)
2. When it can’t figure it out, it must use a 6 cycle loop
3. Tell the compiler maximum number of iterations with:

   ```c
   #pragma MUST_ITERATE(min,max,factor);
   ```

The C code MUST_ITERATE pragma (and the Assembly Optimizer’s counterpart, the .trip directive) allows you to specify one, two, or three important specifications for a given loop:

- **Min** What is the minimum number of cycles the loop will run.
  This is used by the compiler to reduce code-size when software pipelining loops.
- **Max** The compiler uses this along with the interrupt threshold, as described in this section.
- **Factor** Describes the divide-by factor for the loop count. In other words, if you know the loop count will always be even, or say, divisible by 4, the factor argument allows you to specify this.
It comes down to this, the more the compiler knows about your code, the better job it can do producing efficient, high-speed code.

**MUST_ITERATE Example**

Here is a simple example that demonstrates the combined use of –mi and #pragma MUST_ITERATE.

```c
int dot_prod(short *a, Short *b, int n) {
    int i, sum = 0;
    #pragma MUST_ITERATE ( ,512)
    for (i = 0; i < n; i++)
        sum += a[i] * b[i];
    return sum;
}
```

- **Provided:**
  - If interrupt threshold was set at 1000 cycles (-mi 1000),
  - Assuming this can compile as a single-cycle loop,
  - And 512 = max# for Loop count (per MUST_ITERATE pragma).
- **Result:**
  - The compiler knows a 1-cycle kernel will execute no more than 512 times which is less than the 1000 cycle interrupt disable option (-mi1000)
  - Uninterruptible loop works fine
- **Verdict:**
  - 3072 cycle loop (512 x 6) can become a 512 cycle loop

Given both of these inputs, the compiler is allowed to produce much better code. While it probably can beat the 6:1 ratio without these inputs (via loop unrolling), there is no doubt that the compiler can produce significantly better code with this information.

**Hint:** You can use #pragma MUST_ITERATE even if you don’t know all three values (min, max, and factor). As in the example above, just fill in the values you know to be valid.
**Writing Interruptible Code**

**SPLOOP Buffer (C64x+)**

- **SPLOOP is Interruptible**
  - **Prolog**
  - **Kernel**
  - **Epilog**

- SPLOOP provides interruptibility for software pipelined loops!
- Pipes-down the loop before branching to interrupt
  1. Hardware detects an interrupt
  2. Unless termination condition is true (ILC=0), Pipe-down the SPLOOP.
  3. Store return address of SPLOOP in IRP or NRP.
  4. Save system state (TSR register) with SPLOOP enable bit set
  5. Begin execution at interrupt service routine target address.
- Upon return from interrupt, loop must be piped back up
- Worst case delay is 150 cycles (considerably less than our –mi100 example)

**Example 1 – C64x to C64x+**

- -mi<n> can cause decreased performance since compiler must either assure loops are smaller than <n> cycles or use loops with >= 6 and single assignment
- SPLOOP provides interruptibility for s/w piped loops, minimizing impact of -mi
- TCPI/IP has several small loops with a size < 6, notice big affect of –mi on the C64x
-mi100 does not affect gsmefr on C64x as much as tcpip because the loops tend to be larger, therefore, the compiler doesn't have to do as much to make these loops interruptible.
Using Hardware Interrupts

The Interrupt Service Routine (ISR) is simply the function that is called to run by an interrupt. The C6000 provides hardware to automatically branch to this routine when an interrupt is received based on an interrupt service vector table - which is pointed to be the Interrupt Service Table Pointer (ISTP). Once the branch is complete, execution begins at the first execute packet of the ISR.

Certain states must be saved upon entry to an ISR in order to ensure program accuracy upon return from the interrupt. For this reason, all registers that are used by the ISR must be saved to before they can be used.

This topic discusses the various interrupt functions required, setting up interrupts via the DSP BIOS Hardware Interrupt object (HWI), and creating Interrupt Service Routines in C code.

Interrupt Functions and the HWI Object

Interrupt Functions

Two functions must be written in order to use hardware interrupts:

- A function to enable the various interrupts in the system. This function is usually called from main. It enables the appropriate interrupt enable bits in the IER register discussed earlier in the chapter.
- The Interrupt Service Routine (ISR) is the second function required. On page 15-20 we discuss how to turn an ordinary C function into an ISR.

```c
MY_enableHWI()
{
    C64_enableIER( mask );
}

void MY_edmaHwi( void )
{
    //perform some action, say,
    //posting a Software Interrupt
    SWI_post( &mySWI );
}
```

- Two functions are required when using Hardware Interrupts
  1. Enable the interrupt(s) you want to use
  2. An interrupt service routine which runs when each interrupt occurs
**Configuring a HWI Object**

TI’s DSP/BIOS realtime kernel provides an easy way to configure the interrupts you want to use. Of the two items shown below:

- The *Function* field is used by BIOS to create the interrupt vector table for you.
- The *Interrupt Source* associates the interrupt event source you choose with the hardware interrupt object (HWI) number. (This is further discussed in the *Interrupt Selector* topic on page 15-32.)

---

**Note:** To keep the development of the CSL and BIOS libraries completely independent of each other, BIOS 5.2 and beyond does not call IRQ_map() for you. To this end, your hardware initialization routine will need to call both IRQ_map() and IRQ_enable for each interrupt you plan to use. After calling IRQ_map once, you can freely use IRQ_enable(), IRQ_disable(), and IRQ_restore() without calling IRQ_map() again.
Two Ways to Create Interrupt Service Routines in C

**HWI Dispatcher**

The *Hardware Interrupt Dispatcher* is an easy way to handle all interrupt context saves and restores. By the way, HWI Dispatcher is especially useful when nesting interrupt service routines written in C. Just look at some of its other features in the figure below.

![Using the HWI Dispatcher](image)

**Using the HWI Dispatcher**

- **Select Dispatcher tab**
- **Click Use Dispatcher**
- **Default Mask is Self**, this means all interrupts will preempt except this one
- **Select another mask option, if you prefer**
  - **All**: Best choice if ISR is short & fast
  - **None**: Dangerous, Make sure ISR code is re-entrant
  - **Bitmask**: Allows custom mask
Currently executing code
{  
  interrupt occurs
  next EP
}

HWI Dispatcher
Flow of Events

<table>
<thead>
<tr>
<th>Vector Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>HWI Dispatcher:</td>
</tr>
<tr>
<td>Context Save</td>
</tr>
<tr>
<td>Context Restore</td>
</tr>
</tbody>
</table>

- Uses standard (unmodified) C function
- Use algorithm from an object file or library
- Required when HWI ISR uses some DSP/BIOS scheduler functions
- Easy to use - simple checkbox
- Simple way to nest interrupts
- Saves code space - since ints can share one context save/restore routine

```c
void MY_edmaHwi(arg)
{
  //perform some action,
  // say, posting a SWI
  SWI_post(&mySWI);
}
```
**Interrupt Keyword**

The C compiler provides an interrupt keyword that specifies that a function is treated as an interrupt function. Functions that handle interrupts follow special register-saving rules (context saves) and a special return sequence (context restore and branch to next execution packet before interrupt was received). You can only use the interrupt keyword with a function that returns void and has no parameters.

- **When using the `interrupt` keyword:**
  - Compiler handles register preservation
  - Returns to original location
- **No arguments (void)**
- **No return values (void data type)**
- **You cannot call BIOS scheduler functions**
**ISR Summary**

**Interrupt Creation Summary**

1. **HWI Dispatcher**
   - Allows nesting of interrupts
   - Saves code space
   - Required when ISR uses BIOS scheduler functions
   - Allows an argument passed to ISR

2. **Interrupt Keyword**
   - Provides highest code optimization (by a little bit)

**Notes:**
- Choose *HWI dispatcher* and *Interrupt* keyword on an interrupt-by-interrupt basis

**Caution:**
For each interrupt, use only one of these two interrupt context methods

---

**Note:** Please don’t miss the CAUTION in the figure above!
Additional Interrupt Information

Summary of Interrupt Flow

In the C6000 Optimization Workshop we introduce the basic fundamentals of C6000 interrupts and how to enable them. This provides a basis upon which we can discuss the issues between interruptibility of code and high-speed optimization of code.

While some of the issues in the following diagram go beyond the scope of the Optimization Workshop (OP6000), it does provide a good overview of the interrupt model for the C6000. It has been provided for reference only, as some instructors find it useful when answering questions during breaks, or before and after class.

Interrupts and how to use them are discussed at length in the C6000 Integration Workshop (IW6000). Please refer to Appendix B for further information on the IW6000 Workshop. Also, you may want to check the Introduction chapter to find out how to sign up for the IW6000 Workshop.
Additional Notes – Hardware Interrupts (HWI)

- DSP/BIOS Config tool automatically creates vector table
- Both methods (HWI Dispatcher, Interrupt keyword) use the System Software Stack discussed in Advanced Memory Mgmt chapter
- HWI are considered the highest priority thread within DSP/BIOS and will preempt any other code (if enabled)
- HWIs are serviced in order of priority (INT4 to INT15)
- Hardware default – one HWI does not preempt another; when a running HWI returns, then execution will pass to the highest priority HWI flagged in the Interrupt Flag Register. This is done by setting GIE=0 upon start of ISR
- When preemption amongst HWIs is desired, default HWI scheduling can be manually overridden
  - HWI Dispatcher make HWI preemption easy
  - Preemption is more difficult using the interrupt keyword
  - While seemingly desirable, HWI preemption is often less desirable, rather it’s becoming more popular to keep HWIs short & fast so nesting is not required
Interrupts – New C64x+ Features

- SPLOOP  (Discussed Earlier)
- Saving State  (TSR Register)
- Disabling Int’s  (DINT/RINT)
- Interrupt Controller
- Exceptions

Interrupt Support is *almost* unchanged.

As discussed on page 15-16, the SPLOOP buffer now provides interruptibility for highly optimized software-pipelined loops.

New registers provide system state information: TSR, ITSR, NTSR. These were required to support new features found in C64x+ based devices.

Two new instructions introduced to support interrupts: DINT, RINT.

The Interrupt Controller was greatly enhanced. It provides the ability to route up to 128 different system events to the hardware interrupt (HWI) objects. Four of these events can even be combined from the other 124.

Default value for ISTP (Interrupt Service Table Pointer) has changed:

- Was 0 on C64x DSP
- (Default RESET) Is non-zero on C64x+ DSP
- Refer to datasheet for correct value for a specific device

Interrupt Latency has changed:

- Interrupt latency was 7 cycles on C64x. Now 9 cycles on C64+.
- Interrupt overhead was 11 cycles on C64x. Now 13 cycles on C64x+.
Saving State *(TSR, ITSR, NTSR Registers)*

The TSR (Task State Register) supports new product features
- SPLOOP
- Exceptions
- Privilege mode operation

The TSR stores information about the current executing environment:
- Is an SPLOOP currently operating?
- Are exceptions and/or interrupts enabled?
- What is the Privilege state (user mode or privileged mode)?
- Is an exception or interrupt currently processing?
- Are interrupts architecturally blocked?

---

**Task State Register (TSR)**

- **New C64x+ features** (SPLOOP, Privilege mode, Exceptions) required enhanced processor state information...
- **TSR stores info about the current executing environment**

<table>
<thead>
<tr>
<th>IB</th>
<th>SPLX</th>
<th>Rsvd</th>
<th>EXC</th>
<th>INT</th>
<th>Rsvd</th>
<th>CXM</th>
<th>Rsvd</th>
<th>XEN</th>
<th>GEE</th>
<th>SGIE</th>
<th>GIE</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-0</td>
<td>R-0</td>
<td>R-0</td>
<td>R/C-0</td>
<td>R-0</td>
<td>R-0</td>
<td>R/W-0</td>
<td>R-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
</tr>
</tbody>
</table>

- **“Task” in TSR’s name is not related to DSP/BIOS TSK**
- **TSR.GIE**
  - Physically same bit as CSR.GIE
  - Keeping CSR bit provides backward compatibility

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-16</td>
<td>Reserved</td>
<td>Reserved (Read as 0)</td>
</tr>
<tr>
<td>15</td>
<td>IB</td>
<td>Interrupts Blocked</td>
</tr>
<tr>
<td>14</td>
<td>SPLX</td>
<td>SPLOOP Executing</td>
</tr>
<tr>
<td>10</td>
<td>EXC</td>
<td>Processing Exception</td>
</tr>
<tr>
<td>9</td>
<td>INT</td>
<td>Processing Interrupt</td>
</tr>
<tr>
<td>6-7</td>
<td>CXM</td>
<td>Current Execution Mode (Supervisor/User)</td>
</tr>
<tr>
<td>3</td>
<td>XEN</td>
<td>Maskable Exception Enable</td>
</tr>
<tr>
<td>2</td>
<td>GEE</td>
<td>Global Exception Enable</td>
</tr>
<tr>
<td>1</td>
<td>SGIE</td>
<td>Saved Global Interrupt Enable</td>
</tr>
<tr>
<td>0</td>
<td>GIE</td>
<td>Global Interrupt Enable</td>
</tr>
</tbody>
</table>
ITSR and NTSR save copies of the TSR register during interrupts and NMI/Exceptions respectively.

### TSR, NTSR, and ITSR Registers

- Previously, upon interrupt CSR.GIE was copied to CSR.PGIE

... Now ...

- ITSR and NTSR save copies of the TSR register during interrupts and NMI/Exceptions respectively

![Diagram of TSR, NTSR, and ITSR Registers]

### TSR, NTSR, and ITSR Registers

- Previously, upon interrupt CSR.GIE was copied to CSR.PGIE

... Now ...

- ITSR and NTSR save copies of the TSR register during interrupts and NMI/Exceptions respectively
Disabling/Restoring Interrupts (DINT, RINT Instructions)

DINT and RINT Instructions (C64x+)

- Global enable/disable of maskable interrupts:
  - H/W saves GIE bit into SGIE bit in TSR

    | IB | SPLX | Rsvd | EXC | INT | Rsvd | CXM | Rsvd | XEN | GEE | SGIE | GIE |
    |----|------|------|-----|-----|------|-----|------|-----|-----|------|-----|
    |    |      |      |     |     |      |     |      |     |     |      | 1   |
    |    |      |      |     |     |      |     |      |     |     |      | 0   |

- Used to avoid possible race conditions by making the interrupt disable/enable an atomic operation

<table>
<thead>
<tr>
<th>C64x code</th>
<th>C64x+ code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVC CSR,B0</td>
<td>DINT</td>
</tr>
<tr>
<td>EXTU B0,29,30,B1</td>
<td>// Clear GIE</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>MVC B0,CSR</td>
<td>// value in TSR.SGIE</td>
</tr>
<tr>
<td>...</td>
<td>// Protected // Instructions</td>
</tr>
<tr>
<td>MVC CSR,B0</td>
<td>RINT</td>
</tr>
<tr>
<td>[B1] OR B0,1,B0</td>
<td>// Restore GIE from B1 to B0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>MVC B0,CSR</td>
<td>// Write CSR</td>
</tr>
</tbody>
</table>

// Read CSR
// Store GIE in B1
// Clear GIE in B0
// Write CSR

Technical Training Organization

ttt
Interrupt Controller

C62x, C67x, C64x Interrupt Selector

The precursor to the new C64x+ Interrupt Controller was the Interrupt Selector found on the previous devices in the C6000 family. The earlier selector allowed the user to configure any of the 12 maskable interrupt objects with any of the interrupt event sources.

There are 12 configurable interrupts
- C6000 devices have more than 12 interrupt sources
- The interrupt selector allows you to map any interrupt source to any HWI object
- Side benefit is that you can change the hardware interrupt priority
**C62x, C67x, C64x Interrupt Selection**

<table>
<thead>
<tr>
<th>Sel #</th>
<th>C6701 Sources</th>
<th>Interrupt Multiplexer High (INT10 - INT15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b</td>
<td>(HPI) DSPINT, TINT0, TINT1</td>
<td>INTSEL15, INTSEL14, INTSEL13</td>
</tr>
<tr>
<td>0001b</td>
<td>SD_INT</td>
<td>INTSEL12, INTSEL11, INTSEL10</td>
</tr>
<tr>
<td>0010b</td>
<td>EXT_INT4, EXT_INT5, EXT_INT6</td>
<td></td>
</tr>
<tr>
<td>0011b</td>
<td>EXT_INT7, DMA_INT0</td>
<td></td>
</tr>
<tr>
<td>0100b</td>
<td>DMA_INT1</td>
<td></td>
</tr>
<tr>
<td>0101b</td>
<td>DMA_INT2</td>
<td></td>
</tr>
<tr>
<td>0110b</td>
<td>DMA_INT3, DMA_INT4, DMA_INT5, DMA_INT6, DMA_INT7</td>
<td>INTSEL9, INTSEL8, INTSEL7</td>
</tr>
<tr>
<td>1000b</td>
<td>DMA_INT1, DMA_INT2, DMA_INT3</td>
<td>INTSEL6, INTSEL5, INTSEL4</td>
</tr>
<tr>
<td>1010b</td>
<td>XINT0, RINT0, XINT1, RINT1</td>
<td></td>
</tr>
</tbody>
</table>

- Interrupt Selector registers are memory-mapped
- Configured by HWI objects in **Config Tool**
- Or, set dynamically using **IRQ_map()**

**Interrupt Selection with Config Tool**

<table>
<thead>
<tr>
<th>Sel #</th>
<th>C6701 Sources</th>
<th>Interrupt Multiplexer High (INT10 - INT15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b</td>
<td>(HPI) DSPINT, TINT0, TINT1</td>
<td>INTSEL15, INTSEL14, INTSEL13</td>
</tr>
<tr>
<td>0001b</td>
<td>SD_INT</td>
<td>INTSEL12, INTSEL11, INTSEL10</td>
</tr>
<tr>
<td>0010b</td>
<td>EXT_INT4, EXT_INT5, EXT_INT6</td>
<td></td>
</tr>
<tr>
<td>0011b</td>
<td>EXT_INT7, DMA_INT0</td>
<td></td>
</tr>
<tr>
<td>0100b</td>
<td>DMA_INT1</td>
<td></td>
</tr>
<tr>
<td>0101b</td>
<td>DMA_INT2</td>
<td></td>
</tr>
<tr>
<td>0110b</td>
<td>DMA_INT3, DMA_INT4, DMA_INT5, DMA_INT6, DMA_INT7</td>
<td>INTSEL9, INTSEL8, INTSEL7</td>
</tr>
<tr>
<td>1000b</td>
<td>DMA_INT1, DMA_INT2, DMA_INT3</td>
<td>INTSEL6, INTSEL5, INTSEL4</td>
</tr>
<tr>
<td>1010b</td>
<td>XINT0, RINT0, XINT1, RINT1</td>
<td></td>
</tr>
</tbody>
</table>

- Interrupt Selector registers are memory-mapped
- Configured by HWI objects in **Config Tool**
- Or, set dynamically using **IRQ_map()**
C64x+ Interrupt Controller

The Interrupt Controller is part of the C64x+ Megamodule and is discussed in the TMS320C64x+ DSP Megamodule Reference Guide (SPRU871.PDF).

Interrupt Selector

Inside the C64x+ Interrupt Controller, you’ll find a Interrupt Selector.
Event Combiner

Also in the Interrupt Controller is the Event Combiner. This takes the same 124 events that feed into the Interrupt controller and provide 4 maskable, combined outputs.
Interrupt Error Detection (IDROP)

The C64x+ CPU provides detection for dropped interrupts. That is, a dropped error can be detected whenever a new interrupt signal is detected at an Interrupt Flag Register (IFR) bit, but the bit is already set. This signal is fed back to the IDROP mask which stores the interrupt in error, and can signal the Interrupt Controller (and thus, the CPU) via Event 96 (EVT96).

C64x+ Dropped Interrupt Detection

Non-Maskable Event (NMEVT)

Maskable Events (EVT4 – EVT127)

EVT96

INTERR

IDROP Mask

IDROP[15:4]

C64x+ Interrupt Controller

Event Combiner

Interrupt Selector

CPU Interrupt Logic

HWI1

(NMI)

C6000 CPU

Missed Interrupt
Event Mapping (EVT0-EVT127)

<table>
<thead>
<tr>
<th>EVT #</th>
<th>Event</th>
<th>From</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EVT0</td>
<td>INT</td>
<td>Output of event combiner 0, for events 1-31</td>
</tr>
<tr>
<td>1</td>
<td>EVT1</td>
<td>INT</td>
<td>Output of event combiner 1, for events 32-63</td>
</tr>
<tr>
<td>2</td>
<td>EVT2</td>
<td>INT</td>
<td>Output of event combiner 2, for events 64-95</td>
</tr>
<tr>
<td>3</td>
<td>EVT3</td>
<td>INT</td>
<td>Output of event combiner 3, for events 96-127</td>
</tr>
<tr>
<td>4-12</td>
<td>Reserved</td>
<td>/ Unused</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>IDMINT0</td>
<td>EMC</td>
<td>IDMA channel 0 interrupt</td>
</tr>
<tr>
<td>14</td>
<td>IDMINT1</td>
<td>EMC</td>
<td>IDMA channel 1 interrupt</td>
</tr>
<tr>
<td>15-95</td>
<td>Reserved</td>
<td>/ Unused</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>INTERR</td>
<td>INT</td>
<td>Dropped CPU interrupt event</td>
</tr>
<tr>
<td>97</td>
<td>EMC_IDMERR</td>
<td>EMC</td>
<td>Invalid IDMA parameters</td>
</tr>
<tr>
<td>98-117</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>PDC_INT</td>
<td>PDC</td>
<td>PDC sleep interrupt</td>
</tr>
<tr>
<td>119</td>
<td>SYS_CMPA</td>
<td>SYS</td>
<td>CPU memory protection fault</td>
</tr>
<tr>
<td>120</td>
<td>PMC_CMPA</td>
<td>PMC</td>
<td>CPU memory protection fault</td>
</tr>
<tr>
<td>121</td>
<td>PMC_DMPA</td>
<td>PMC</td>
<td>DMA memory protection fault</td>
</tr>
<tr>
<td>122</td>
<td>DMC_CMPA</td>
<td>DMC</td>
<td>CPU memory protection fault</td>
</tr>
<tr>
<td>123</td>
<td>DMC_DMPA</td>
<td>DMC</td>
<td>DMA memory protection fault</td>
</tr>
<tr>
<td>124</td>
<td>UMC_CMPA</td>
<td>UMC</td>
<td>CPU memory protection fault</td>
</tr>
<tr>
<td>125</td>
<td>UMC_DMPA</td>
<td>UMC</td>
<td>DMA memory protection fault</td>
</tr>
<tr>
<td>126</td>
<td>EMC_CMPA</td>
<td>EMC</td>
<td>CPU memory protection fault</td>
</tr>
<tr>
<td>127</td>
<td>EMC_BUSERR</td>
<td>EMC</td>
<td>Bus error interrupt</td>
</tr>
</tbody>
</table>

C64x+ Interrupts - Additional Notes

Additional Notes – C64x+ Interrupts

- Default value for ISTP (Interrupt Service Table Pointer) has changed:
  - Was 0 on C64x
  - Default RESET location is non-zero on C64x+
  - Refer to datasheet for correct value for a specific device

- Interrupt Latency has changed
  - Interrupt latency was 7 cycles on C64x. Now 9 cycles on C64x+
  - Interrupt overhead was 11 cycles on C64x. Now 13 cycles on C64x+
# Summary of Interrupt Controller Registers

## C64x+ Interrupt Controller Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Base address</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVTFLAG[3:0]</td>
<td>0x01800000</td>
<td>Event flag registers</td>
<td>Status</td>
</tr>
<tr>
<td>EVTSET[3:0]</td>
<td>0x01800020</td>
<td>Event set registers</td>
<td>Command</td>
</tr>
<tr>
<td>EVTCLR[3:0]</td>
<td>0x01800040</td>
<td>Event clear registers</td>
<td>Command</td>
</tr>
<tr>
<td>EVTMASK[3:0]</td>
<td>0x01800080</td>
<td>Event mask registers</td>
<td>Control</td>
</tr>
<tr>
<td>EXPMASK[3:0]</td>
<td>0x018000C0</td>
<td>Exception mask registers</td>
<td>Control</td>
</tr>
<tr>
<td>MEVTFLAG[3:0]</td>
<td>0x018000A0</td>
<td>Masked event flag registers</td>
<td>Status</td>
</tr>
<tr>
<td>MEXPFLAG[3:0]</td>
<td>0x018000E0</td>
<td>Masked exception flag registers</td>
<td>Status</td>
</tr>
<tr>
<td>INTMUX[3:1]</td>
<td>0x01800104</td>
<td>Interrupt mux registers</td>
<td>Control</td>
</tr>
<tr>
<td>INTXSTAT</td>
<td>0x01800180</td>
<td>Interrupt exception status</td>
<td>Status</td>
</tr>
<tr>
<td>INTXCLR</td>
<td>0x01800184</td>
<td>Interrupt exception clear</td>
<td>Command</td>
</tr>
<tr>
<td>INTDMASK</td>
<td>0x01800188</td>
<td>Dropped interrupt mask register</td>
<td>Control</td>
</tr>
<tr>
<td>EVTASRT</td>
<td>0x018001C0</td>
<td>Event assert register</td>
<td>Command</td>
</tr>
</tbody>
</table>
Exceptions

C64x+ Exceptions - Purpose

- Trap illegal instruction
  - Code or data corruption
  - Resource conflicts
  - Invalid use of hardware
- Mechanism to communicate between privilege levels
- Error signaling mechanism for peripherals and system resources

Four Exception Types

Previous to the C64x+, the C6000 devices only had a single, high-priority interrupt which was generated by the NMI pin. The C64x+ CPU expands on this by allowing four different event sources to generate a high priority interrupt – this new capability is called an exception. If exceptions are enabled, each of the following exception sources will be serviced by the NMI interrupt vector. (The details of enabling and using exceptions will be covered shortly.)

C64x+ Exception Types

1. External Non-maskable (NMI pin)
   - Serious/fatal Hardware problem

2. External Maskable (EXCEP)
   - External pins
   - L2 error detection
   - Chip level hardware exceptions

3. Internal (IERR)
   - Generated by CPU error
   - Errors listed in upcoming slide
   - Non-maskable

4. Software Generated (SWE/SWENR)
   - Triggered by SWE or SWENR instructions
   - CPU Ref Guide groups this with “Internal” exception
1. Non-Maskable External Exception (NXF)

Note, these are external to the CPU. The maskable exceptions could be generated from off-chip or on-chip events. “On-chip” events is meant to include the on-chip peripherals or any other control or error signals the designers need to use for a specific device.
1. **EFR.NXF (NMI) and 2. EFR.EXF (EXCEP)**

- **EFR.NXF (NMI)**
  - NMI pin
  - EVT[127:4]
  - NMEVT
  - 1

- **EFR.EXF (EXCEP)**
  - EVT[96]
  - INTERR
  - 2

**Interrupt Controller**
- Event Flags
- Event Combiner
- Interrupt Selector
- IDROP Mask

**CPU**
- EXCEP
- NMI
- INT[15:4]
- IDROP[15:4]
2. Maskable External Exception (EXF)

The maskable exception is a combination of all the desired event inputs in the Interrupt Controller. As with the Interrupt Event Combiner, the Event Combiner has both a mask register and a masked event flag register. The mask register lets you choose which events should affect the EXCEP signal. The masked event flag register provides latched bits that can be read in the exception service routine to determine the cause of a maskable exception.

![Diagram of Exception Combiner](image-url)
3. Internal Exceptions (IXR, IERR)

Internal exceptions are those generated from within the C64x+ megamodule.

3. C64x+ Typical Causes of Internal Exceptions

- Branch to middle of 32-bit instruction
- Branch to fetch packet header
- Illegal or reserved instruction
- Attempt to execute reserved opcode
- Attempt to access restricted register or instruction (privilege violation)
- Simultaneous writes to same register
- Two taken branches in same execute packet
- SPLOOP buffer exception
  - Unit conflict (attempt to use same unit)
  - Encountered unexpected SPKERNEL
  - Write to ILC or RILC in prohibited timing window

These internal exceptions are flagged in Internal Event Exception Register (IERR). Whenever the EFR.IXF bit is set, you need to read the IERR to determine the actual cause.

3. Exception Register (IERR)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-8</td>
<td>Reserved</td>
<td>Reserved. Read as 0.</td>
</tr>
<tr>
<td>7</td>
<td>LBX</td>
<td>Loop buffer exception (SPLOOP stall required)</td>
</tr>
<tr>
<td>6</td>
<td>PRX</td>
<td>Privilege exception (user access to supervisor memory page)</td>
</tr>
<tr>
<td>5</td>
<td>RAX</td>
<td>Resource access exception (user access to registers)</td>
</tr>
<tr>
<td>4</td>
<td>RCX</td>
<td>Resource conflict exception (Concurrent writes to registers)</td>
</tr>
<tr>
<td>3</td>
<td>OPX</td>
<td>Op-code exception (illegal instruction)</td>
</tr>
<tr>
<td>2</td>
<td>EPX</td>
<td>Execute packet exception</td>
</tr>
<tr>
<td>1</td>
<td>FPX</td>
<td>Fetch packet exception</td>
</tr>
<tr>
<td>0</td>
<td>IFX</td>
<td>Instruction fetch exception</td>
</tr>
</tbody>
</table>
4. Software Generated Exceptions (SWE, SWENR)

The fourth type of exception is also considered an “internal exception”. In this case, though, it is generated by the program code by calling either the SWE or SWENR instruction.

4. Software Generated Exception (SWE)

- **SWE (Software Exception)**
  - Forces an exception under software control
  - Can be used by a user mode task to request system services which require privileged access to memory, registers or other resources
  - Return address is stored in NRP so that the exception routine can return to the calling code
  - Transfer is to NMI interrupt vector

- **SWENR (Software Exception – No Return)**
  - Similar to SWE except that no provision is made to return to the point of the exception
  - Transfer of control is to the address specified by the REP register instead of the Interrupt Service Table
  - SWENR can be used by a user mode task to terminate and return to the privileged mode task which invoked it.
How Exceptions Work

Enabling Exceptions

1. Enabling Exceptions

- (default = off)
- C64x+ CPU
- OR
- NMI

To enable exceptions:
1. IER.NMIE – done for you by BIOS_init()
2. TSR.XEN
3. TSR.GEE

Exception - Occurs, Is Flagged, Is Recognized/Responded To

How Exceptions Work?

1. Enable Exceptions

2. An Exception occurs

- Maskable
- Non-Maskable
- Internal
- Software (SWE/SWENR)

3. Sets flag in EFR register

4. CPU acknowledges exception by automatically...

- Stops what it is doing
- Does not clear flag(s) in EFR
- Turns off IER.NMIE
- TSR is copied to NTSR
- TSR is modified:
  - TSR.EXC set (exception processing)
  - TSR.CXM cleared (supervisor mode)
  - Clears TSR.GIE, .SGIE, .XEN, .DBGM, .INT
  - Saves return-to location in NRP
- Branches to NMI interrupt vector (or REP register if caused by SWENR)

5. Exception Service Routine...
Exception Service Routine

How Exceptions Work?

5. Exception Service Routine

- Save context of system*
  • Any registers needed by exception routine

- Determine cause(s) of exception
  • Test EFR to determine what caused the exception
  • If EFR.IXF, read Internal Excep Reg (IERR)
  • If EFR.EXF, read MEXPFLAG[3:0]
  • Clear any/all flag bits in EFR, IERR, EVTFLAG, MEXPFLAG

- Perform whatever code needed to respond to the exception

- Before returning, verify if safe return is possible
  • Can the cause of the exception be returned from? Based on the cause, you must determine this.
  • Read NTSR.IB – if set, the exception cannot return safely

- Restore context of system*
- Continues where left off*
  • Branch to NRP (causes TSR to be restored from NTSR)

* Must be done in user code, unless you choose to use the DSP/BiOS HWI dispatcher

Exception - Additional Notes

Exceptions – Additional Notes (1)

- Backward Compatibility
  • Unless GEE is set, previous NMI functionality is unchanged
  • When GEE is set, NMI is converted to handle exceptions
  • Since GEE is a “new” bit in a “new” register, backward compatibility is maintained (you have to know about it to set it)

- Once set, GEE cannot be disabled (except via Reset)

- NTSR.IB (Interrupts Blocked)
  • Branches and their delay slots are uninterruptible, hence IB indicates that interrupts are blocked
  • When NTSR.IB = 1, no interrupt can occur, not even NMI
  • Before C64x+ any loop < 6 cycles always had branches, therefore even a watchdog timer hooked to NMI could be missed (Often NMI and Reset were both required to guarantee a watchdog response)
  • Workaround: Use the -mi option to keep loops <6 cycles
  • Exceptions solve this dilemma as they can disrupt branches, which is why it’s important to verify NTSR.IB is not set before returning from exception routines
Exceptions – Additional Notes (2)

- Only thing that can delay an Exception is a memory stall
- Nested Exceptions
  - Exceptions can be nested by re-enabling NMIE within your Exception Service Routine
  - If an exception occurs when TSR.EXC is set:
    - Reset vector is used (rather than NMI’s vector)
    - TSR is copied to ITSR (rather than NTSR)
    - Return address is put into IRP (rather than NRP)
    - As usual for interrupts, TSR is set to default exception processing values and IER.NMIE is cleared (to prevent further nesting)
  - Boot code should check ITSR, NTSR, IRP, and NRP to determine if Reset was caused due to Reset pins or nested exceptions

Exception Related Registers

C64x+ Exception Control Registers

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Register Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECR</td>
<td>Exception clear register</td>
<td>Used to clear pending exception flags.</td>
</tr>
<tr>
<td>EFR</td>
<td>Exception flag register</td>
<td>Contains pending exception flags.</td>
</tr>
<tr>
<td>IER</td>
<td>Interrupt enable register</td>
<td>Contains NMI exception enable (NMIE) bit</td>
</tr>
<tr>
<td>IERR</td>
<td>Internal exception report register</td>
<td>Indicates cause of an internal exception</td>
</tr>
<tr>
<td>REP</td>
<td>Restricted entry point address register</td>
<td>Contains the address to where the SWENR instruction transfers control.</td>
</tr>
<tr>
<td>TSR</td>
<td>Task state register</td>
<td>Contains global exception enable (GEE) and exception enable (XEN) bits.</td>
</tr>
<tr>
<td>NTSR</td>
<td>Non-maskable interrupt/exception task state register</td>
<td>Stores contents of TSR upon taking an exception</td>
</tr>
<tr>
<td>ISTP</td>
<td>Interrupt service table pointer register</td>
<td>Pointer to the beginning of the interrupt service table that contains the exception interrupt service fetch packet</td>
</tr>
<tr>
<td>NRP</td>
<td>Non-maskable interrupt return pointer register</td>
<td>Contains the return address used on return from an exception. This return is accomplished via the B NRP instruction.</td>
</tr>
</tbody>
</table>
This page intentionally left blank.
Numerical Issues

Introduction

It is important to understand the basic numbering system used by the C6000 and common issues that need to be dealt with while manipulating these numbers. This module covers binary numbers and how to handle overflows.

Outline

- Overview of binary numbers
- Handling multiplicative overflow
- Describe three methods for handling accumulative overflow
Chapter Topics

Binary Basics ................................................................. 16-3
  (Unsigned) Binary Numbers ........................................... 16-3
  2’s Complement .......................................................... 16-4
  Where Numbers Come From ........................................ 16-6

Multiplicative Overflow .................................................. 16-7
  Definition of Multiplicative Overflow ......................... 16-7
  Solving Multiplicative Overflow ................................. 16-8
  Fractional Multiply ..................................................... 16-9
  Binary Multiplication .................................................. 16-13
  16-bit Fractional Math ................................................ 16-16
  Coding Example .......................................................... 16-17
  -1 x -1 = ? ................................................................. 16-19
  Handling Multiplicative Overflow in C Code ................. 16-20
  Multiplicative Overflow – Summary ........................... 16-23

Accumulative Overflow .................................................. 16-24
  Definition of Accumulative Overflow ......................... 16-24
  Accumulative Overflow Solutions ............................... 16-25
  Saturating the Result ................................................... 16-26
  Gaining Headroom (Guard bits) ................................. 16-32
  Overflow Allowed by Design ...................................... 16-34
  Accumulative Overflow - Review ............................... 16-36
Binary Basics

Before jumping into discussing binary operations, 2’s complement and sign extension, let’s spend a few minutes reviewing some of the basic concepts.

(Unsigned) Binary Numbers

Each binary digit carries a specific weight similar to the decimal numbering system. For example, if you plug in 0011_2 into the weighting system below, you get 2 + 1 = 3. Likewise, you obtain the value 10_10 when 1010_2 is used. The largest number you can represent in 4 bits is 15 or 1111_2. The smallest number is 0 or 0000_2. The weighting of each binary digit becomes more important as we study both positive and negative numbers.

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>

0011_2 = ____ 10

How about 1010_2?
2’s Complement

The ‘C6x performs 2’s complement multiply and adds using signed inputs and generates signed outputs. 2’s complement allows the device to perform addition and subtraction using the same hardware: an adder. You can represent any negative value by determining the 2’s complement of the positive number.

2’s complement is simply the 1’s complement plus 1. To find the 1’s complement, invert the bits of the positive value. Then, to generate the 2’s complement, add 1. For example, if you want to find the 2’s complement of 2 (0010), invert the bits (1101) and add 1 $\Rightarrow$ 1110 = -2. You can now add this value to another number.

However, if you plug 1110 into the weighting system shown previously, the result is not -2, but positive 14! What needs to change to obtain the correct result of -2? Actually, the previous model only worked for positive numbers (0-15). The 2’s complement model, though, comprehends both positive and negative numbers. The difference is found in the sign bit. The MSB of the bit range (in our case it’s bit 3 in a 4-bit system) carries the most weight and is also negative. Now, if you plug in 1110, you get -2. The most positive and negative numbers you can represent in this model are 7 (0111) and -8 (1000) respectively.
4-bit Binary Numbers

- Signed (aka 2’s complement)
  - MSB is negative
- Signed vs. Unsigned
  - Same range of precision
  - Signed centered around zero

Unsigned

<table>
<thead>
<tr>
<th>$8$</th>
<th>$0$</th>
<th>$2$</th>
<th>$0$</th>
<th>$= 10_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^3$</td>
<td>$2^2$</td>
<td>$2^1$</td>
<td>$2^0$</td>
<td></td>
</tr>
</tbody>
</table>

Unsigned

|  $1$ |  $0$ |  $1$ |  $0$ |

Signed

|  $-2^3$ |  $2^2$ |  $2^1$ |  $2^0$ | $= -6_{10}$ |
|---|---|---|---|
| $-2^3$ | $2^2$ | $2^1$ | $2^0$ |

Signed (aka 2’s complement)
- unsigned
- signed
- same range of precision
- signed centered around zero

Unsigned

<table>
<thead>
<tr>
<th>$15$</th>
</tr>
</thead>
</table>

Signed

<table>
<thead>
<tr>
<th>$15$</th>
<th>$+7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$15$</td>
<td>$+7$</td>
</tr>
</tbody>
</table>

Unsigned

<table>
<thead>
<tr>
<th>$8$</th>
<th>$0$</th>
</tr>
</thead>
</table>

Signed

<table>
<thead>
<tr>
<th>$8$</th>
<th>$0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

Unsigned

<table>
<thead>
<tr>
<th>$15$</th>
<th>$+7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$15$</td>
<td>$+7$</td>
</tr>
</tbody>
</table>

Signed

<table>
<thead>
<tr>
<th>$8$</th>
<th>$0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

Unsigned

<table>
<thead>
<tr>
<th>$15$</th>
<th>$+7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$15$</td>
<td>$+7$</td>
</tr>
</tbody>
</table>

Signed

<table>
<thead>
<tr>
<th>$8$</th>
<th>$0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

Unsigned

<table>
<thead>
<tr>
<th>$15$</th>
<th>$+7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$15$</td>
<td>$+7$</td>
</tr>
</tbody>
</table>

Signed

<table>
<thead>
<tr>
<th>$8$</th>
<th>$0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8$</td>
<td>$0$</td>
</tr>
</tbody>
</table>
Where Numbers Come From

In a DSP based system, the numbers we work with often come from A/D converters. Using a 4-bit system, here’s an example of how inputs could be translated from reference voltages on the converter to signed hex values for digital processing.
Multiplicative Overflow

We have broken down multiplicative overflow into these topics:

- What is Overflow
- Binary Fractions
- Multiply Example (4-bit)
- Coding Example (16-bit)
- -1 x -1

Definition of Multiplicative Overflow

Multiplication of integers yields results larger than the inputs. How does the user store and reuse these values in subsequent calculations?

Using single digit decimal values as inputs to a multiply can result in a double digit result. Which digit do you store, the upper (1) or lower (2) result? Both, to maintain the integrity of the result, both digits must be kept, resulting in a double precision result.
Solving Multiplicative Overflow

Multiplication overflows can be handled with double precision. However, additional resources are needed to store and reuse the results. Why?

- Because the multiplier cannot handle a double precision input value,
- the double precision value is not easily converted back to an analog signal,
- and additional resources (cycles, words of code and RAM locations) are required to store the value in RAM.

### Solving Multiplicative Overflow

1. Use double-precision result
   - cannot be used as input to multiplier
   - cannot be easily stored to DAC
   - requires additional RAM to store

2. Use fractions to represent input values

Looking ahead, how can the double-sized result be used recursively as an input in later calculations, given that the multiplier inputs are single-width, and after each iteration, the results continues to grow in size? A better solution would be to use fractional numbers.
**Fractional Multiply**

Using fractional numbers allows the user to bias results towards the upper accumulator, allowing the result to be treated equally to the inputs. When multiplying fractional numbers, the product never exceeds the range of a fraction, as can be seen in the example below. As a matter of fact, the values continue to get smaller and smaller.

![Fractional Multiplication Diagram](image)

- **Which digit should be stored?**
- **Result precision equals input’s**
  - Do we lose accuracy?
  - Was error introduced?

Don’t we still have double size results to store? No, we can store just the upper results (.1) and still maintain the integrity of the product. This single precision result can then be used recursively without requiring additional resources. Likewise, the results can be stored with fewer resources and the results are easily converted back to analog.

What about lose of accuracy? Since the least significant bits are being sacrificed, the loss of accuracy is minimal. It is the programmer’s responsibility to decide how this is handled. Rounding or truncation methods can be used.
Binary Fractions

Even integer numbers have a decimal point. It is just ignored by the programmer because it doesn’t ever need to be considered during numerical manipulations. But when dealing with fractions, it is important to know where the decimal point is located.
Just as the analog signal ranges from +5V to -5V, fractional numbers are digitally represented from approximately +1 to -1. Therefore in the same system, using 4 bit binary numbers, the range would be from +7 to -8 in hex representation.

**Binary Fractions - Range**

<table>
<thead>
<tr>
<th>Analog</th>
<th>Digital</th>
</tr>
</thead>
<tbody>
<tr>
<td>positive rail</td>
<td>+5V</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A/D</td>
<td>0</td>
</tr>
<tr>
<td>negative rail</td>
<td>-5V</td>
</tr>
<tr>
<td>0</td>
<td>-8</td>
</tr>
</tbody>
</table>

**Binary Fraction Examples**

<table>
<thead>
<tr>
<th>0 1 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2^0</td>
</tr>
</tbody>
</table>

0110_2 = \_10

1110_2 = \_10
The answers should be ¾ and -¼ to the previous Binary Fractions Examples graphic.

Most systems however use 16-bit hexadecimal numbers. The range for the analog and digital representation would be the same, but you would have more bits to represent the hexadecimal number giving you more precision and a smaller margin of error when dropping off “extra” digits that do not fit into a single precision word.

### Coding 16-bit Fractions (Map into Hex)

<table>
<thead>
<tr>
<th>Fractions</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>~1</td>
<td>7FFFh</td>
</tr>
<tr>
<td>½</td>
<td>4000h</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>-½</td>
<td>C000h</td>
</tr>
<tr>
<td>-1</td>
<td>8000h</td>
</tr>
</tbody>
</table>

For convenience we are using fractions, but the processor still uses 2’s complement (hex)

**Example:** Encode the fraction 0.14

```c
value .short 0x7fff * 14/100
```

or

```c
value .short 0x11eb
```
Binary Multiplication

Here is a simple example of how binary fractional multiplication works.

```
  3/4  0110
x -1/4  1110
   0000
   0110
   0110
  1010
- 3/16 1110100
  reg
```
Sign Extension

Sign extension is used when the results uses less bits than the number of bits in the register. The most significant bit is “extended” out so that the sign and the value of the number does not change.

**Fractional Multiplication**

\[
\begin{array}{c|c}
3/4 & 0110 \\
\times & -1/4 & 1110 \\
\hline
0000 & \\
0110 & \\
0110 & \\
1010 & \\
\hline
-3/16 & 1110100 \\
\end{array}
\]

Where did this extra bit come from?

**Sign Extended**

If loaded into a larger register the bits must be filled, if zeroes are used the value changes.

\[
\begin{array}{cccccc}
1 & 1 & 1 & 0 & 1 & 0 \\
\hline
-2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
\end{array}
\]

Original: \[-8 + 0 + 2 + 0 = -6\]

Zero Fill: \[0 + 0 + 8 + 0 + 2 + 0 = 10\]

Sign Ext: \[-32 + 16 + 8 + 0 + 2 + 0 = -6\]
Storing Product to Memory

<table>
<thead>
<tr>
<th>3/4</th>
<th>0.110</th>
</tr>
</thead>
<tbody>
<tr>
<td>x -1/4</td>
<td>1.110</td>
</tr>
<tr>
<td></td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td>1010</td>
</tr>
</tbody>
</table>
\[\frac{-3}{16} \quad 1110100\]

What is stored in memory?

Put the binary point into our example’s operands to determine what is stored in data memory.

Q Notation

Fractional Multiplication (Q-notation)

<table>
<thead>
<tr>
<th>3/4</th>
<th>0.110</th>
</tr>
</thead>
<tbody>
<tr>
<td>x -1/4</td>
<td>1.110</td>
</tr>
<tr>
<td></td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td>1010</td>
</tr>
</tbody>
</table>
\[\frac{-3}{16} \quad 1110100\]

We use Q-format to keep track of the binary point.
The Q types in this example:

- inputs: \( Q3 \)
- result: \( Q6 \) \((Q3 + Q3)\)
- output: \( Q3 \)
Final Result of Binary Multiply

Fractional Multiplication (Q-notation)

<table>
<thead>
<tr>
<th>3/4</th>
<th>0.110</th>
</tr>
</thead>
<tbody>
<tr>
<td>x -1/4</td>
<td>1.110</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0110</td>
</tr>
<tr>
<td>0110</td>
<td>1010</td>
</tr>
</tbody>
</table>

- 3/16 1110100

reg 11.110100

data memory 1.1 1 0 -4/16

One final question:
Did we get the expected result?
Yes
- 1/4 vs - 3/16 due to truncation
This is the most accurate result.

16-bit Fractional Math

Result is Q30 - How do we Store Q15?

Result: Q30 → Store: Q15

CPU

Q15 sxxxxxxxxxxxxx

x Q15 syyyyyyyyyyyy

Q30 ssszzzzzzzzzzzzzzzzzzzzzzzzz

Data Memory

Q15 szzzzzzzzzzzzzzzzzzzzzzzzz
**Coding Example**

To store our result back into the original resolution (Q15), requires a right-shift by 15 bits, followed by a store-halfword (STH). Since we know our code we can plan for this.

---

**Know the Code?**

<table>
<thead>
<tr>
<th>Q15</th>
<th>MPY</th>
<th>NOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>S.x×x×x×x×x×x×x×x×x×x</td>
<td>A3, A4, A6</td>
<td></td>
</tr>
<tr>
<td>x Q15</td>
<td>S.y'y'y'y'y'y'y'y'y'y'y'y'y'y'y</td>
<td>A3, A4, A6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q30</th>
<th>SHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>S.s.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z</td>
<td>A6, A7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q15</th>
<th>Store to Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>S.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z</td>
<td>A6, A7</td>
</tr>
</tbody>
</table>

---

**Know the Code?**

<table>
<thead>
<tr>
<th>Q15</th>
<th>MPY</th>
<th>NOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>S.x×x×x×x×x×x×x×x×x×x</td>
<td>A3, A4, A6</td>
<td></td>
</tr>
<tr>
<td>x Q15</td>
<td>S.y'y'y'y'y'y'y'y'y'y'y'y'y'y'y</td>
<td>A3, A4, A6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q30</th>
<th>SHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>S.s.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z</td>
<td>A6, A7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q15</th>
<th>Store to Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>S.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z.z</td>
<td>A6, A7</td>
</tr>
</tbody>
</table>
Rounding the Result

Remember when we talked about loss of accuracy or error in our results. You can round up to compensate for this.

Rounding

Add 1 to ? bit then truncate

If ? = 0, no effect (i.e. rounded down)
If ? = 1, result “rounded up”
-1 x -1 = ?

Notice our scale goes almost to +1 but +1 is not on the scale. What if …?

**-1 x -1 = ?**

**The problem:**

-1 x -1 = +1

- +1 isn’t on the scale
- … and there is no hex value to represent +1

Assumes 16-bit numbers

**-1 x -1 Solution**

- The solution is Saturate and Multiply:
  - SMPY
  - SMPYH
- In one cycle, these instructions:
  - Multiply
  - Shift left by 1-bit
  - Saturate if sign bits are “01”

<table>
<thead>
<tr>
<th>Results</th>
<th>MPY (H)</th>
<th>SMPY (H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive Result</td>
<td>00.xxxxx</td>
<td>0.xxxxx</td>
</tr>
<tr>
<td>Negative Result</td>
<td>11.xxxxx</td>
<td>1.xxxxx</td>
</tr>
<tr>
<td>-1 x -1 Result</td>
<td>01.11111</td>
<td>0.111111</td>
</tr>
</tbody>
</table>

**Bottom line:** Use SMPY/SMPYH when multiplying Q-numbers (2’s complement fractions) and MPY/MPYH when multiplying integers.
Handling Multiplicative Overflow in C Code

Now that we’ve seen how to deal with numerical issues with assembly language, let’s see if we can do it using C. By default, C treats everything as integers and knows nothing about fractional math or Q-notation. So, when you multiply a 16-bit number by a 16-bit number in C, you’ll get something like this:

```
short x16, y16, z16;
int z32;

32-bit product
x16 = x16 * y16;
```

This answer will be correct as long as the product did not overflow. If it did, you will get incorrect results from this operation. To make sure that you get the correct results everytime, you need to store the result to a 32-bit data type.
Tell C to keep a 32-bit result by changing the data type of the product. What C actually does in this case is *implicitly* promote the 16-bit operands to 32-bits before the operation. In this case, most programmers feel that it is better to *explicitly* cast the two 16-bit operands to 32-bits.

If you need to store an accurate 16-bit result, you’ll need to use Q-notation within C combined the appropriate casting of the operands and result.
You could also use intrinsics to make this operation even more explicit.

With intrinsics, you can also handle that nasty -1 x -1 problem that we talked about earlier.
Multiplicative Overflow – Summary

Multiplication overflows can be handled with double precision. However, additional resources are needed to store and reuse the results. Thus a better method is to use fractional arithmetic which scales all of the numbers between +1 and -1 and produces smaller more accurate results. We’ve seen how you can do this easily from ASM and C.

1. Use double-precision result
   - cannot be used as input to multiplier
   - cannot be easily stored to DAC
   - requires additional RAM to store

2. Use fractions to represent input values
Accumulative Overflow

Definition of Accumulative Overflow

Using fractions solves the problem with multiplication overflow, but it does not manage overflow during addition or accumulation. When adding fractions, the sum gets larger and larger and can eventually become too large to represent in 16-bits. This can produce faulty result and cause drastic swings in your analog signal.

Accumulative Overflow

\[ f \times f < f, \text{ but what about } f + f? \]
Accumulative Overflow Solutions

You can always test for saturation. It is fairly simple on the C6000. An obvious solution might be to create extra bits to hold the results of accumulations. These bits are called guard bits. And of course you can always design against overflow if your system is bounded and linear.

Dot Product - Accumulative Overflow

1. Saturate the result and Test for saturation
2. Use guard bits
3. Non-gain algorithm (i.e. digital gain ≤ unity)
Saturating the Result

1. Saturating the Result

SADD/SSUB .L src1, src2, dst

If saturation occurs, the SAT bit in the CSR is set to 1

Note: we show 16-bit numbers for convenience.

The result gets saturated (clipped) to 7FFFh. Actually it would be 7FFF FFFFh, but we were using 16-bit numbers since they fit better on the slide.

Besides saturating the result, the SADD and SSUB instructions set the latched SAT bit if saturation occurs.

src1: 7F00h
src2: 300h
ADD: dst = 8200h
SADD: dst =

If saturation occurs, the SAT bit in the CSR is set to 1
Control Status Register (CSR)

- Saturating a result is OK in many systems. Others require test/fix.
- SAT bit is set when saturation occurs while using saturate instructions: SADD, SSUB, SAT, SMPY/H, SSHL
- SAT bit is latched. Stays set until cleared.

<table>
<thead>
<tr>
<th>31</th>
<th>24 23</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU ID</td>
<td>Revision ID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15</th>
<th>10 9 8 7 5 4 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PWRD SAT EN PCC DCC PGIE GIE</td>
</tr>
</tbody>
</table>

Saturating a result is OK in many systems. Others require test/fix. SAT bit is set when saturation occurs while using saturate instructions: SADD, SSUB, SAT, SMPY/H, SSHL. SAT bit is latched. Stays set until cleared.
Clearing the SAT bit

The SAT bit is latched, that is, it stays set until you clear it. How is this done?

**How do you clear SAT?**

SAT bit is cleared by:

1. System RESET
2. Write 0 to CSR\textsubscript{SAT}

```
fixSat: ...
... 
clrSat: MVC .S2 CSR, B0
        CLR .S2 B0, ____, ____, B0
        MVC .S2 B0, CSR
```

This is simple, you might remember the CLR instruction from our first exam:

```
clr.s2 b0,9,9,b0
```

You could use an AND mask but CLR is simpler.

**How do you Test for Saturation (Overflow)?**

At the end of your filter, or whatever algorithm your are using, you will probably want to test the SAT bit and branch to an error/scaling routine if necessary. The Extract (EXT) is perfect for this purpose; it extracts the bit-range (single bit in this example) and places it right-justified in the destination register. **Complete the following EXT instruction:**
How can you use SAT?

- Test for saturation, then ...
- Branch to error/scaling routine

Here’s the completed instruction:

```
ext.s2 b0, 22, 31, b0
```

Some find these values puzzling? The key to understanding them lies in how the EXT instruction works. Think of it as a double-barrel-shift instruction. A *left*, then *right* shift in a single-cycle.

Here it is graphically:
Accumulative Overflow

Extract SAT

\[\text{EXT} .S \ B0, 22, 31, B0\]
\[\text{EXTU} .S \ B0, 22, 31, B0\]

\[csta = 31 - \text{MSB field} = 31 - 9 = \text{22}\]

\[\text{cstb} = \text{csta} + \text{LSB field} = 22 + 9 = \text{31}\]

C6000 Optimization Workshop - Numerical Issues
How Extract Works

Here’s a generic example of EXTract. It demonstrates the ability to extract a range of bits.

![Bit-Field Extract Diagram]
There is also a *dynamic* version of the extract instructions. By dynamic, we mean that the *shift* values are not specified in the instruction itself. Rather, an source register is specified which should contain the shift values at runtime. This allows you to calculate them at runtime.

### Bit-Field Extract (Dynamic)

\[
\text{EXT/EXTU .S src2, src1, dst}
\]

\[
\text{cstb = LSB}_{\text{field}} + \text{csta}
\]

\[
\text{csta = 31 - MSB}_{\text{field}}
\]

### Gaining Headroom (Guard bits)

**Gaining Headroom**

**Fractions**

How can you increase the range?  
(Get more bits?)
Using Guard Bits

Guard bits are 8 extra bits in the companding register that are available to the programmer.

Using the 40-bit Result

SAT instruction saturates 40-bit value to 32-bits

If you store extra bits:
- Additional memory is required
- Values can’t be used as inputs to multiplier
Overflow Allowed by Design

Another scheme for managing accumulative overflow is to design algorithms such that the result of their final summation is guaranteed not to have overflowed. At first, this seems limiting to the designer, and difficult to assure, but it turns out to often be quite easy – especially when one considers that a DSPs inputs and outputs are limited to a fixed number of bits, thus bounding the input and output values automatically.

In a non-gain system, there are several considerations:

BIBO: bounded input, bounded output

What does this mean? It means that you have behavior that is within limits. You will never see an input signal that is greater than a certain amount, nor will you see an output signal that goes beyond a limit.

LTIS: linear time invariant system

What does this mean? A system is linear if and only if the system’s response to the sum of two signals, each multiplied by arbitrary scalar values, is equal to the sum of the system’s responses to the two signals, each multiplied by the same arbitrary scalar values.

A system is said to be time-invariant if, when an input is delayed (shifted) by $n_0$, the output is delayed by the same amount.

### 3. Non-Gain System

If system is bounded and linear, no overflow occurs

For: \[ Y = H(x) \]
Given: \[ |x| < 1 \] (fraction)
Constraint: \[ |Y| < 1 \] (bounded)

Thus, algorithm cannot contain gain

---

Accumulative Overflow

Overflow Allowed by Design

Another scheme for managing accumulative overflow is to design algorithms such that the result of their final summation is guaranteed not to have overflowed. At first, this seems limiting to the designer, and difficult to assure, but it turns out to often be quite easy – especially when one considers that a DSPs inputs and outputs are limited to a fixed number of bits, thus bounding the input and output values automatically.

In a non-gain system, there are several considerations:

BIBO: bounded input, bounded output

What does this mean? It means that you have behavior that is within limits. You will never see an input signal that is greater than a certain amount, nor will you see an output signal that goes beyond a limit.

LTIS: linear time invariant system

What does this mean? A system is linear if and only if the system’s response to the sum of two signals, each multiplied by arbitrary scalar values, is equal to the sum of the system’s responses to the two signals, each multiplied by the same arbitrary scalar values.

A system is said to be time-invariant if, when an input is delayed (shifted) by $n_0$, the output is delayed by the same amount.

### 3. Non-Gain System

If system is bounded and linear, no overflow occurs

For: \[ Y = H(x) \]
Given: \[ |x| < 1 \] (fraction)
Constraint: \[ |Y| < 1 \] (bounded)

Thus, algorithm cannot contain gain
If you really need gain, the least expensive solution is to add it to the analog output:

### What About Gain?

Use analog to add gain
- Practical
- Inexpensive

---

If we can guarantee that the final result won’t “go over the edge” is the problem solved? Yes

What about intermediate results, can’t they overflow? Yes, but if “Y” is bounded, when the summation passes the overflow mark, it must eventually come back

---

Overflowed Intermediate Results
Valid Final Result

---

It’s the Final Result that Counts

---

Valid Final Result

---

I Really Don’t Have to Worry About Overflow?
Accumulative Overflow - Review

Accumulative Overflow

✓ 1. Saturate the result and Test for overflow
✓ 2. Use guard bits
✓ 3. Non-gain algorithm
Numerical Issues – Review Questions

**Exercise**

1. What are three ways to handle accumulative overflow? Which one is preferred?
2. Why is saturation useful?
3. What is the purpose of CSR SAT bit?
4. How is a 40-bit value specified in code (e.g. using registers A2 and A3)?
This page intentionally left blank.