TMS320C54x DSP Design Workshop

Student Guide

DSP54-NOTES-4.02
May 2000

Technical Training

Copyright © 2000 Texas Instruments Incorporated.
All rights reserved.

Notice

No part 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 of Texas Instruments.

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.

Revision History

April 1998, Version 3.1

July 1998, Version 3.2

June 1999, Version 3.3

October 1999, Version 4.0

December 1999, Version 4.01

May 2000, Version 4.02

Welcome to the ‘C54x Workshop

Introduce Yourself

- **A Show of Hands...**
  - **TMS320, C54x, other DSP/µP experience**
  - **C and/or Assembly**
- **About You…(one minute)**
  - **Name, job function**
  - **Expectations**
  - **Which C54x DSP are you interested in using?**

0 - 2

TMS320C54x Workshop Agenda

- **1. Architectural Overview**
  2. Software Development
  3. Addressing
  4. Programming FIR filters
- **5. Numerical Issues**
  6. Solving a Block FIR
  7. Pipeline Implications
  8. Application-Specific Instructions
- **9. Managing Interrupts**
  10. Setting Up and Using Peripherals
- **11. Mixing C and Assembly**
  12. Making a C54x System Work

0 - 3

This is a tentative agenda. Your instructor is free to rearrange and add or delete material as they see fit. If you have any specific interests please make the instructor aware of them.

Your promptness will help keep the workshop on schedule. Try to arrive in the morning and return from breaks on time.

Welcome to the 'C54x Workshop

Administrative Topics

- Name Tags (fill 'em out)
- Start & End Times, Breaks
- Bathrooms
- Phone calls (make or get)
- Lunch
- Let us know if you'll miss part of the workshop

0 - 4

Latest TMS320C54x Information

- Application notes, databooks, etc.:
  `www.ti.com/sc/docs/general/dsmenu.htm`
- Application Software:
  `www.ti.com/sc/docs/apps/index.htm`

0 - 5

Whenever a document is printed on paper it seems it is immediatley out if date. Refer to TI's web resources for the very latest information.

Architectural Overview

Introduction

This chapter will provide an introduction to the TMS320C54x (or “ ’C54x ”). We’ll start by building a device to solve our most important task; high speed multiply – accumulates. Next we’ll zoom in on a feature that greatly enhances our speed; the pipeline. Then we’ll look closely at the ‘C5409, which we’ve selected as the processor for this class. We selected the ‘C5409 since it contains most of the features found on just about any ‘C54x (except multi-core parts). Finally we’ll get a chance to browse the documentation files on the computers. Let’s face it, knowing where to find answers is half the battle.

Learning Objectives

Learning Objectives

- **Describe the basic ‘C54x CPU architecture**
- **Discuss the ‘C54x pipeline phases**
- **List the key features of the ‘C54x memory map and peripherals**

DSP54.1 - 2

Module Topics

Architectural Overview **1-1**

Module Topics *1-2*

Creating a Solution *1-3*

'C54x Block Diagram *1-4*

The Pipeline *1-5*

'C5409 Memory Maps *1-8*

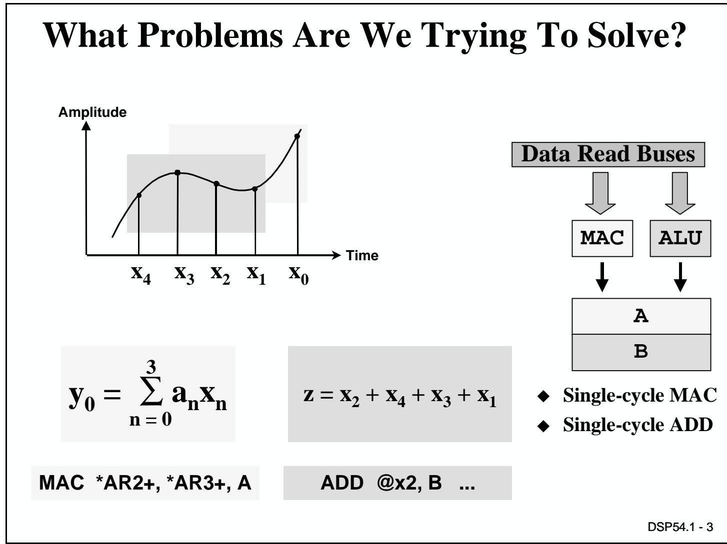
Review *1-11*

LAB 1 – Exploring the Documents **1-12**

Solutions *1-13*

Some Additional Information… *1-14*

Creating a Solution



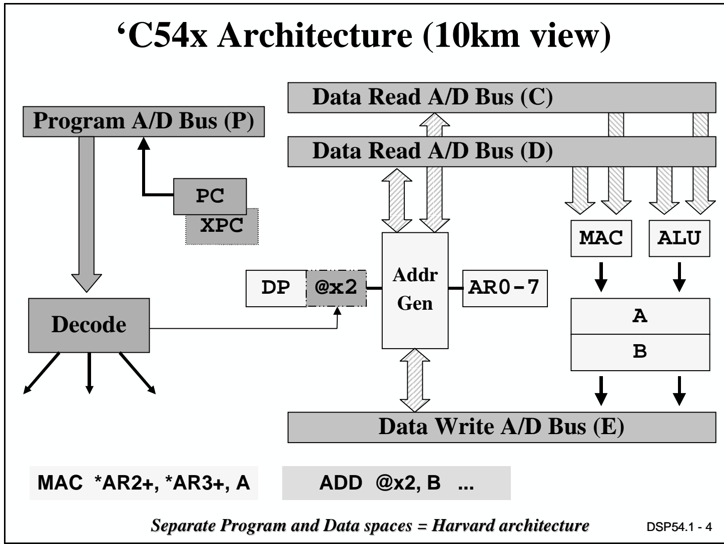
The code you see at the bottom of the slide is a preview if things to come. We'll be dealing with the assembly language code in depth later. For the present let's focus on the architecture.

The multiply-accumulate (MAC) is the basis for DSP. Since just about any physical system can be modeled using a Taylor series, being able to multiply two numbers together and add them to a previous result can be very powerful. The MAC becomes even more powerful if you can do them quickly, say, every 8.3 nS.

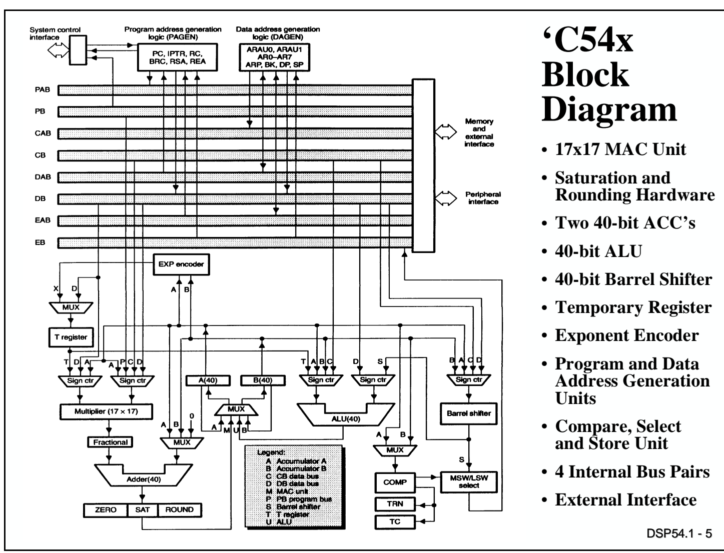
Alternately, other operations require an Arithmetic Logic Unit, capable of performing adds, subtracts and Boolean instructions.

You C-programmers may find the assembly language code on the lower left pretty readable …

‘C54x Block Diagram

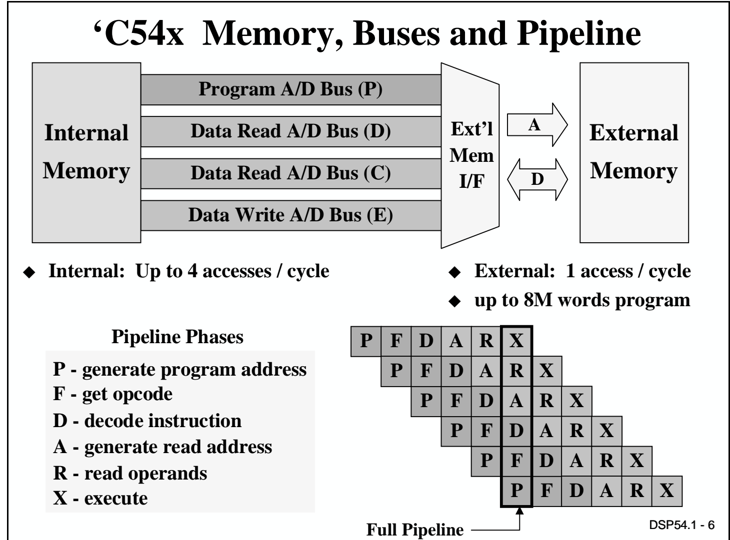


If you’ve got a powerful math engine, your next problem is keeping it fed with data. Separate Program and Data Spaces guarantee access to these areas without conflict. Dual data read busses and a data write bus mean we can access two operands simultaneously and still be able to write a result. Addresses for these operations are generated in a variety of ways. Instructions are fetched using the program bus and are fed to the decoder.



The obligatory block diagram. Your instructor will point out the aspects of this drawing that we’ve already seen. We’ll cover all of its parts in detail during the workshop. You can find a better print of this diagram in the appendix of this workbook.

The Pipeline



The most important thing to understand about the pipeline is that 6 instructions are being worked on simultaneously. This means any given instruction takes at least 6 cycles to reach the execute phase, yet we can "retire" an instruction every cycle. While this allows us a great speed advantage, it has certain implications on operation that we'll deal with in detail later.

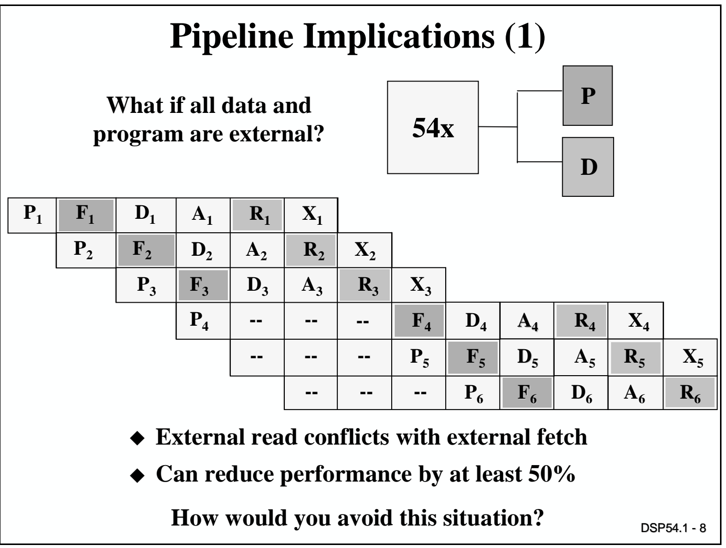
'C54x Pipeline Bus/Hardware Use

P	Generate Program address	$\rightarrow P_A$	PC
F	Get opcode	$\leftarrow P_D$	Program mem
D	Decode instruction		Decoder
A	Generate Read address	$\rightarrow D_A/C_A$	ARs, ARAU
R	Read operands Generate Write address	$\leftarrow D_D/C_D$ $\rightarrow E_A$	Data mem ARs, ARAU
X	Execute instruction Write result	 $\rightarrow E_D$	MAC, ALU Data mem

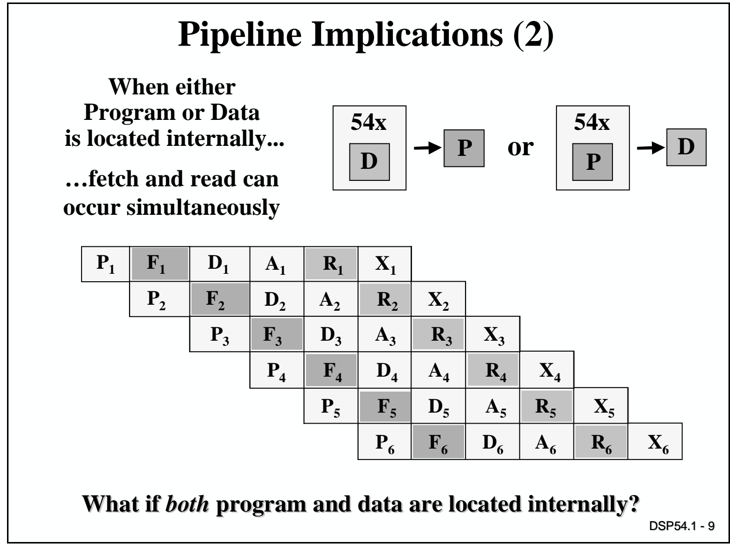
- When storing results back to memory, the "write" is broken into two phases:
  - generating the write address
  - writing the result
- Overlaid onto R & X phases

DSP54.1 - 7

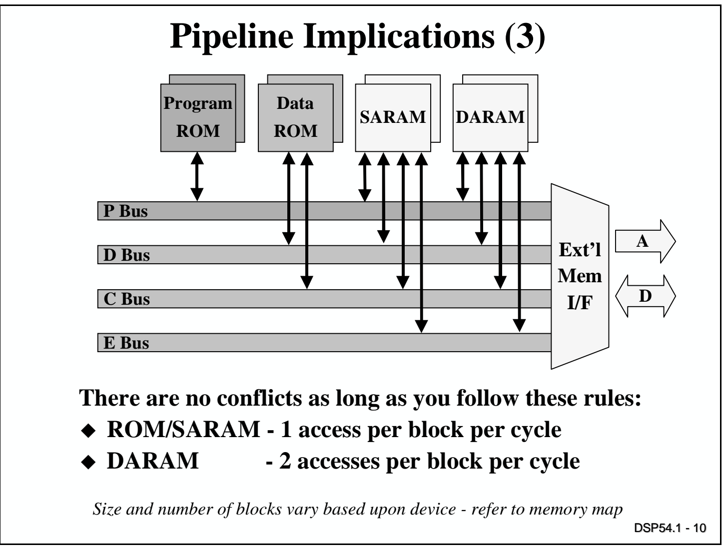
Note how every part of the device operates during every cycle, with little or no overlap of resource utilization. Since writing is a relatively rare event in DSP algorithms it is not allocated its own pipeline stage.



As mentioned earlier, pipelining has implications and the boss won't be happy that we turned a 160 Mhz screamer into a 80 Mhz slug.

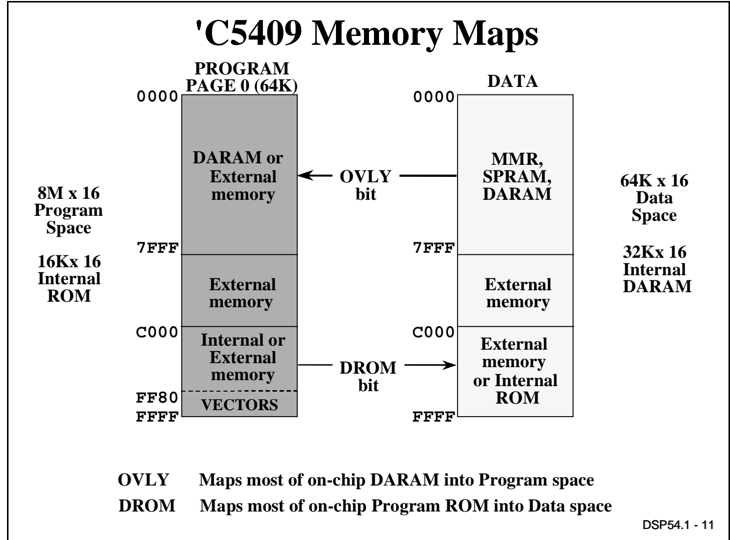


Locating either Program or Data internally will prevent the access problem shown here. Of course, this doesn't take into account delays incurred if your memory is slower than 0 wait-state.



Just how much memory you have available depends on the device you're using. Some devices have no Single-Access RAM and some have no ROM. The number of blocks and the size of each block is also dependent on the device you've selected.

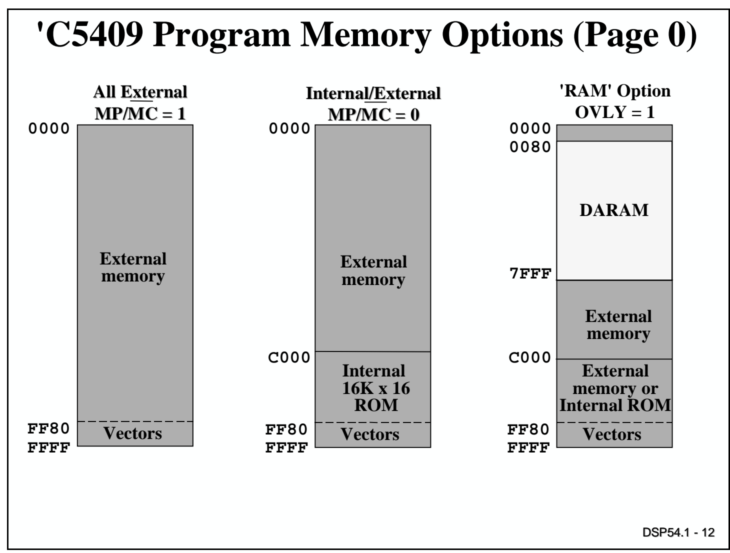
‘C5409 Memory Maps



The OVLY bit maps a portion of data space into program space so that you can load and run a program from data space. While the OVLY bit is set you can also access this area as data.

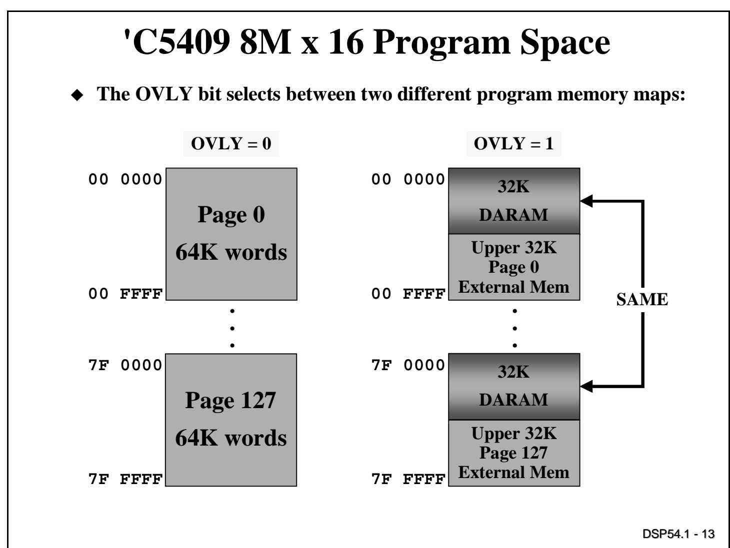
The DROM bit operates similarly, but it maps a portion of program space into data space. This way, non-volatile tables contained in the on-chip ROM can be accessed as data. While DROM is set you can still access this area as program.

The exact size of these areas depends on the device selected.

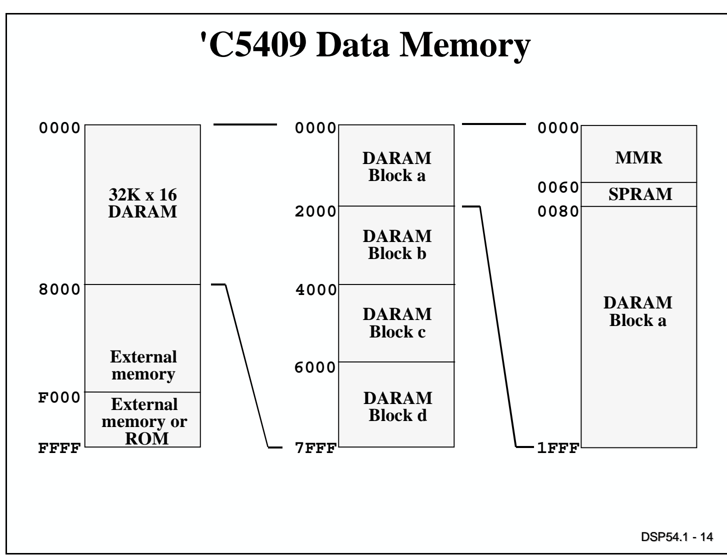


The state of the MP/MC- bit determines whether program memory is located externally or uses the internal ROM (should any exist). All internal ROM is full speed (zero wait state).

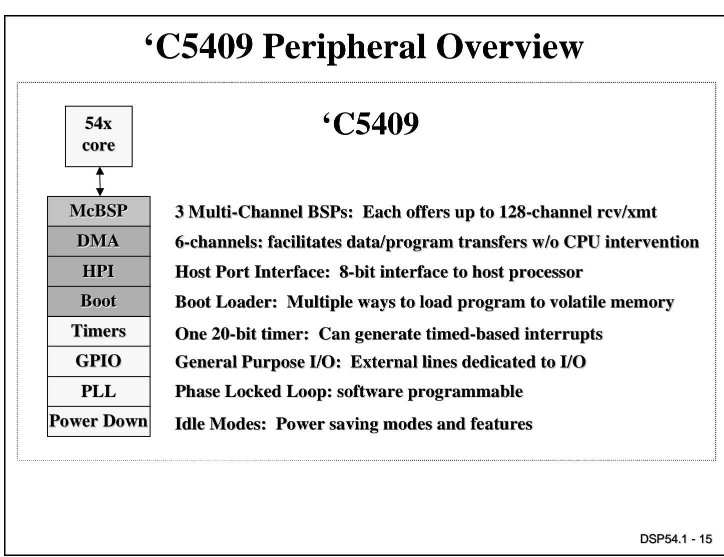
We’ll look at the rest of program space later. For the present, page 0 has the most interesting features.



The OVLY = 0 option allows you access to the entire 8M of Program memory. The OVLY = 1 option gives you fast (not FAR) access to common routines and assembly language programs.



Internal dual access RAM is full speed (zero wait state). You can access any block twice per clock cycle. Locations 00 – 60h contain out Memory Mapped Registers while 60h – 80h contains our Scratch Pad RAM.



All of these “peripherals” are included with the’C5409. There is some sharing of signals and pins.

Review

'C54x Review

- Name the buses on the 'C54x
- How large are the accumulators?
- How many adders are on the part?
- Where are the Memory Mapped Registers located?
- Where is the Reset Vector located?

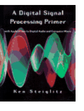
DSP54.1 - 16

Answers to reviews, exercises and labs are always located at the end of each module.

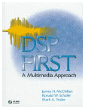
Looking for Literature on DSP?

- "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



- "A DSP Primer : With Applications to Digital Audio and Computer Music" by Ken Steiglitz; ISBN 0-8053-1684-1



- "DSP First : A Multimedia Approach (Matlab Curriculum Series)" James H. McClellan; ISBN 0-1324-3171-8

DSP54.1 - 18

Visit any of the online bookstores for a more comprehensive list of DSP primers. These are just a few that we've encountered.

LAB 1 – Exploring the Documents

LAB 1 - Exploring the Documents

Time: 15 minutes

- **If login is required, Userid/Pswd: DSP54**
- **Double click on the CPU and Peripherals Guide icon (you may find it under "C5400 Manuals")**

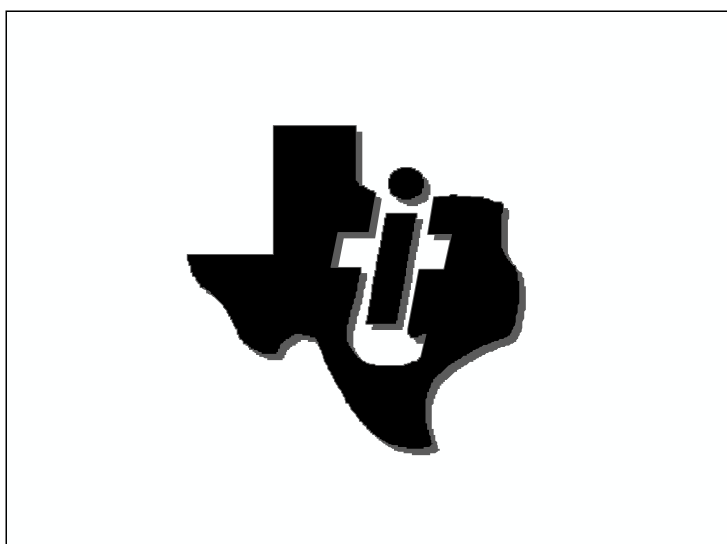
  1. Find the status register diagram for ST0, ST1 and PMST.
  2. Find the Pipeline Latency chapter.
  3. Find the block diagram of the C54x Internal Hardware.
  4. At what address is the reset vector? (after you find it, don't close Acrobat…)

- **Double click on the Mnemonic Instruction Set Guide icon**

  5. In Acrobat, click on "Window" and observe that both guides are available .
  6. How many cycles does a branch (B) take?
  7. Using the bookmarks on the left, find the "MVPD" instruction. What does it do?

DSP54.1 - 19

Probably the most important thing you can learn in this workshop is WHERE to find the answers to your question. With so many databooks and resources the process can be a bit daunting.



This symbol is the "TI-bug". In each module you'll find additional resources "after the TI-bug". If you have specific questions on anything you see, ask your instructor to present or explain the material.

Solutions

'C54x Review - Solutions

- **Name the buses on the 'C54x**
  PA,PD CA,CD DA,DD EA,ED
- **How large are the accumulators?**
  40 bits
- **How many adders are on the part?**
  2, one in the MAC and the other in the ALU
- **Where are the Memory Mapped Registers located?**
  From 0x00 to 0x60 in Data Memory
- **Where is the Reset Vector located?**
  0xFF80 in Program Memory

DSP54.1 - 17

Some Additional Information…

For your reference we've included this list. New devices are being added all the time so this list may be out of date.

'C54x Flavors

	SARAM	DARAM	ROM	Prog. Space	Core Ver.	CVdd	DVdd	Max. Speed
C541		5K	28K	64K	1.0	5V	5V	25nS
C542		10K	2K	64K				
LC541		5K	28K	64K				
LC542		10K	2K	64K				
LC543		10K	2K	64K	1.0	3.3V	3.3V	20nS
LC545		6K	48K	64K				
LC546		6K	48K	64K				
LC541A		5K	28K	64K	2.0	3.3V	3.3V	20nS
LC541B		5K	28K	64K				
LC545A		6K	48K	64K	2.5	3.3V	3.3V	15nS
LC546A		6K	48K	64K				
LC548	24K	8K	2K	8M	2.5	3.3V	3.3V	15nS
LC549	24K	8K	16K	8M	2.5	3.3V	3.3V	12.5nS
VC549	24K	8K	16K	8M	2.5	2.5V	3.3V	8.3nS
VC5402		16K	4K	1M	2.5	1.8V	3.3V	
VC5409		32K	16K	8M		1.8V	3.3V	10nS
VC5410	56K	8K	16K	8M	2.5	2.5V	3.3V	8.3nS
VC5420*	168K	32K		256K	2.5	1.8V	3.3V	10nS

*** dual core*** DSP54.1 - 21

Newest of the New...

	SARAM	DARAM	ROM	Prog. Space	Core Ver.	CVdd	DVdd	Max. Speed
VC5416	128K	?	?	8M	2.5	1.8V	3.3V	6.25nS
VC5421*	2x32K	2x32K +128K	2x2K	256K	2.5	1.8V	3.3V	10nS
VC5402		16K	4K	1M	2.5	1.2V	3.3V	33nS

*** dual core*** DSP54.1 - 22

'C54x Peripheral Mix

	SSP	TDM	BSP	McBSP	DMA	HPI	Timers	PLL	GPIO
C541	2						1	H/W	
C542		1	1			8 bit	1	H/W	
LC541	2						1	H/W	
LC542		1	1			8 bit	1	H/W	
LC543		1	1				1	H/W	
LC545	1		1			8 bit	1	H/W	
LC546	1		1			8 bit	1	H/W	
LC541A	2						1	H/W	
LC541B	2						1	S/W	
LC545A	1		1			8 bit	1	S/W	
LC546A	1		1				1	S/W	
LC548		1	2			8 bit	1	S/W	
LC549		1	2			8 bit	1	S/W	
VC549		1	2			8 bit	1	S/W	
VC5402				2	yes		2	S/W	20*
VC5409				3	yes	8 bit+	1	S/W	26*
VC5410				3	yes	8 bit+	1	S/W	21*
VC5420				6	yes	16 bit	2	S/W	8/42*

+ enhanced ** muxed with McBSP / HPI pins*

DSP54.1 - 23

Software Development Tools

Introduction

The software development environment for TI-DSPs is similar to most microprocessors. It is expected that the user will wish to develop multiple files in parallel, assemble and test them for syntax errors, then create a single executable file by linking the elements together. For the present, we're going to be programming in assembly. If you've only programmed in C, you'll miss things like dynamic memory allocation, but you'll appreciate the performance increase you get by coding in assembly.

Learning Objectives

Learning Objectives

- **Use assembler directives to create sections of code, variables and constants**
- **Create a linker command file to place the allocated sections into a memory map**
- **Create a reset vector**
- **Describe the software tool flow**

DSP54.2 - 2

Module Topics

Software Development Tools 2-1

Module Topics 2-2

Modular Software Development 2-3

Setting Up Hardware 2-4

The Linker Command File 2-5

Vectors.ASM 2-6

Assembly Directives and Data Types 2-7

Software Development Tool Suite 2-8

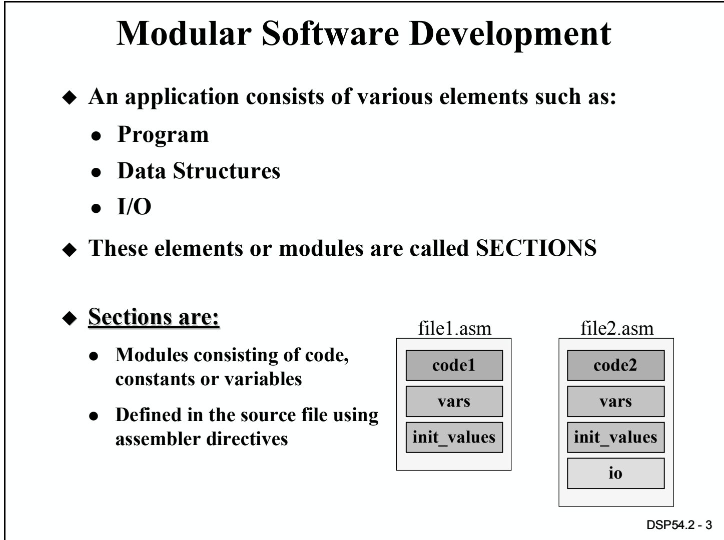
LAB2 – Software Development 2-10
LABx-A vs. LABx-B 2-10
Objective 2-10

LAB2-A Procedure 2-11
Check Code Composer Studio Setup 2-11
Create a New Project 2-11
Edit LAB2A.ASM 2-12
Assemble LAB2A.ASM 2-12
Create VECTORS.ASM 2-12
Assemble VECTORS.ASM 2-13
Edit LAB2A.CMD 2-13
Link LAB2A 2-13
Simulate LAB2A 2-14
Graph Memory Contents 2-16

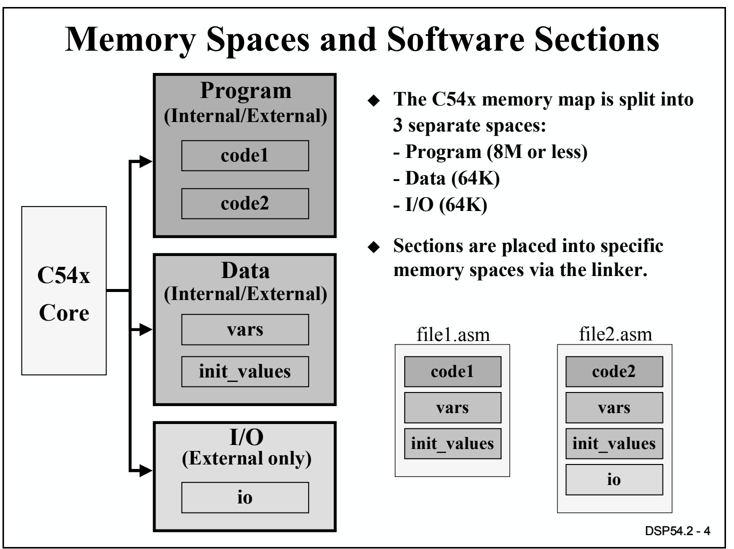
LAB2-B Procedure 2-18

Solutions 2-20

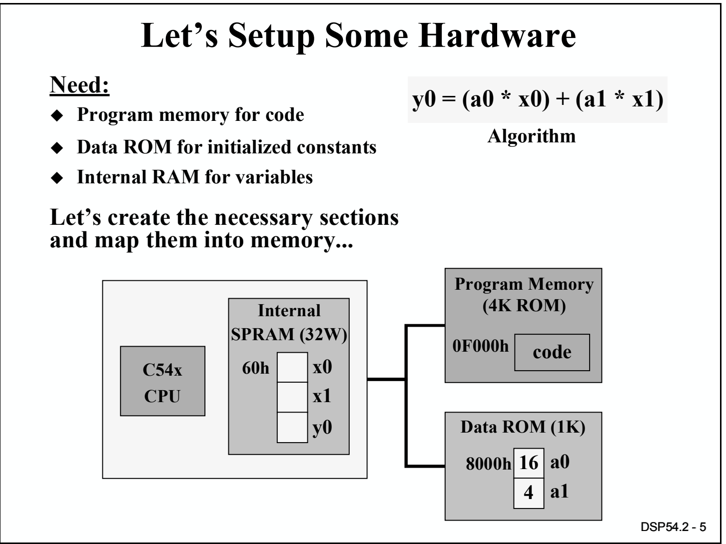
Modular Software Development



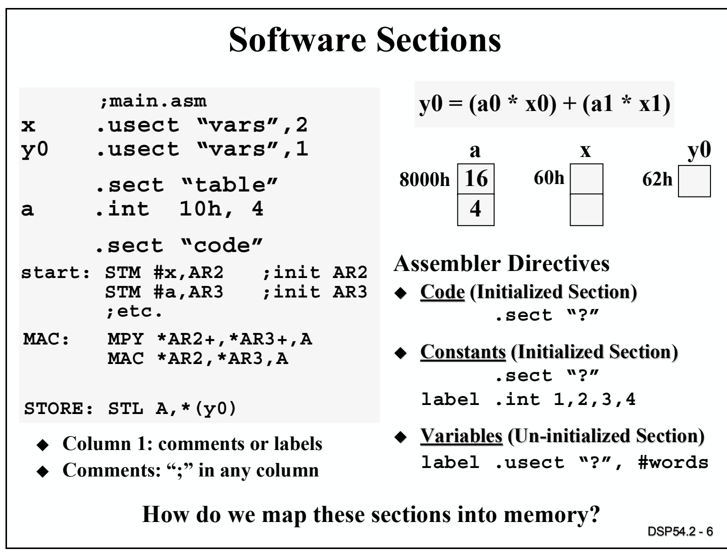
Using a modular development environment allows the user(s) to work on multiple files simultaneously and combine these efforts into a single executable as a final step. To do this, the programmer needs to understand and use “sections” that will allow the linker to correctly map your program elements.



Setting Up Hardware

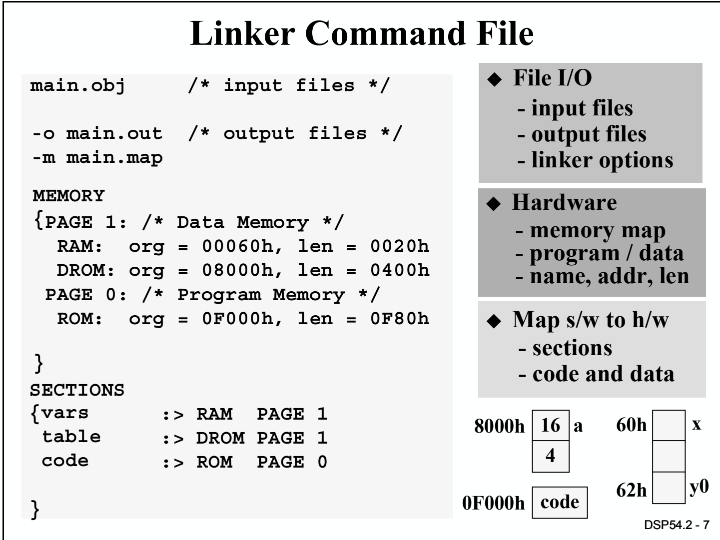


Remember that Program and Data are in separate "spaces" in a Harvard architecture machine like the 'C54x.

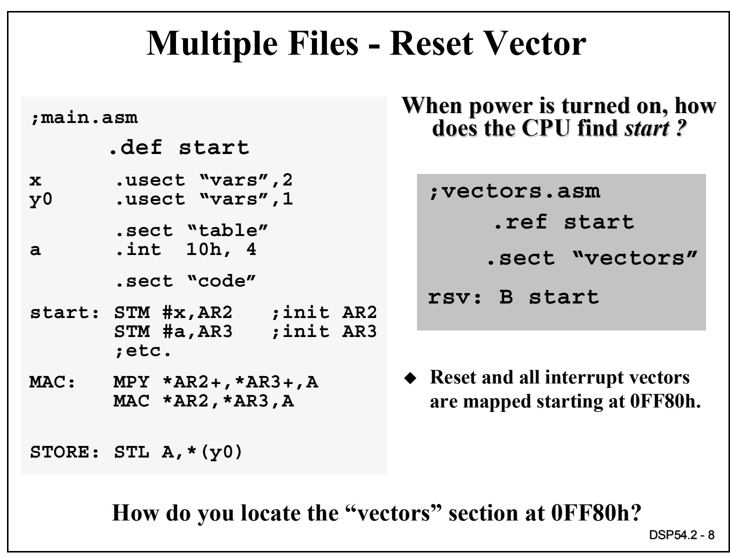


Most, if not all code will contain these basic elements: Initialized Program memory for code, Initialized Data memory for constants and Uninitialized Data memory for your variables.

The Linker Command File



The linker command file is the heart of the development environment. It must understand where your files are and what outputs you wish to generate, what your memory map looks like and where to place your sections within this map. While future tools may eliminate the need to write this file by hand, it will remain important to understand this files construction.



The 'C54x interrupt vectors are "soft". That is, the Program Counter (PC) is directed to 0FF80h and the processor begins "running" code at the indicated location. This is different from a "hard" vector, where only the address to be loaded in the PC is resident. While this a slightly slower than the "hard" vector, it is more flexible.

Vectors.ASM

Linking in Vectors.ASM...

```
main.obj      /* input files */
vectors.obj
-o main.out   /* output files */
-m main.map

MEMORY
{PAGE 1:
 RAM:  org = 00060h, len = 0020h
 DROM: org = 08000h, len = 0400h
 PAGE 0:
 ROM:  org = 0F000h, len = 0F80h
 VECS: org = 0FF80h, len = 0080h
}
SECTIONS
{vars      :> RAM  PAGE 1
 table     :> DROM PAGE 1
 code      :> ROM  PAGE 0
 vectors   :> VECS PAGE 0
}
```

- **Memory spaces cannot overlap**
- **Sections link in the same order as .obj files are listed**
- **Don't forget to assemble ALL input files**

DSP54.2 - 9

The linker command file will need to be instructed exactly where to place the vectors file. This a critically important since we want "`B START`" to land precisely at 0FF80h.

Assembly Directives and Data Types

Basic Directives

Directive	Description
.sect	create <u>initialized</u> named section for code or data
.usect	create <u>uninitialized</u> named section for data
.byte	8-bit constant word-aligned
.int (.word)	16-bit constant
.long	32-bit constant
.ref/.def	used for symbol references
.global	.ref and .def combined
.set/.equ	equate a value with a symbol*
.asg	assign an assembly constant*. Will display in debugger

Data Types

Value	Type
10	Decimal (default)
0Ah, 0xA	Hex
1010b, 1010B	Binary

** takes no memory space*

DSP54.2 - 10

You may wish to recreate batch files like these on your own PC.

Assembly Directives and Data Types

Assembly Directives and Data Types

Basic Directives

Directive	Description
.sect	create initialized named section for code or data
.usect	create uninitialized named section for data
.byte	8-bit constant word-aligned
.int (.word)	16-bit constant
.long	32-bit constant
.ref/.def	used for symbol references
.global	.ref and .def combined
.set/.equ	equate a value with a symbol*
.asg	assign an assembly constant*. Will display in debugger

Data Types

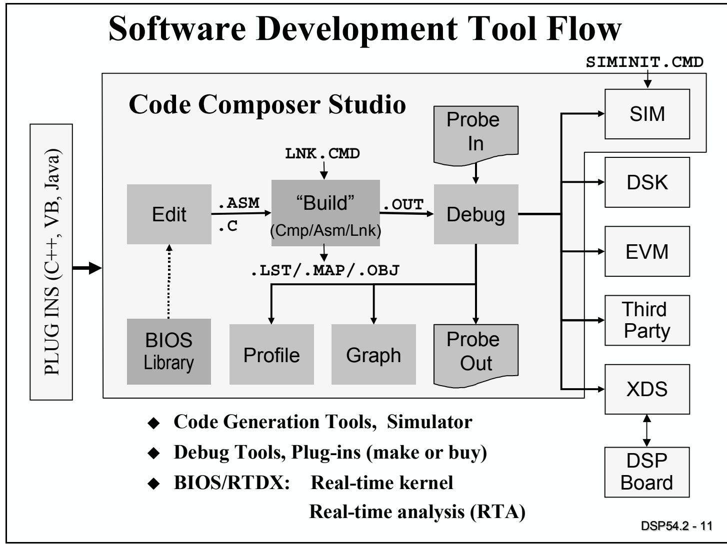
Value	Type
10	Decimal (default)
0Ah, 0xA	Hex
1010b, 1010B	Binary

** takes no memory space*

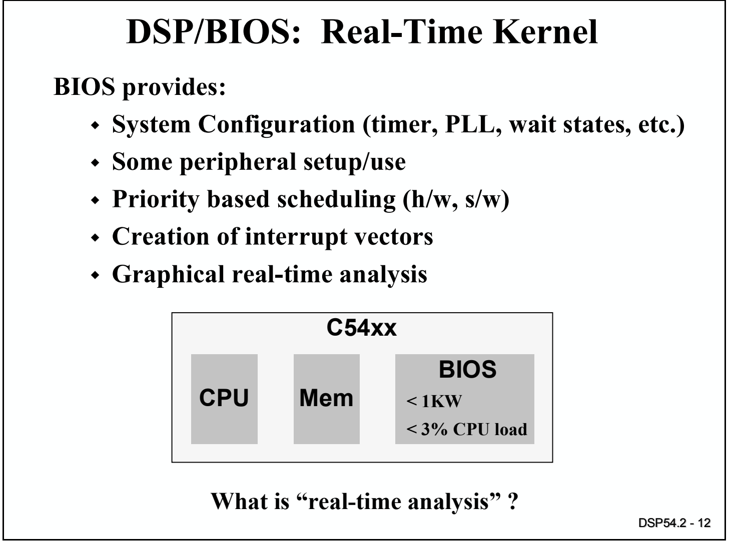
DSP54.2 - 10

There are more directives and types than can be listed here. Refer to the Assembly Language Tools User Guide for more information.

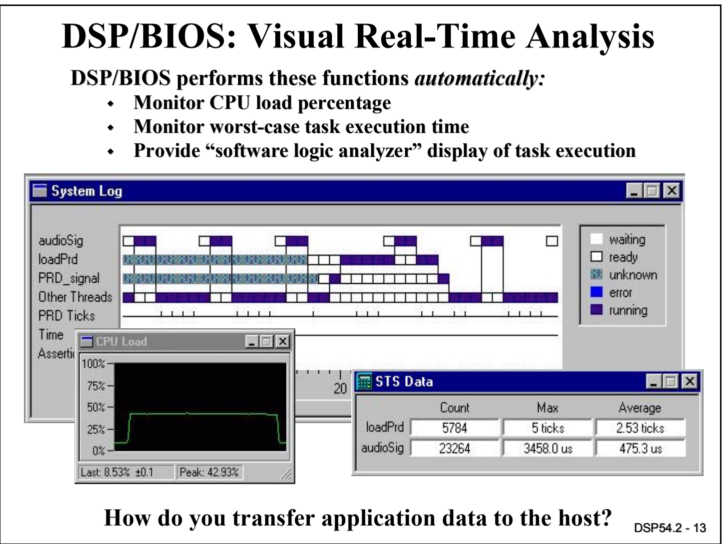
Software Development Tool Suite



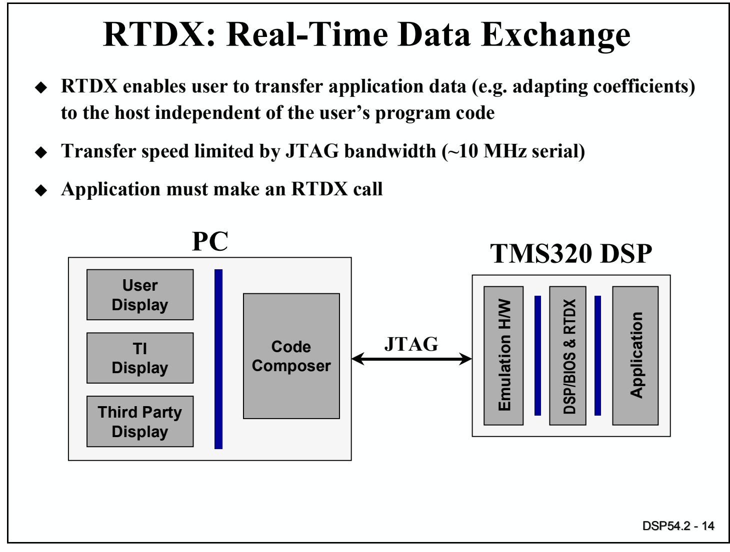
TI has developed an open toolset. Users or 3rd party developers can write code that can be "plugged into" the Code Composer Studio (CCS) tool suite. In addition, on-chip elements like BIOS and RTA interface directly through the JTAG port to CCS.



You get BIOS and it's source code free on newer C54 devices.



Before this tool, determining whether or not your system met real-time goals was a hit or miss proposition. With RTA you can see it directly.



RTDX speeds the usual JTAG port by transmitting a subset of the normal information.

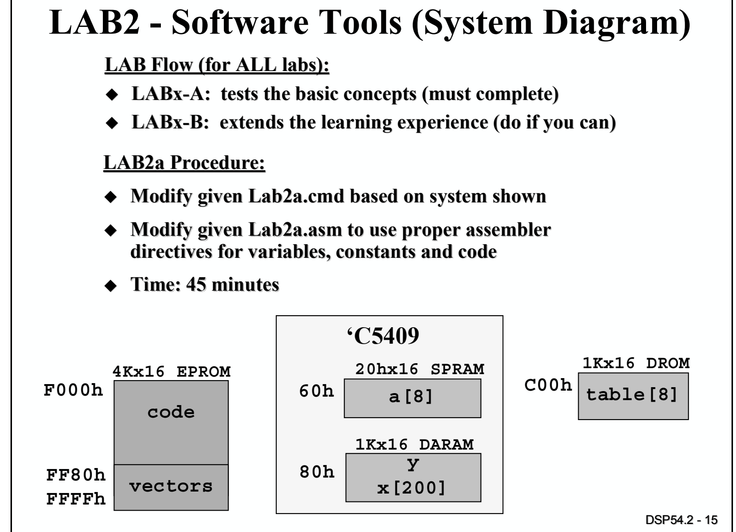
LAB2 – Software Development

LABx-A vs. LABx-B

Each lab contains two labs: part A and part B. Part A will test the basic skills learned in the module. It is important that you spend as much time as necessary to complete part A. If you finish part A and would like to continue challenging yourself and exploring more details, move right on to part B. Part B of every lab contains much more information than you can actually complete in the allotted time. However, any time spent on part B will enhance your understanding of the processor.

Objective

The objective of this lab is twofold: (1) edit the given link.cmd file based on the system diagram; (2) add the proper assembler directives to the given file (`LAB2A.ASM`) to allocate sections for code, constants and variables.



LAB2-A Procedure

For you people suffering from hexaphobia, here's some help:

1K	0400h
4K	1000h
8K	2000h
16K	4000h
32K	8000h
48K	C000h
64K	10000h

Check Code Composer Studio Setup

1. Before we get started, let's make sure Code Composer Studio (CCS) is setup to run on the C54x simulator. Double click on the CCS Setup icon on the desktop. When the "`Import Configuration`" window appears click the `Clear System Configuration` button. When prompted if you are sure click `Yes` the close the "`Import Configuration`" window.

2. The middle pane displays the available platforms that can be installed. Double click on the `C54xx Simulator` to install it.

3. A window displaying `Board Properties` will appear. Click `Next`. In the next window select `sim549.cfg` as the simulator config file (click the **...** button). The C5409 is currently not a selection, but the `C549` has the same memory map.

4. Click `Finish` and close the CCS Setup window. When prompted to save changes to the system configuration click `Yes`.

Create a New Project

1. Double click on the CCS icon on the desktop. Maximize CCS to fill your screen. The menu bar (at the top) lists File ... Help. Note the horizontal tool bar below the menu bar and the vertical tool bar on the left-hand side. The window on the left is the project window and the large right hand window is your workspace.

2. A *project* is all the files you'll need to develop an executable output file (`.OUT`) which can be run on the simulator or target hardware. Let's create a new project for this lab. On the menu bar click:

   `Project → New`

   and make sure the "SAVE IN" location is: `C:\DSP54\LABS` and type `LAB2A` in the file name window. This will create a *make* file which will invoke all the necessary tools (assembler, linker, compiler) to build your project.

3. Let's add the assembly file to the new project. Click:

   `Project → Add Files to Project`

and make sure you're looking in `C:\DSP54\LABS`. Change the "files of type" to view assembly files (`.ASM`) and select `LAB2A.ASM` and click `OPEN`. This will add the file `LAB2A.ASM` to your newly created project.

4. Add `LAB2A.CMD` to the project using the same procedure.

5. In the project window on the left click the plus sign (+) to the left of `Project.` Now, click on the plus sign next to `LAB2A.MAK`. Notice that the `LAB2A.CMD` file is listed. Click on `Source` to see the current source file list (i.e. `LAB2A.ASM`).

Edit LAB2A.ASM

6. To open and edit `LAB2A.ASM`, double click on the file in the project window. The code you see in this file is not related to the setup you will create in the following steps. This code is simply a place holder for future labs and doesn't do much.

7. Create uninitialized sections for *a* called *coeffs* and *y* called *result*.
   Refer to the system diagram for the sizes. Remember the format is:

```
label:  .usect "section_name",size
```

8. `LAB2A.ASM` already contains the values that need to be allocated in *table[8]*. 16 values are given, but 8 of them are commented out. The 2nd set of 8 values will be used in future labs. Define an initialized data section called *init* for the values and place a label (*table*) next to the first 4. *Table[8]* should contain the following values:

```
7FCh, 7FDh, 7FEh, 7FFh, 800h, 801h, 802h, 803h
```

9. Create an initialized program section for code. Add a label definition for the beginning label of the code (*start*). Save your changes by clicking the disk on the horizontal tool bar "`Save`".

Assemble LAB2A.ASM

10. Assemble `LAB2A.ASM` by clicking on the top button on the vertical toolbar. When your mouse hovers over this button, you will see the words `Compile File` and check for errors before moving on to the next step. If you get an assembly error, scroll the `Build` window at the bottom of your screen until you can see the error and simply double-click the error shown in red. Your cursor should now be positioned at the start of the line with the error in your assembly file. Save any changes you made before going on.

Create VECTORS.ASM

11. Create a new file by clicking on the left most button on the horizontal toolbar "New".

12. Add an initialized code section named `"vectors"` that contains: `rsv: B start`
    Make sure that both labels (*start* and *rsv*) are visible to the linker. Save your file by clicking on the `Save` button on the horizontal toolbar. When prompted, save your file and name it "vectors" as type "Assembly Source File" in the `C:\DSP54\LABS` directory.

13. FYI: a complete 'C54xx instruction summary is available by clicking:

```
Help → DSP Instructions
```

on the menu bar. If you need help with a specific instruction you've already typed, highlight the instruction with your mouse and hit `<F1>`. Try it now. The workbook appendix also contains a complete list of instructions for the 'C54xx. Feel free to use either reference in this and future labs.

Assemble VECTORS.ASM

14. Assemble `VECTORS.ASM` by clicking on the compile button as you did before to assemble `LAB2A.ASM`. Check for errors before moving on. Save your work.

15. Add `VECTORS.ASM` to the project using the procedure shown earlier.

Edit LAB2A.CMD

16. To open and edit `LAB2A.CMD`, double click on the filename in the project window.

17. Setup the file I/O and any options necessary to create map and output files. Don't forget to link in the vectors file (`vectors.obj`). Add an option: `-e rsv` to define an entry point for the simulator.

    NOTE: You can delete the entire file I/O section of your linker command file and perform the same functions in CCS under `Project, Options, Linker`. All files added to the project will be linked. Try linking your project using the CCS options and use the method you prefer.

18. Edit the `Memory{}` declaration by describing the given system diagram's memory map.

19. Place the sections defined in `LAB2A.ASM` and `VECTORS.ASM` into the appropriate memories via the `Sections{}` area. Save your work.

Link LAB2A

20. Setup the linker options by clicking:

    `Project → Options`

    on the menu bar. In the middle of the screen select "`No Autoinitialization`", then `OK`. We will cover the other options shown during the module on compiling C code. To open up more work space, close any open files that you may have.

21. FYI: CCS can automatically load the output file into the simulator after a successful build. On the menu bar click:

    `Option → Program Load`

    and select: "`Load program after build`", then click `OK`.

22. The top four buttons on the vertical toolbar control code generation. Hover your mouse over each button as you read the following descriptions:

Button	Name	Description
1	Compile File	Compile, assemble the current open file
2	Incremental Build	Compile, assemble only changed files, then link
3	Rebuild All	Compile, assemble all files, then link
4	Stop Build	Stop code generation

23. Before we build and load the program we need to initialize the simulator. We have included a GEL file to do this for you. GEL files allow the user to automate repetitive procedures. On the menu bar click:

    `GEL → C54x → C549_Init`

    You should initialize the simulator every time before you build/load a program.

24. Click the "`Rebuild All`" button and watch the tools run in the build window. Debug as necessary. Right-click on the build window and Hide the build window.

25. Open and inspect `LAB2A.MAP`. This file will show you the results of the link process. Close the file when you are done.

26. If code generation is successful, a window displaying the `VECTORS.ASM` source file with and a yellow highlight on "B start" should appear. This indicates that you are now ready to simulate.

Simulate LAB2A

27. FYI: Should you experience a problem with CCS, quit, then restart , reload your project and program .

28. As you can probably tell, the windows in the simulator can be moved around and resized. Typically, the default window arrangement is not a desirable one. To customize your display, move the windows around where you want them. You also may want to right click on each window and select: `Float in Main Window`. This will allow each window to be visible when it is active.

29. Right click on the project window and select: `Hide`

30. You'll probably want to see the CPU registers. On the menu bar click:

    `View → CPU Registers → CPU Registers`

31. Right click in the `CPU Registers` window and deselect "`Allow Docking`". You can now move and resize the window as you like. Close the CPU Register window. Locate the "`Register Window`" button on the vertical toolbar, then click it to see if it appears.

32. If you're familiar with the Command Window used by previous TI simulators, you can add one by clicking:

    Tools → Command Window

on the menu bar. Resize and dock or undock to your liking.

33. If you prefer to see the disassembly window, find and click the "View Disassembly" button (at the bottom of the vertical toolbar) or click: `View` → `Dis-Assembly` on the menu bar.

34. You can save your workspace by clicking:

    `File` → `Workspace` → `Save Workspace`

    and selecting a name. Make sure you save it in `C:\DSP54\LABS`. DO NOT save your new workspace as the "default". When you restart CCS, you can reload "your" workspace by clicking:

    `File` → `Workspace` → `Load Workspace`

    and select your filename.

    You may want to save a "generic" workspace rather than one that opens up a project. Make sure that you close the project before you save this workspace.

35. You can edit the contents of any CPU register by double-clicking on it. Try this with AR7 now. Try typing in both hex and decimal numbers. Note that CCS will convert decimal to hex for you.

36. Hit the `<F8>` key or click the single step button on the vertical toolbar repeatedly and single-step through the program, watching the values in the accumulators change. Notice the other step buttons as well: `Step Over` (a function call) and `Step Out` (of a function or subroutine).

37. At the command line, type:

    `Step 20↵`

    and watch the simulator actions. You should see the screen update to reflect the results of each individual "step".

38. Type: `Run↵` or F5 or click the Run button on the vertical toolbar. Notice the words "DSP Running" in the bottom left-hand corner of your screen. If something isn't working properly or the simulator seems like it is stuck, always check this location first to see the status of the simulator.

39. Hit Shift-F5 or the Halt button on the vertical toolbar to stop the simulator. Notice the words "DSP Halted".

40. Type: `Run 20↵`

    This will *not* update the display until the specified time is complete.

41. Type: `Reset ↵` or click `Debug` → `Reset DSP` from the menu bar.

    Notice that reset takes you back to the reset vector located at 0FF80h. Restart, on the other hand, will return you to the entry label you set up by using the *–e* option in the linker command file. Edit your linker command file to set –e to *start* and rebuild the project. Notice that when your program loaded, the simulation begins at the *start* label. Now type reset on the command line and watch the simulation return back to the reset vector location.

42. Now type: `go LOOP ↵`

    Note that labels are case sensitive.

43. Try: `go loop↵`

    and see if that helps.

44. On the menu bar click:

    View → Memory

    or click the View Memory button on the vertical toolbar. Type "table" into the address to display the contents of the memory starting at label *table*. Do the same for label "a". You can display as many independent windows as you require. Do you see your initial values in the memory window displaying "table"?

    Note that by double-clicking on any location you can edit the contents of the memory location.

45. The numbers in "table" are actually signed fractions with values of about 1/16th. In the memory window, displaying "table", right-click and select `Properties`. Select a Q value of `15` and "`16-bit signed int`" format. In later modules you'll see what Q values represent.

46. We can set a watch on a variable. Type: `wa *y ↵` in the command window or click the "`Watch Window`" button on the vertical toolbar or select `View → Watch Window` from the menu bar. Right click in the watch window and select "`Insert New Expression`" and type: *y. This will display the contents (*) of the location y.

    If a watch fails to display, the variable may be unavailable to the debugger. Make sure you used the `-s` switch when assembling, or, declare *y* as global using `.def` or `.global`. Either method will ensure that the debugger recognizes the name.

    Note that you can have up to 4 watch windows open where you can group your watches.

47. Type: `wd *y↵` to remove the watch or right-click on *y and select "`Remove Current Expression`".

48. From the menu bar click:

    `Profiler → View Clock`

    to watch the system clock. Click:

    `Profiler → Enable Clock`

    to allow the clock to run. Double-click on Clock in the Profile Clock window to zero it. Step through your code and watch the display accumulate the cycle count.

Graph Memory Contents

49. Located in the `C:\DSP54\LABS` directory is a file called `IN6.DAT`. It contains initialized values for a sine wave created by adding low and high frequency sine waves together. You can open the file and view its contents if you like. This data will be the input to the filter we will design in future labs, so we need to add it to our assembly source file.

50. You should be able to see the `LAB2A.ASM` source file on the screen. Add an initialized section called "`indata`" with a label pointing to its first location called "`x`". After this line type:

    ```
    .copy in6.dat
    ```

51. Now add the `indata` section to the DARAM data memory in `LAB2A.CMD`.

52. Save your changes and rebuild the project.

53. Click on the View memory button on the vertical toolbar and type "x" in the address field. You should see lots of data displayed, but it sure would be nice to see this as a graph.

54. Select:

    `View → Graph`

    on the menu bar and pick "Time/Frequency". Change the following fields to reflect the information shown below:

Graph Title:	`Input Data`
Start Address:	`x`
Acquisition Buffer Size:	`200`
Display Data Size:	`200`
DSP Data Type:	`16-bit signed integer`
Q type:	`15`

    Click OK to see your graph. Resize it to your liking.

55. We might also want to see the plot as a frequency display. Right-click on the graph, select `Properties`, and change the `Display Type` field to "`FFT Magnitude`". Then click OK to view the plot.

56. If you're finished with part A of this lab and you have some more time, move on to part B on the next page.

LAB2-B Procedure

Further details about each of the following commands are in the C Source Debugger User's Guide. You might try locating this .pdf file on your PC and look through it. You can obtain the answers to any of these questions by looking in the on-line help guide or asking your instructor.

1. The simulator supports connecting a file to any address to allow file i/o. Look up how in the CCS help file and familiarize yourself with them.

2. Create a custom command string using the `ALIAS` command. Alias can be used to rename any command (to avoid lots of typing) or to combine several commands into one user-defined name. For example, type:

   ```
   alias r, restart ↵
   ```

   Look up the ALIAS command in the debugger guide and read more about it.

3. Use the FILL command to initialize a block of memory. Look on the menu bar under Edit ... memory .. fill.

LAB Debrief

1. What was the most difficult aspect of using the linker?

2. What kind of syntax errors did you get when you assembled your code?

3. When using the simulator, were you able to view the contents at address *table* and *a*?

4. What did you learn?

5. Did the lab procedure provide CLEAR directions?

6. Did anyone get to part B of the lab?

DSP54.2 - 16

Solutions

LAB2A.ASM - Solution

```
; allocate label definition here
        .def start

; allocate uninitialized data sections here
a       .usect "coeffs",8
y       .usect "result",1

; allocate initialized data sections here
; only the first 8 values are used in Labs 2a and 3a
        .sect "init"
table   .int 7FCh,7FDh,7FEh,7FFh
        .int 800h,801h,802h,803h
;       .int 803h,802h,801h,800h
;       .int 7FFH,7FEH,7FDH,7FCH

        .sect "indata"
x       .copy "in6.dat"

; allocate code section here
        .sect "code"
start:  LD      #0,A
        LD      #0,B
loop:   ADD     #1,A
        ADD     A,B
        ADD     #1,B
        ADD     B,A
        B       loop
```

DSP54.2 - 18

LAB2A.CMD - Solution

```
/* file I/O and options */
vectors.obj
lab2a.obj
-m lab2a.map
-o lab2a.out
-e start

MEMORY {
PAGE 1:        /* Data memory */
  SPRAM:    org = 00060h,  len = 00020h
  DARAM:    org = 00080h,  len = 00400h
  DROM:     org = 00C00h,  len = 00400h

PAGE 0:        /* Program memory */
  EPROM:    org = 0F000h,  len = 00F80h
  VECS:     org = 0FF80h,  len = 00080h
}

SECTIONS
{
  coeffs    :> SPRAM   PAGE 1
  result    :> DARAM   PAGE 1
  init      :> DROM    PAGE 1
  indata    :> DARAM   PAGE 1
  code      :> EPROM   PAGE 0
  vectors   :> VECS    PAGE 0
}
```

DSP54.2 - 19

Introduction

In a machine that can perform as many as 160 million multiply-accumulates per second, getting data to and from the computational units is of critical importance. Different tasks access data differently, so the ‘C54x incorporates addressing modes to perform those tasks at the fastest possible speed.

Learning Objectives

Objectives

- **Understand the 5 basic addressing modes and why to use them**
- **Perform an exercise using these modes**
- **List the remaining addressing modes**

DSP54.3 - 2

Module Topics

Addressing Modes **3-1**

Module Topics *3-2*

The Need For Addresses *3-3*

A Review *3-4*

Generating Data Addresses *3-5*

Indirect Addressing *3-6*

MMR Addressing *3-7*

Direct Addressing *3-8*

Immediate Addressing *3-9*

Direct Addressing … A How-To *3-10*

Absolute Addressing *3-11*

What have we missed? *3-12*
MMR Issues 3-12
A List of Indirect Addressing Options 3-13
Direct Addressing Issues 3-14
Some Definitions 3-14

Review *3-15*

Exercise *3-16*

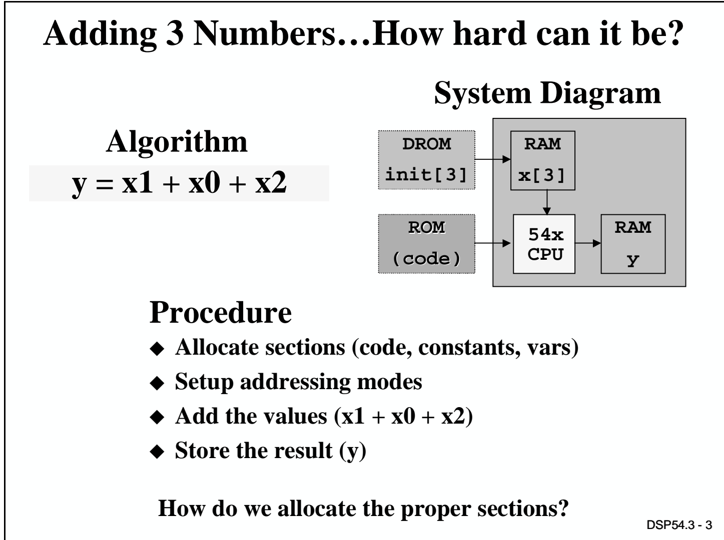
LAB3 – Addressing **3-17**
Objective 3-17

LAB3-A Procedure *3-18*
Copy Files, Create Make File 3-18
Copy table[8] to a[8] – Write/Debug 3-18
Add the values, Store result to `y` – Write/Debug 3-19
Profile Your Code 3-20

LAB3-B Procedure *3-21*

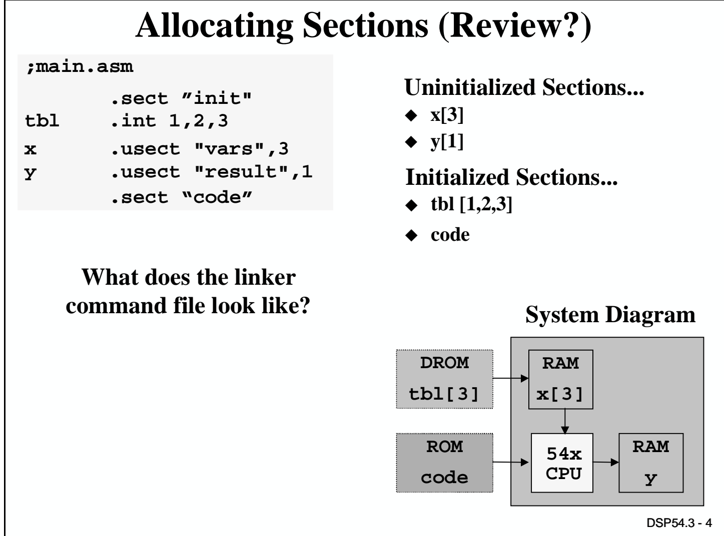
Solutions *3-23*

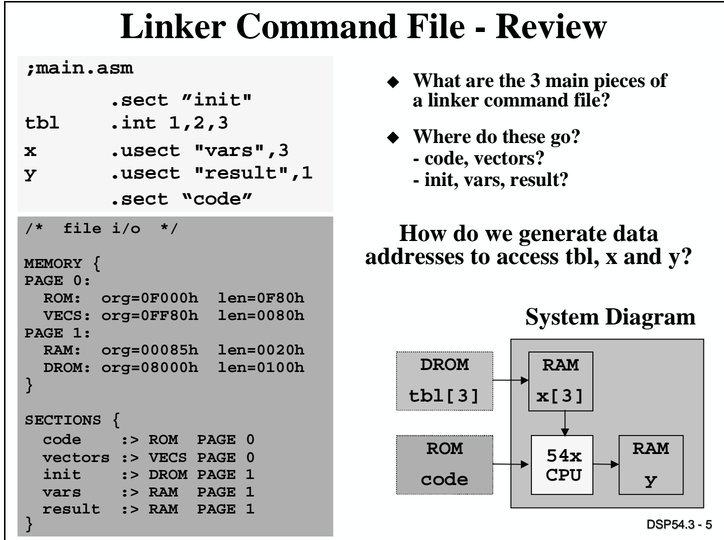
The Need For Addresses



We've already done this, so this should be merely a review of the techniques covered in module 2.

A Review





Generating Data Addresses

Generating Data Addresses

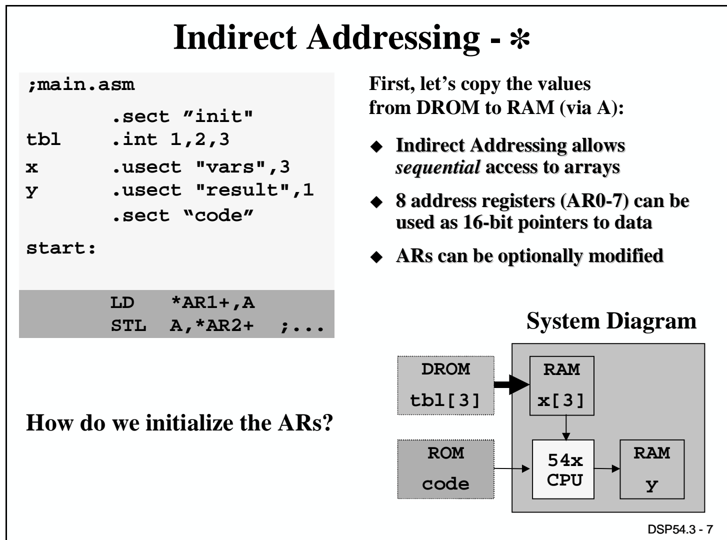
The 'C54x uses 5 basic data addressing modes:

Mode	Description
Indirect	Uses 16-bit registers as pointers
Direct	Random access from a specified base address
Absolute	Specify entire 16-bit address
Immediate	Instruction contains the data operand
MMR	Access memory mapped registers

DSP54.3 - 6

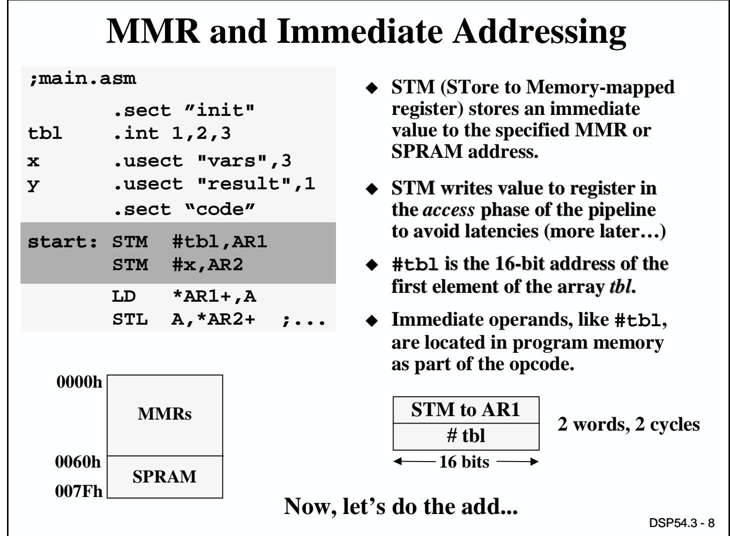
Generating Data addresses is what the programmer will be doing most. How can it be done at the fastest possible speed?

Indirect Addressing



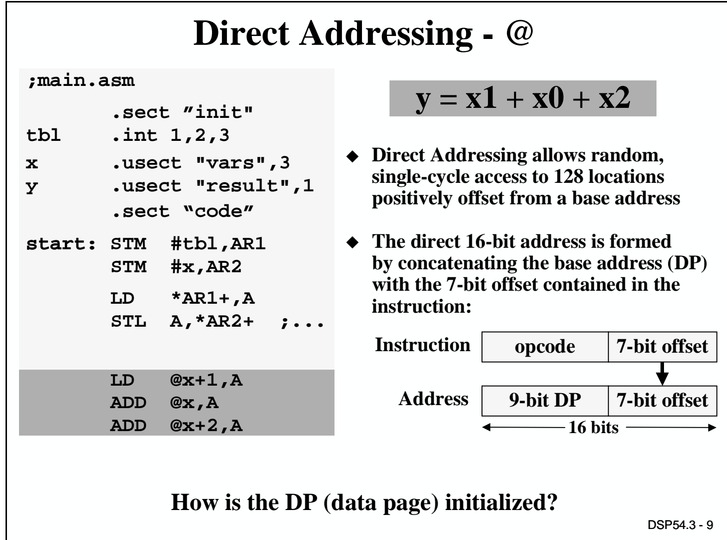
Indirect addressing is typically used for addressing arrays of information where stepping up or down through the data is necessary. We'll cover the modifications a little later.

MMR Addressing



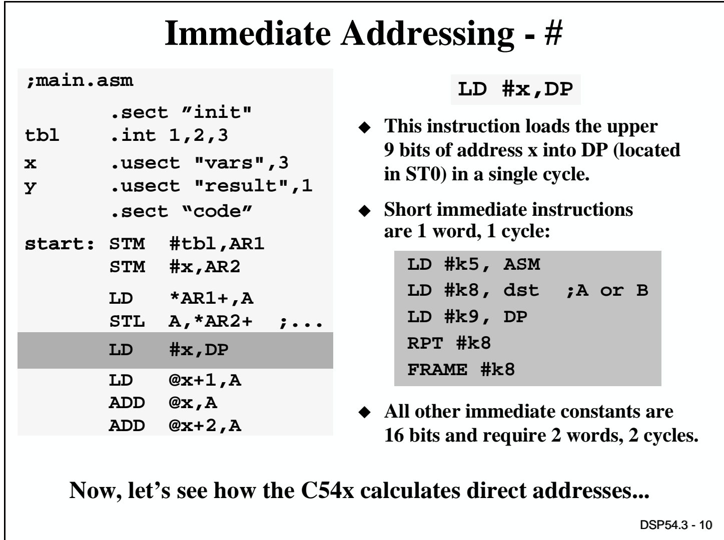
MMRs contain the information needed to control the ‘C54x functions and you’ll need to program them periodically. MMR addressing gives you a method to easily access those registers. Additionally, since writing to control registers can incur latencies, instructions like `STM` operate early and avoid most pipeline problems.

Direct Addressing



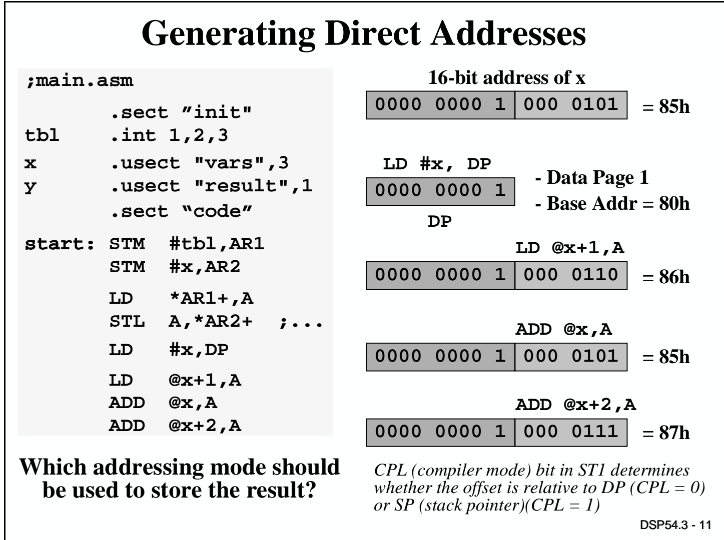
If you need to randomly access some variables, indirect addressing would be a poor choice since you'd need to reprogram the AR before each access. Direct addressing allows us random access into a "page" of memory. Why doesn't direct addressing access the entire data space? Because instructions in the 'C54x are only 16 bits long. If you specify the entire data address it alone will require 16 bits, forcing the instruction to take 2 cycles. In order to meet the single cycle access need, designers broke up data memory into 512   128 word memory "pages".

Immediate Addressing



An instruction using Immediate Addressing transfers the information from program to data space.

Direct Addressing … A How-To



Confused? It seems everyone gets confused the first time (or 3rd or 4th) that they encounter direct addressing. We'll be doing more exercises using the mode so don't worry if it is not 100% clear right now.

Absolute Addressing

Absolute Addressing

```
;main.asm
        .sect "init"
tbl     .int 1,2,3
x       .usect "vars",3
y       .usect "result",1
        .sect "code"
start:  STM  #tbl,AR1
        STM  #x,AR2
        LD   *AR1+,A
        STL  A,*AR2+   ;...
        LD   #x,DP
        LD   @x+1,A
        ADD  @x,A
        ADD  @x+2,A
        STL  A,*(y)
```

- **Guarantees access to *any* location in the memory map by supplying the entire 16-bit address**
- **Uses the indirect hardware to generate the address, hence the ✻**
- **Always MINIMUM of 2 words, 2 cycles**

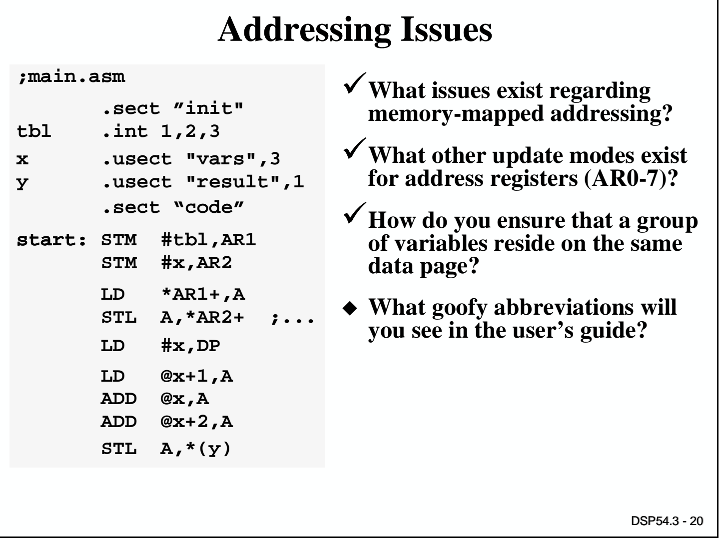
What other issues exist concerning the basic addressing modes?

DSP54.3 - 12

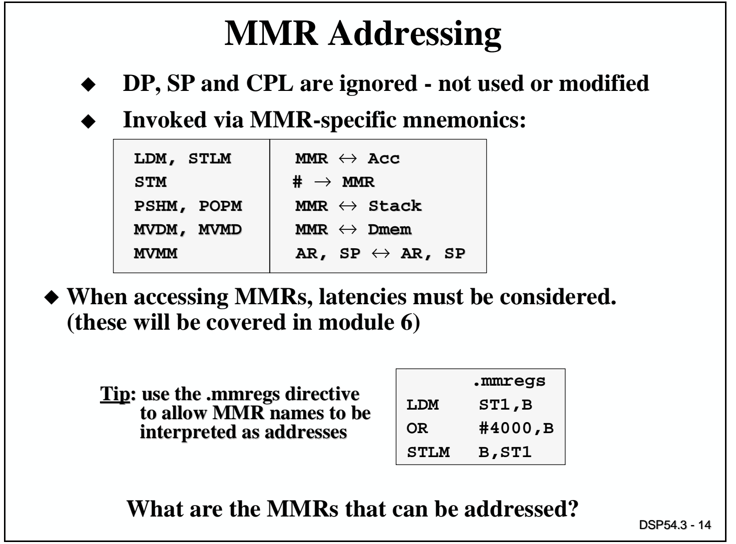
Gee, this looks like what direct addressing would look like if we didn't need a DP. But notice the performance penalty.

The form of the instruction seems to denote indirect addressing and indeed, absolute addressing using the indirect hardware to generate its addresses.

What have we missed?



MMR Issues



MMR addressing allows access to the MMRS regardless of the page or mode, which is very handy. You'll have to remember that these are the only MMR specific instructions. If you use an MMR name as an address in any other instruction (say a direct address) you may need to set the DP to 0.

Memory-Mapped Registers (MMR)

Name	Addr. (Hex)	Description
IMR	0000	Interrupt Mask Register
IFR	0001	Interrupt Flag Register
-----	2 - 5	Reserved
ST0	0006	Status 0 Register
ST1	0007	Status 1 Register
AL	0008	A accumulator low (A[15:00])
AH	0009	A accumulator high (A[31:16])
AG	000A	A accumulator guard (A[39:32])
BL	000B	B accumulator low (B[15:00])
BH	000C	B accumulator high (B[31:16])
BG	000D	B accumulator guard (B[39:32])
T	000E	Temporary Register
TRN	000F	Transition Register

Name	Addr. (Hex)	Description
AR0	0010	Address Register 0
AR1	0011	Address Register 1
AR2	0012	Address Register 2
AR3	0013	Address Register 3
AR4	0014	Address Register 4
AR5	0015	Address Register 5
AR6	0016	Address Register 6
AR7	0017	Address Register 7
SP	0018	Stack Pointer Register
BK	0019	Circular Size Register
BRC	001A	Block Repeat Counter
RSA	001B	Block Repeat Start Address
REA	001C	Block Repeat End Address
PMST	001D	PMST Register
-------	01E-01F	Reserved

XPC and Peripheral MMR locations are device dependent

DSP54.3 - 15

These are core MMRs only. Peripheral MMRs will be covered in the peripheral module.

A List of Indirect Addressing Options

Indirect Addressing Options

Option	Syntax	Action	Affected by:
No Modification	*ARn	no modification to ARn	
Increment / Decrement	*ARn+ *ARn-	post increment by 1 post decrement by 1	
Indexed	*ARn+0 *ARn-0	post increment by AR0 post decrement by AR0	AR0
Circular	*ARn+% *ARn-% *ARn+0% *ARn-0%	post increment by 1 - circular post decrement by 1 - circular post increment by AR0 - circular post decrement by AR0 - circular	BK BK, AR0
Bit-Reversed	*ARn+0B *ARn-0B	post inc. ARn by AR0 with reverse carry post dec. ARn by AR0 with reverse carry	AR0 (=FFT size/2)
Pre-modify	*ARn (lk) *+ARn (lk) *+ARn (lk)% *+ARn	*(ARn+LK), ARn unchanged *(ARn+LK), ARn changed *(ARn+LK), ARn changed - circular pre-increment by 1, during write only	BK
Absolute	*(lk)	16-bit lk is used as an absolute address See Absolute Addressing	

ARs are read in access phase and modified in read phase of the pipeline, so the debugger will appear to show ARs changing early.

DSP54.3 - 17

Other than pre-modify and absolute, these modifications incur no time penalty. The pointer updates occur as you direct within the address generation hardware.

Direct Addressing Issues

Forcing Variables Onto A Single Data Page

vars1: **Guaranteed to reside on DP=2.**
- get a warning if vars1 section is larger than 80h

vars2: **Guaranteed to reside on the *same* data page.**
- blocking forces section sizes of 2, 4, 8, …128

```
MEMORY {
PAGE 1:   HISRAM: org=100h  len=080h
          HERRAM: org=210h  len=200h
          ITSRAM: org=500h  len=100h
}
SECTIONS {
vars1 :>  HISRAM PAGE 1
vars2 :>  HERRAM PAGE 1  BLOCK=128
vars3 :>  ITSRAM PAGE 1
}
```

vars3: **Guaranteed to reside on the *same* data page.**
- uses blocking flag to force variables onto same data page
- must use ".def y" to see variable in the debugger

```
x  .usect "vars3",4,1
y  .set   x+3
```

All 3 methods require close management of the linker.cmd file. DSP54.3 - 19

Using direct addressing effectively involves good management of variable placement. The hardware has no method to determine that your data access was from the correct or incorrect page.

Some Definitions

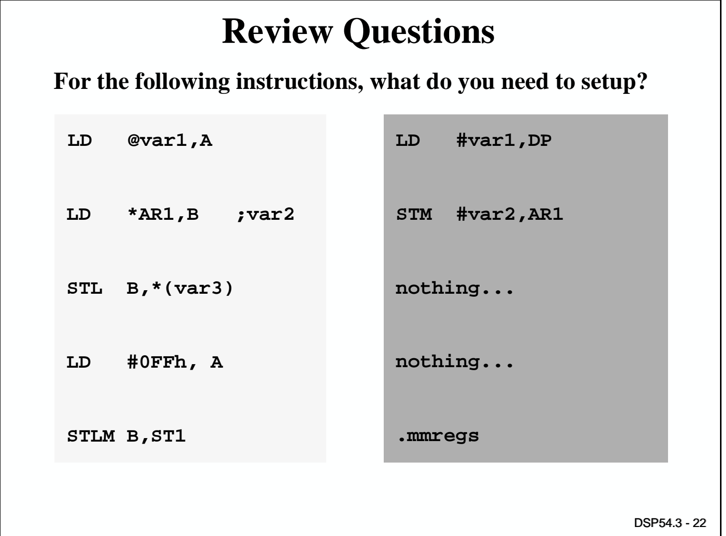
Terms From the User's Guide...

Term	What it means
Smem	16-bit single data memory operand
Xmem	16-bit dual data memory operand used in dual-operand instructions and some single-operand instructions. Read through D bus.
Ymem	16-bit dual data-memory operand used in dual-operand instructions. Read through C bus.
lk	16-bit long constant
dmad	16-bit immediate data memory address (0 - 65,535)
pmad	16-bit immediate program memory address (0 - 65,535) This includes extended program memory devices
src	Source accumulator (A or B)
dst	Destination accumulator (A or B)
PA	16-bit port (I/O) immediate address (0 - 65,535)

DSP54.3 - 21

There are always hard to understand acronyms and these are a few of the most used.

Review



Exercise

Exercise 3: Addressing

Given:	DP=0		DP=4		DP=6	
Address/Data (hex)	60	20	200	100	300	100
CPL=0	61	120	201	60	301	30
CMPT=0	62		202	40	302	60

Program	A	B	DP	AR0	AR1	AR2
`LD #0,DP` `STM #2,AR0`						

`STM #200h,AR1`
`STM #300h,AR2`
`LD @61h,A`
	120					

`ADD *AR1+,A`
`SUB @60h,A,B`
`ADD *AR1+,B,A`
	260					

`LD #6,DP`
`ADD @1,A`
`ADD *AR2+,A`
	390					

`SUB *AR2+,A`
`SUB #32,A`
`ADD *AR1-0,A,B`
		380				

`SUB *AR2-0,B,A` `STL A,62h`						

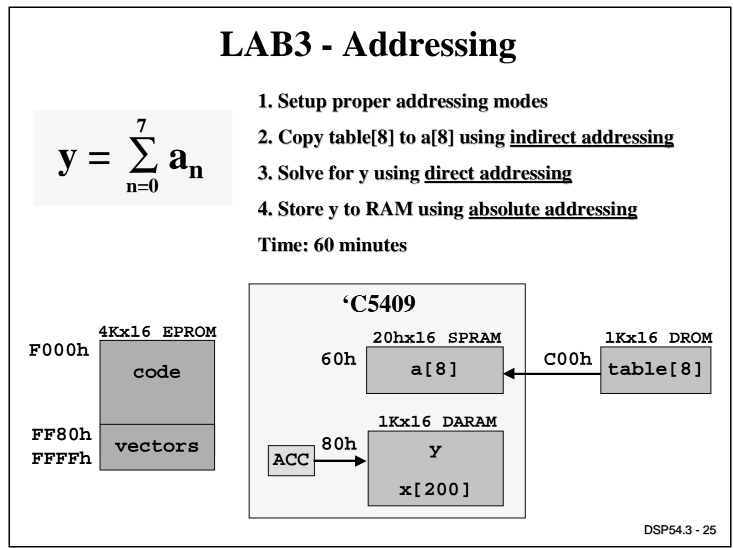
DSP54.3 - 23

In everyday coding you would probably not be using “hard” addresses to load registers but would rather be using symbols. In this exercise we are using hard addresses so we can emphasize addressing modes rather than the use of symbols.

LAB3 – Addressing

Objective

The objective of this lab is to write code to perform a copy of the initialized table from DROM to RAM, add the values stored in the RAM table, and then store the result to y.



LAB3-A Procedure

Copy Files, Create Make File

1. Using CCS, open `LAB2A.CMD` in `C:\DSP54\LABS` and save it as `C:\DSP54\LABS\LAB3A.CMD`. Modify as necessary (especially the file i/o). Save your work.

2. Open `LAB2A.ASM` and save it as `LAB3A.ASM`.

3. Create a new project called `LAB3A.MAK` and add `LAB3.ASM`, `VECTORS.ASM` and `LAB3A.CMD` to it. Check your file-list to make sure all the files are there.

Copy table[8] to a[8] – Write/Debug

4. Edit `LAB3A.ASM` and write code to copy *table[8]* to *a[8]* using <u>indirect</u> addressing. Begin your copy code by writing the actual copy routine, then setup the necessary pointers. This is the "Oreo cookie" approach of coding. First, write the kernel (whatever task you're performing). Then, work on the setup code (like initializing pointers or registers). Then assemble, link and simulate the small kernel to ensure it is working properly before adding other tasks to your code. The worst coding technique is to "write it all", then simulate it all. Gee, if you have an error, you have a zillion places to look. With the "Oreo cookie" method, you'll know exactly where to look.

5. To perform the copy, use a load/store method via the accumulator. Which part of an accumulator (low or high) should be used? Use the following when writing your copy routine:
   - use AR1 to hold the address of *a*
   - use AR2 to hold the address of *table*
   - setup the appropriate indirect addressing registers

6. Save your work. Build and simulate `LAB3A`. You might want to use your workspace from `LAB2A`. You may need to reload your project, remove the `LAB2A.ASM` source window and add the lab3a.asm source window. You may want to re-save the workspace with a "generic" layout without the source file open. Look under File → Recent Workspaces on the menu bar to select your saved workspace quickly.

7. Single-step your copy routine. While single-stepping, it is helpful to see the values located in *table[8]* and *a[8]* at the same time. You can have as many as four memory windows open at the same time. Open two memory windows by using the `"View Memory"` button on the vertical toolbar and using the address labels *table* and `a`. Setting the properties filed to "Hex – TI sytle" will give you more viewable data in the window.

Note:ARs are read/modified early in the pipeline. Therefore, the addresses contained in the register(s) will appear to change early, most often appearing to be two cycles ahead.

First, does *table* contain the proper values? Are the addresses for *table* and *a* what you expected? Single-step your copy routine. Do the values show up in *a*? If not, debug your assembly routine and re-simulate. Get the copy routine working before moving on to the next step.

Add the values, Store result to `y` – Write/Debug

8. Edit `LAB3A.ASM` again and write code to solve for *y* using direct addressing. According to the equation shown on the system diagram, y = a0 + a1 + a2 + a3 + a4 + a5 + a6 + a7. First, write the add routine using one of the accumulators and direct addressing. Then, setup the appropriate direct addressing registers. Create a label (like `add:`) at the beginning of your add routine.

9. Did you check to see if all the values in *a* are contained within a single data page? We can assure they are by adding the "`,1`" switch to the `.usect` command that allocates this section.

10. Store the result *y* using absolute addressing.

11. Create a stop condition at the end of your code with an endless loop. You can use:

```
here:  B here
```

12. Verify that you still have the following selected:

    `Option` → `Program Load` → `Load Program after Build`

13. On the menu bar select: `Project` → `Options` and make sure `–g` is included in the Assembler options box. If it is not, click the box labeled `Enable Symbolic Debug Information`. The `–g` option enables source-level debugging. The options box should now read `–gs.` The `–s` option makes all symbols global so that they can be displayed in Code Composer Studio. Rebuild your `.OUT` file.

14. When the simulator opens, type:

```
go add ↵
```

    on the command line (or use the label you wrote at the beginning of your add routine) to run the code beginning at the *start* label and ending at the *add* label. Use a memory window to view *a* and verify that the values have been properly copied.

15. We know that the copy routine works properly, so now it is time to debug the add routine. Single step your add routine.

Note: When single-stepping, normally the highlighted instruction is sitting at the beginning of the EXECUTE phase of the pipeline. So, when you hit <F8> or press the single-step button, the instruction will complete the EXECUTE phase. This implies that the instructions following the current instruction might have already performed the pre-fetch, fetch, decode, access and read phases. Any updates or modifications made during early phases of the pipeline might already be complete by the time you highlight the specific instruction. For example, `STM` writes in the READ phase. Therefore, when you actually highlight this instruction, it has already "executed".

---

16. The result stored to location *y* should be 16380 (decimal) or 3FFCh. Add the following two expressions to the watch window to display *y* in both decimal and hex:

    `*y` (decimal is the default) … *y will display the *contents* of the address y

    `*y,x` (x selects hexadecimal)

17. View the `CPU Registers` window. Notice that CCS parses `ST0, ST1` and `PMST` into their individual pieces. Review the CCS online help and determine what those pieces are. Find them in the `CPU Registers` window.

18. If your add/store routine is not working properly, debug as necessary before benchmarking your code.

Profile Your Code

19. To re-initialize your debugging session, you have two options. `restart` and `reset.` `restart` loads the PC with your entry label (for example: *start*) and does not reset any registers. `reset` loads the PC with the reset vector address and initializes all reset bits/registers. Use both commands now to see the difference. Type: `reset` or look under `Debug` on the menu bar and select `Reset DSP.`

20. Type: `go start ↵`

    on the command line to run to your beginning label.

21. Click on the line in your source file corresponding to the beginning of your add routine. Click on the "`Toggle Profile-point`" button on the vertical toolbar. You should see a green line appear. Do the same on the instruction performing the store to *y*. Check on the menu bar under `Profiler` and make sure the clock is enabled. Display the Profile window by selecting:

    `Profiler → View Statistics.`

22. Press the Run button, then press the Halt button on the vertical toolbar. You should see the yellow line indicating the current position of the PC on your stop condition.

    Ouch! 64 cycles for 8 instructions? What's happening here is that the software wait state register is set to the maximum (8 cycles per access) at reset so that we can interface with slow memory. We'll see how to set this register in our code later. Type: `?SWWSR=0` on the command line, then `restart` (don't reset) the processor. Run and Halt the simulator and check your results. If you use `reset`, it will set the `SWWSR` to the maximum. To make life easier for future profiling, we created an alias for `?SWWSR=0` called "0ws" (the number ZERO, then ws – for ZERO WAIT STATE). From now on, you can simply type "0ws" at the command line to set wait states to zero.

    Note in the statistics window that you've run the code twice. You should see different values for minimum and maximum. You can clear the statistics by right clicking in the statistics window and selecting "`Clear All`".

23. Take a moment and look at the Appendix of this student guide (at the very back) and study its contents. Is this good stuff or what?

    If you still have some time left and desire a real challenge, move on to LAB3B…

LAB3-B Procedure

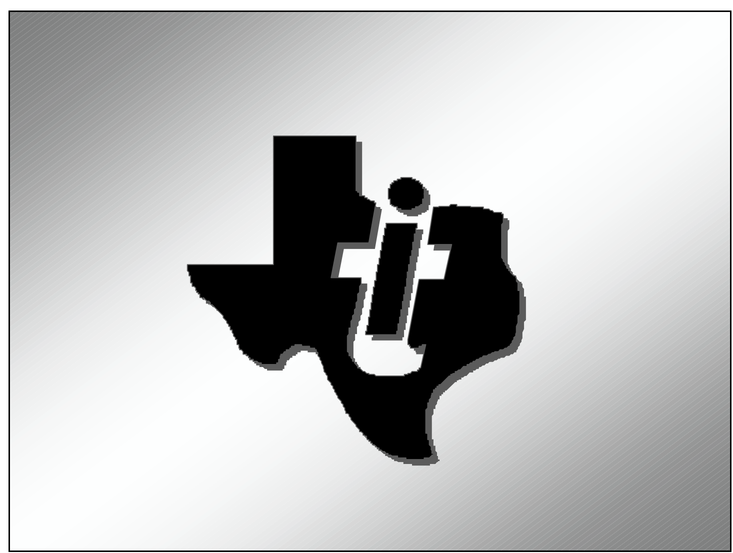
If you plan on modifying your `LAB3A.ASM` code, you might copy the files (`LAB3A.ASM` and `LAB3A.CMD`) to `LAB3B.ASM` and `LAB3B.CMD`. You can obtain the answers to any of these questions by looking in the on-line help guide or asking your instructor.

1. How would you implement a data addressing scheme that crosses data pages using indirect addressing?
2. What addressing modes would you use to implement indexed addressing and what registers would you need to initialize?
3. Determine how to randomly access a memory block greater than 128 words in length.
4. Use READA instruction to perform the copy from *table* to *a*.
5. Implement two ways for forcing variables onto the same data page. Verify your methods via simulation.
6. Review the list of MMR’s in the appendix. You can also locate them in the on-line documentation. If you find them, set a bookmark there for future use.

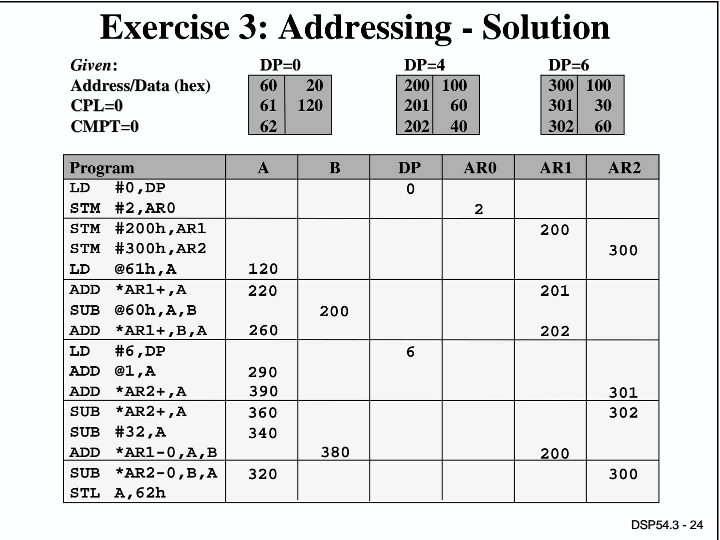
LAB Debrief

1. Which addressing mode was easiest to use? Why?
2. Which mode was the most difficult? Why?
3. Did you watch the copy occur from *table* to *a*?
4. Did the tools behave as expected?
5. What did you learn?
6. Did the lab procedure provide CLEAR directions?
7. Did anyone get to part B of the lab?

DSP54.3 - 26



Solutions



Exercise 3: Addressing - Solution

***Given*:**
Address/Data (hex)
CPL=0
CMPT=0

DP=0	
60	20
61	120
62	

DP=4	
200	100
201	60
202	40

DP=6	
300	100
301	30
302	60

Program	A	B	DP	AR0	AR1	AR2
LD #0,DP			0			
STM #2,AR0				2		
STM #200h,AR1					200	
STM #300h,AR2						300
LD @61h,A	120					
ADD *AR1+,A	220				201	
SUB @60h,A,B		200				
ADD *AR1+,B,A	260				202	
LD #6,DP			6			
ADD @1,A	290					
ADD *AR2+,A	390					301
SUB *AR2+,A	360					302
SUB #32,A	340					
ADD *AR1-0,A,B		380			200	
SUB *AR2-0,B,A	320					300
STL A,62h						

DSP54.3 - 24

LAB3A.ASM : Solution

```
        .def start

a       .usect "coeffs",8,1
y       .usect "result",1

        .sect "init"
table   .int 7FCh,7FDh,7FEh,7FFh
        .int 800h,801h,802h,803h
;       .int 803h,802h,801h,800h
;       .int 7FFH,7FEH,7FDH,7FCH

        .sect "indata"
x       .copy "in6.dat"

        .sect "code"
start:  STM     #a,AR1
        STM     #table,AR2
        LD      *AR2+,A         ;1
        STL     A,*AR1+
        LD      *AR2+,A         ;2
        STL     A,*AR1+
        LD      *AR2+,A         ;3
        STL     A,*AR1+
        LD      *AR2+,A         ;4
        STL     A,*AR1+
```

```
        LD      *AR2+,A         ;5
        STL     A,*AR1+
        LD      *AR2+,A         ;6
        STL     A,*AR1+
        LD      *AR2+,A         ;7
        STL     A,*AR1+
        LD      *AR2+,A         ;8
        STL     A,*AR1+

        LD      #a,DP

add:    LD      @a,A
        ADD     @a+1,A
        ADD     @a+2,A
        ADD     @a+3,A
        ADD     @a+4,A
        ADD     @a+5,A
        ADD     @a+6,A
        ADD     @a+7,A
        STL     A,*(y)

here:   B       here
```

DSP54.3 - 28

LAB3A.CMD : Solution

```
/* file I/O and options       */
vectors.obj
lab3a.obj
-m lab3a.map
-o lab3a.out
-e start

MEMORY {
PAGE 1:        /* Data memory */
  SPRAM:    org = 00060h,  len = 00020h
  DARAM:    org = 00080h,  len = 00400h
  DROM:     org = 00C00h,  len = 00400h

PAGE 0:        /* Program memory */
  EPROM:    org = 0F000h,  len = 00F80h
  VECS:     org = 0FF80h,  len = 00080h
}

SECTIONS
{ coeffs         :> SPRAM   PAGE 1
  result         :> DARAM   PAGE 1
  init           :> DROM    PAGE 1
  indata         :> DARAM   PAGE 1
  code           :> EPROM   PAGE 0
  vectors        :> VECS    PAGE 0
}
```

DSP54.3 - 29

Programming FIR Filters

Introduction

While the purist might quibble that what we're covering here isn't everything needed to perform FIR filter, we will cover the most important aspect; multiply-accumulates. We'll introduce the concept of a sampled signal and work a complete problem using just 4 data points. In later modules we'll look at how important managing pointers and data is.

Learning Objectives

Objectives

- **Introduce DSP Filtering**
- **Compare/contrast array vs. scalar math**
- **Solve an FIR filter using basic math and program flow instructions**
- **List additional instructions**

DSP54.4 - 2

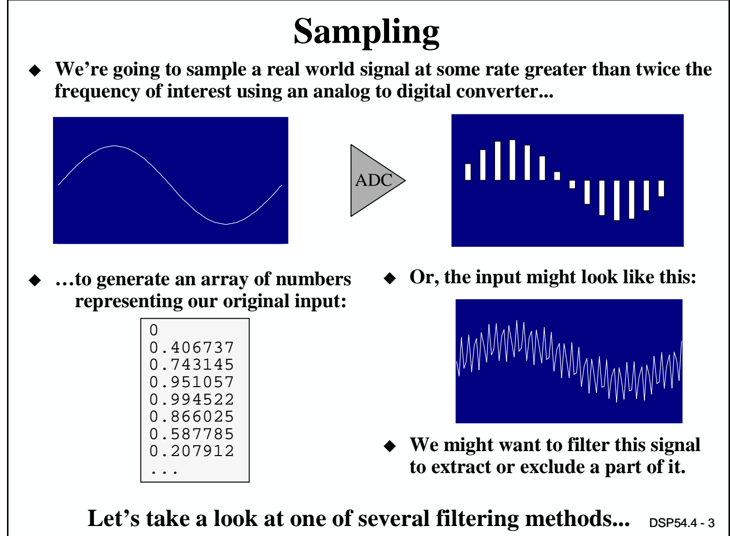
Module Topics

Programming FIR Filters **4-1**

Module Topics *4-2*
Converting the Analog World to Digital *4-3*
FIR Filters *4-4*
Array Math *4-5*
Multiply and Accumulate *4-6*
Store to Memory Mapped Registers *4-7*
Loads *4-8*
Store Accumulator to Memory *4-9*
Repeat Single *4-10*
Move Instructions *4-11*
Program Flow *4-12*
The Stack *4-13*
Review *4-14*

LAB4 – 16-TAP FIR **4-15**
Objective 4-15
LAB4-A Procedure *4-16*
Copy Files, Create Make File 4-16
Edit LAB4A.CMD 4-16
Setup 16-TAP FIR and Stack – Write/Debug 4-16
Optimize Copy Routine – Write/Debug 4-17
FIR Routine – Write/Debug 4-17
Optimize Your FIR Routine – Write/Debug 4-18
LAB4-B Procedure *4-19*
What Have We Missed? *4-20*
IIR Filters 4-20
More Multiply Instructions 4-21
Adds and Subtracts 4-22
32 Bit Operations 4-22
Aligning Long Operands 4-23
Far Operations 4-23
Solutions *4-25*
Some Additional Information … *4-27*

Converting the Analog World to Digital

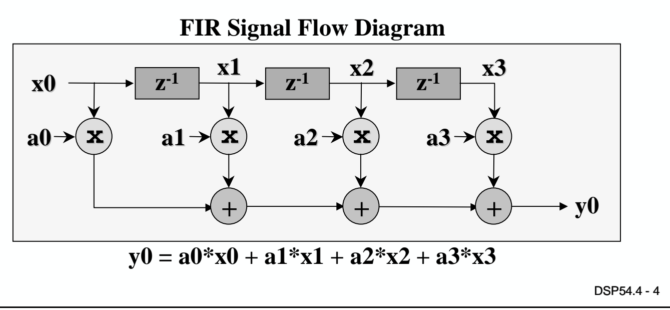


There certainly isn't time or space here to cover time-invariant sampling theory, but the idea of sampling a real world signal at a given rate is at the heart of DSP. Remember Nyquist? We must sample our signal at a minimum of twice the frequency of interest. Since voice ranges up to 4KHz, we must sample voice signal at a minimum of 8KHz or 125uS.

FIR Filters

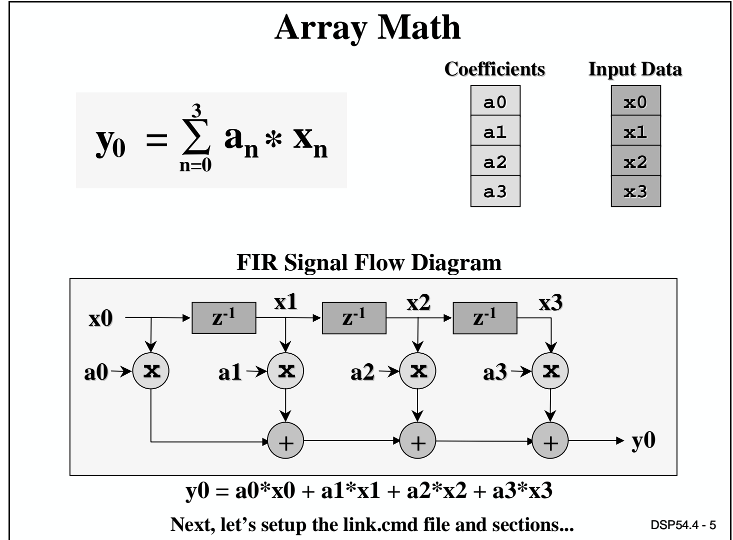
DSP Filtering Using an FIR Filter

- **Unconditionally stable (no feedback)**
- **Linear phase possible**
- **Usually requires many taps - so typically array math and indirect addressing are used**
- **MAC is the basic instruction used to solve an FIR filter**

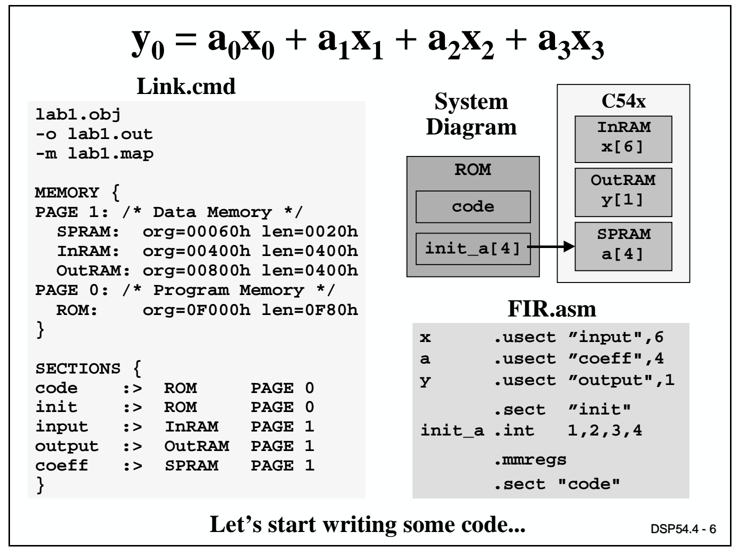


An FIR filter in its simplest implementation is an averaging filter. To determine the average high temperature for a week we'd take the high for each day, add together all 7 and divide by 7. Since multiplication is easier than division we can multiply each "data point" by our "coefficient" (1/7) and accumulate the result. Unfortunately, FIR filters are relatively poor performers, so many "taps" or multiply-accumulates will be needed.

Array Math

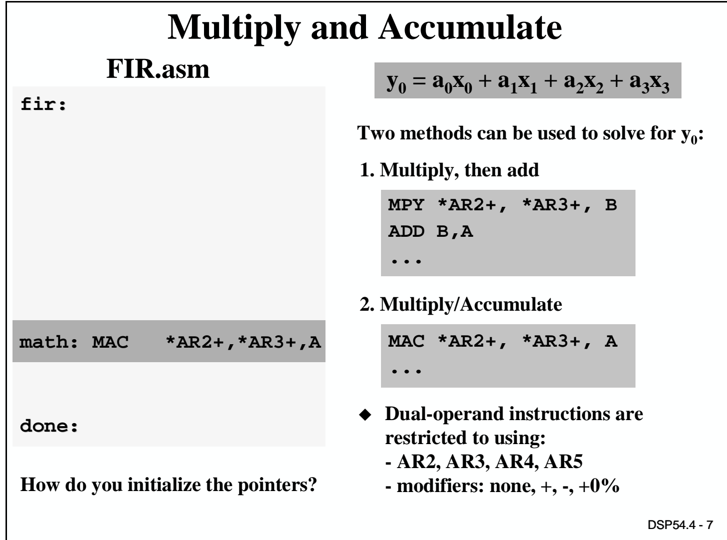


There they are … an array of data and an array of coeffiecients.



Let's make sure they're correctly placed in memory to avoid access conflicts.

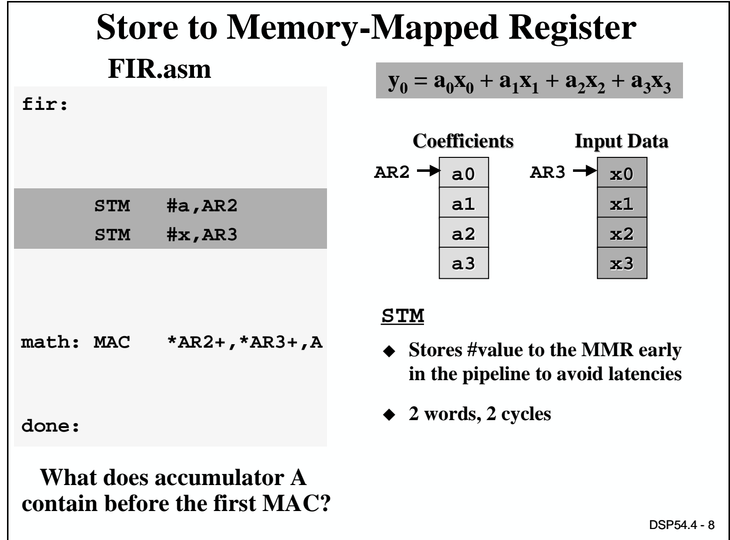
Multiply and Accumulate



This is the "Oreo" method of coding. An Oreo cookie is eaten from the inside out, and this is how the "Oreo" technique of coding works. The equation is a MAC, so start out by writing the MAC instruction. Then you can think about what register will be needed to be initialized to make it work. Finally you can work on storing the results.

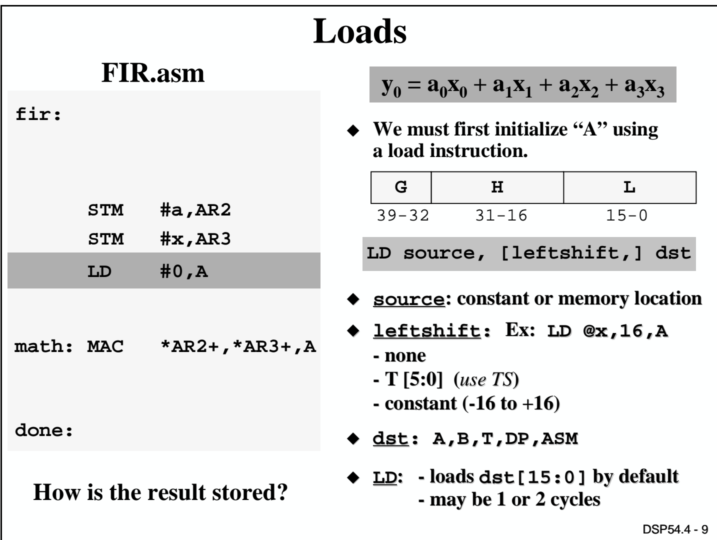
Note that the post-increments will step through our data array.

Store to Memory Mapped Registers



If you intend to use pointers you must initialize them. Duh.

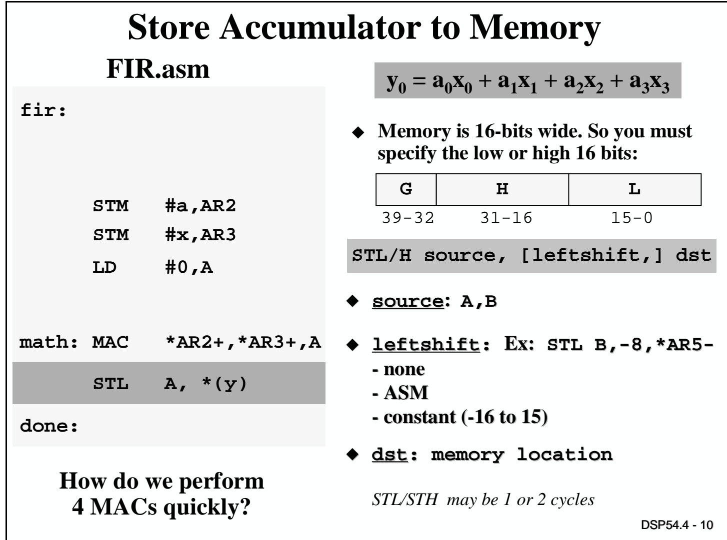
Loads



Since MAC merely accumulates to the A or B accumulator, any previous value would be incorporated as well. So we need to initialize it by loading A with 0. This load would be a 8 bit constant and the instruction would take a single cycle. The assembler will look at the immediate value and decide on the correct interpretation of the LD mnemonic. Since a load clears the upper bits, this will work perfectly.

Can you think how we might more efficiently initialize the accumulator?

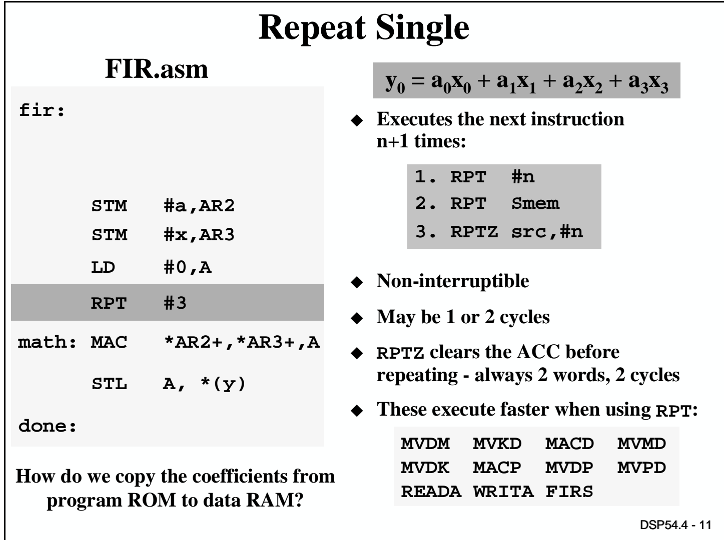
Store Accumulator to Memory



Accumulators are 32 bits plus 8 guard bits. You must indicate which portion of the accumulator you wish to store (with an optional shift value).

The ASM bit field may be used to perform a shift by a preset amount.

Repeat Single

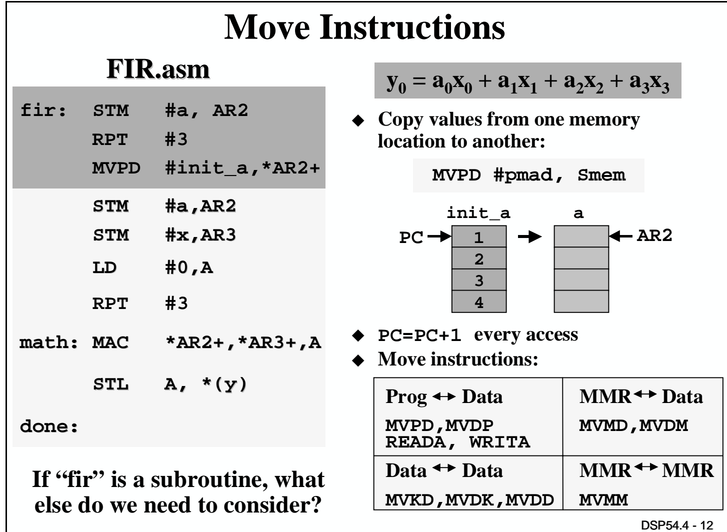


Repeat Single repeats the following instruction by the programmed number + 1. This is because efficient loop control runs the 0th iteration.

Using RPTZ would eliminate the need to use the "`LD #0,A`" instruction by zeroing A the first time through.

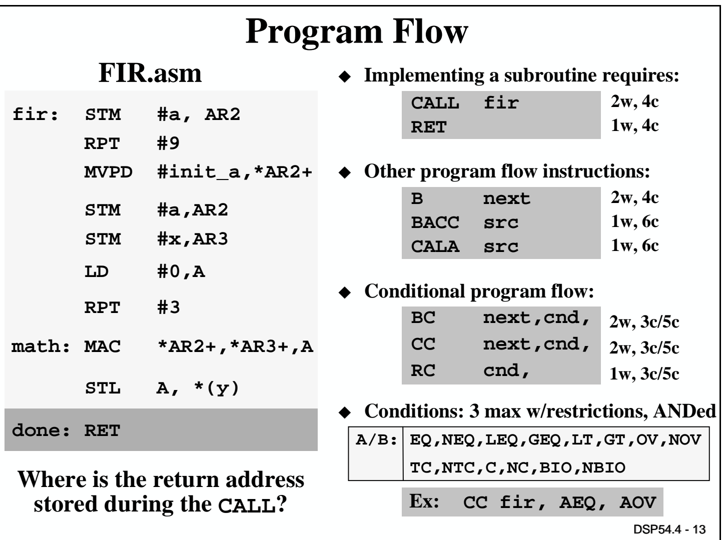
The listed instruction execute faster when placed in a RPT loop since they require the hardware to set up a tear down pointers internally.

Move Instructions



Move instructions allow program and data movement, but require program and processor attention. The DMA transfers information without attention, but only moves between data memories. Some C54x devices place additional restrictions on DMA movements.

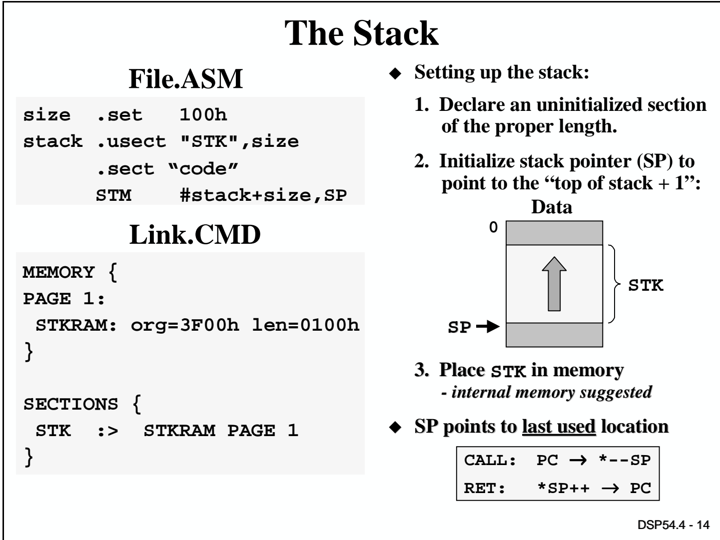
Program Flow



RET is a return from a subroutine. We'll see other variants on returns later.

Restrictions on conditions are that you can choose up to 3 from the top row or up to 2 from the bottom, but you may not mix them. See the CPU and Peripherals User Guide for more detailed information.

The Stack



The stack pointer is a pre-decrementing pointer which points to the last used location. Before the stack pointer is used the first time it should point to one location higher (after) the stack. In assembly, you must take care of stack initialization yourself. If you use C, the boot.asm routine will do this for you.

Review

Review

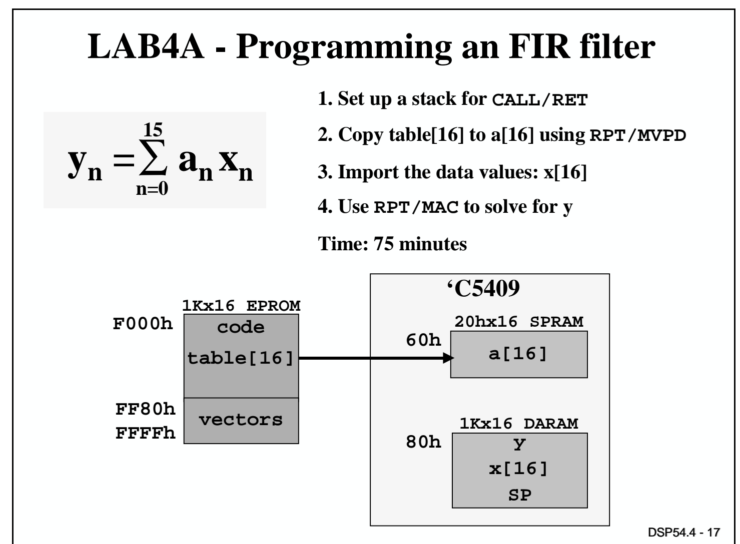
1. Name some characteristics of FIR filters?
2. What restrictions exist for a dual-operand MAC?
3. What does “ `RPT 5` ” do?
4. Write the instruction to store A[23:8] to @y.
5. Which part of memory should the stack (SP) be located in and why?

DSP54.4 - 15

LAB4 – 16-TAP FIR

Objective

The objective of this lab is to write code to perform a 16-tap FIR (actually, it is a sum of products because we don’t get any new data, but who’s asking?) as described in module 4. You have already copied the coefficients (*a* values) into RAM, but we need to modify the code to use 16 values instead of 8. The input file, `in4.dat`, contains the 16 input data values of the filter.



LAB4-A Procedure

Copy Files, Create Make File

1. In CCS, open `LAB3A.CMD` and save it as `LAB4A.CMD`. Modify the file i/o and save your work.

2. Open `LAB3A.ASM` and save it as `LAB4A.ASM`.

3. Create a project called `LAB4A` and add the necessary files to it.

Edit LAB4A.CMD

4. Change the routing of the "init" section (which contained *table[8]* before) from data space (PAGE 1) to program space (PAGE 0). Place the "init" section into `EPROM` (along with your code).

5. Allocate a section called "`STK`" for the stack and route it to `DARAM` along with the input samples (*x*) and results (*y*). Save your work.

Setup 16-TAP FIR and Stack – Write/Debug

6. Edit `LAB4A.ASM`. Change your copy statement to read:

```
x  .copy "in4.dat"
```

7. You can open up and view the `in4.dat` file if you wish, just to see what it contains.

8. Because we are now using a 16-tap FIR which requires 16 input values and 16 coefficients, you need to change the size of the uninitialized data section for your coefficients (related to label *a*) to 16 instead of 8. Also, remove the semi-colons (comments) on the last two sets of 4 coefficients in the initialized section for *table*. Now *table* contains 16 coefficients.

9. Allocate an uninitialized data section 100 words in length for the stack. Name the section "STK". You cannot name this section "stack" because this name is reserved by the linker. This might be a good time to use the .set directive (for example STKLEN .set 100). Make sure your `.usect` uses a label such as `BOS` (for bottom of stack). This label will be used to load the stack pointer.

10. Add the `.mmregs` directive to the top of your code to facilitate using MMR names as addresses. Anywhere before your code section is okay.

11. Load the stack pointer (SP) as described in class. Make sure your *start* label is on this instruction. Otherwise, your reset vector and restart command will not be able to find the beginning of your code.

12. Move the stop condition (`here: B here`) to the next instruction following the SP initialization. You should now have *start* and *here* as sequential labels.

13. Build `LAB4A`. Simulate the code and verify that you now have 16 coefficients in *table*. Open a memory window on address *table*. Do you see your coefficients? Hmmm. Why not? Well, the default memory window shows DATA memory. Where did you map the section

containing *table* in your linker command file? Oh, that's right – it is now in PROGRAM space. To verify if the values in *table* are correct, right-click on the memory window and change the page to `Program`.

14. Also, verify that `in4.dat` was loaded properly. Open a memory window starting at address `x`. It might also help if you select "`16-bit Signed Int`" for the format.

    Do you see some data? The first 4 values should be 1,2,3,4.

15. Single-step until you hit your stop condition (`here: B here`). Now, verify that your stack pointer (SP) is set up correctly by opening a new memory window on SP in TI - Hex Format. What location is SP pointing to? SP is now currently pointing to one location PAST the end of the stack. SP is a pre-decrementing pointer, so the first CALL you make will write the return address to one location before where SP is pointing now. Try changing the memory window address to SP-6. This will now allow you to see return addresses stored on the stack later on.

Optimize Copy Routine – Write/Debug

16. Edit `LAB4A.ASM`. Between the *start* and *here* labels in your code, add an instruction to `CALL` your copy routine.

17. Change your copy routine by doing the following:
    - add a label *copy* pointing to the first instruction
    - use single repeat and `MVPD` to copy 16 coefficients from *table* to *a*
    - make sure you copy all 16 values and return back to the main routine.

18. Build `LAB4A`. Verify that your copy routine works and returns back to the main routine. After the simulator starts:
    - Hit `<F8>` to execute the `STM` instruction and load the stack pointer (SP).
    - Single step again – once – and watch the return address go onto the stack.
    - Open a view memory window to display *a*.
    - Single-step the copy routine and watch the coefficients copy into *a*. Notice that you can't single step inside the RPT loop. Did it work?
    - When you single-step the RET instruction, watch the return address load itself into the PC in the Register window. When you reach your main routine and everything is working properly, move on to the next step.

FIR Routine – Write/Debug

19. Edit `LAB4A.ASM`. Add a `CALL` to the code label *fir* after the `CALL` to *copy*.

20. Write your *fir* routine by doing the following:
    - Write a dual-operand `MAC` instruction using the proper pointers
    - Use a repeat single to accumulate 16 products. This might be a good opportunity to use `RPTZ`.

- Initialize the appropriate registers
- Store the result to *y* using absolute addressing (this should already be in your code). Make sure you store the correct part of the accumulator.
- Return back to the main routine

21. Build `LAB4A`. Simulate and verify that your FIR (currently just a sum of products) works correctly:
    - Because everything should work up to the *fir* routine, type: `go fir` on the command line.
    - Add a watch on *y* in hexadecimal.
    - Single-step the `MAC` loop. The result in the accumulator and stored to *y* should be `14h`.
22. Profile your *fir* routine by setting profile points at the label *fir* and on the store to *y*. Type `restart`, then Run, Halt and check your statistics. You did remember to enable the clock and view the statistics window, right? You should get about 22 cycles, unless you forgot to set the wait states to zero (oops…).

Optimize Your FIR Routine – Write/Debug

23. Is there a better way to initialize the accumulator instead of loading it with zero? Of course. Why else would we be asking? Instead of loading it with zero or using `RPTZ`, initialize your accumulator with the first product of the MAC and reduce the repeat count by one. Write it and profile it. Did you get one less cycle?
24. If you're done with `LAB4A` and you still have some time, move on to `LAB4B`…

LAB4-B Procedure

1. What instructions should NOT be placed in a single repeat? Why?
2. Where should the stack be located and why?
3. What conditions can you combine in a conditional instruction?
4. Rewrite lab4a fir routine to use MACP

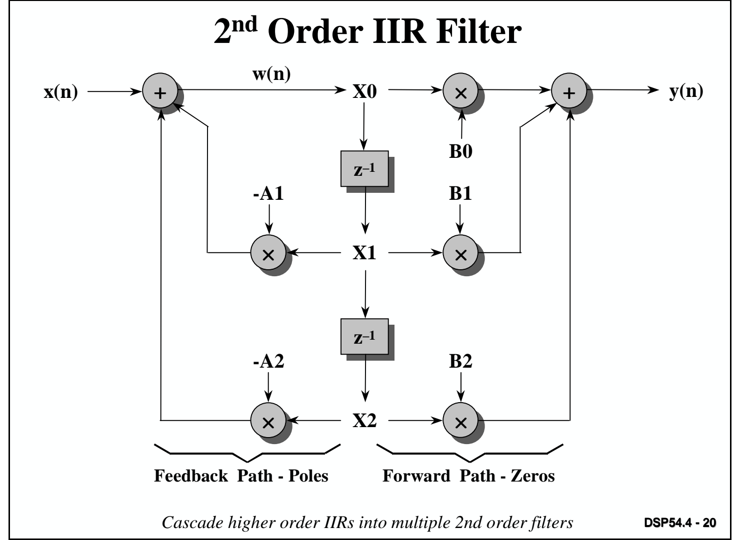
What Have We Missed?

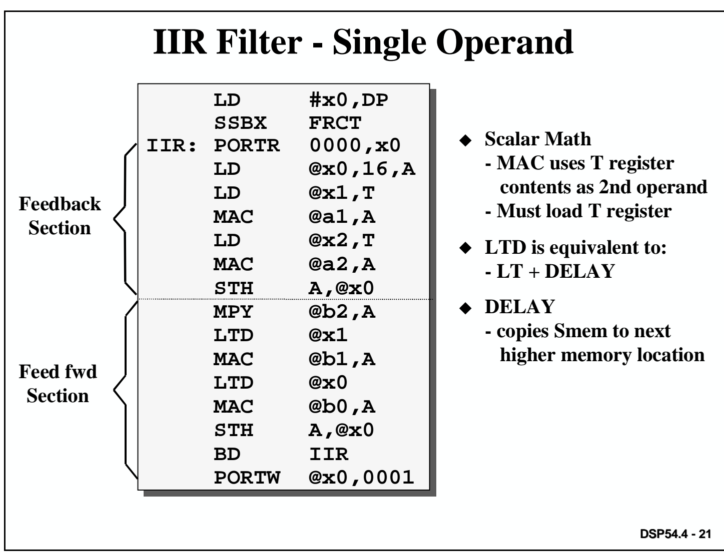
IIR Filters

IIR Characteristics

- **Potentially unstable due to feedback path**
- **More computationally efficient than FIR**
- **Programmer must use care to ensure proper operation**
- **Much better frequency performance**
- **Best for frequency discrimination**

DSP54.4 - 19





More Multiply Instructions

Multiply Instructions

`MPY`	Single or dual-operand multiply
`MPYA`	Uses AH as multiplicand
`MAC`	Single or dual-operand multiply/accumulate
`MACA`	Uses AH as multiplicand
`MAS`	Single or dual-operand multiply/subtract
`MASA`	Uses AH as multiplicand
`MACP`	Uses pmad as one multiplicand
`MACD`	Copy data to next higher address
`SQUR`	Square single operand or AH
`SQURA/S`	Square single operand and accumulate/subtract

Each instruction supports multiple options for operands.
Refer to the User's Guide for more information.

DSP54.4 - 23

Adds and Subtracts

Add and Subtract Instructions

Instruction	Description
`ADD`	Single or dual-operand add
`ADDC`	Single operand add with carry
`ADDS`	Add with sign-suppression
`ADDM`	Add a constant to a memory location
`SUB`	Single or dual-operand subtract
`SUBB`	Single operand subtract with borrow
`SUBS`	Subtract with sign-suppression
`SUBC`	Conditional subtract (performs 1-bit divide)

Each instruction supports multiple options for operands.
Refer to the User's Guide for more information.

DSP54.4 - 25

32 Bit Operations

32-bit Operations

Instruction	Description
`DLD`	Loads 32-bit value from memory to ACC
`DST`	Stores 32-bit value from ACC to memory

Instruction	Description
`DADD`	Adds 32-bit value from memory to ACC
`DSUB`	Subtracts 32-bit value from ACC to memory
`DRSUB`	Reverses operands used in DSUB

- Affected by C16 bit in ST1 (splits ACC's into two independent 16-bit registers)

- **Double instructions use long-memory (lmem) operands**
- **Double store (`DST`) requires two cycles for dual E-bus activity**
- **Internal memory hardware is organized as 32-bit. Therefore, double adds/subtracts/loads from any internal memory are single cycle.**
- **Default auto-increment step size is TWO**

How are long operands aligned in memory?

DSP54.4 - 27

Aligning Long Operands

Address	Data
0FFh	1234
100h	5566
101h	7788

	Instruction	A
	`STM #100h,AR1`	
Words 16-bit	`LD *AR1, A`	0000 5566
Longs 32-bit	`DLD *AR1+, A`	5566 7788

- **Long accesses assume address points to MSW**
- **LSW read from <u>same</u> address with LSB *toggled*.**

  1. Ptr=100h (MSW @100h, LSW @101h)
  2. Ptr=101h (MSW @101h, LSW @100h)

<u>You must EVEN-align long operands:</u>

- **Constants: (.int/.word, .long)** — **Auto-aligns on "type" boundary**
- **Variables: (use even-align flag)** — `.usect "Sect",len,1,1`

DSP54.4 - 28

Far Operations

Using FAR Operations

`FB`	Far Branch
`FCALL`	Far Call
`FRET`	Far Return
`FBACC`	Far branch to location specified by ACC[23:0]
`FCALA`	Far call to location specified by ACC[23:0]
`FRETE`	Far return from ISR

23 – 16	15 – 0
XPC	16-bit addr

- **During a `FCALL`, the PC is placed on the stack followed by the XPC**
- **Other instructions do not modify the XPC**
- **The size of the XPC and consequently the address used in computed operations (like `FBACC`) depends on the chosen device.**

DSP54.4 - 30

Solutions

Review

1. **Name some characteristics of FIR filters?**
   unconditionally stable, linear phase possible, typically lots of taps
2. **What restrictions exist for a dual-operand MAC?**
   Use AR2-5 only, modifiers (none, +, -, +0%)
3. **What does " `RPT 5` " do?**
   Repeats the next instruction [1 + (value located at the 5th word from the current DP)]
4. **Write the instruction to store A[23:8] to @y.**
   `STL A,-8,@y` -OR- `STH A,8,@y`
5. **Which part of memory should the stack (SP) be located in and why?**
   Internal RAM, to decrease access time

DSP54.4 - 16

`LAB4A.ASM` : Solution

```
        .mmregs
        .def start

STKLEN  .set    100

a       .usect "coeffs",16,1
y       .usect "result",1
BOS     .usect "STK",STKLEN

        .sect "init"
table   .int 7FCh,7FDh,7FEh,7FFh
        .int 800h,801h,802h,803h
        .int 803h,802h,801h,800h
        .int 7FFH,7FEH,7FDH,7FCH

        .sect "indata"
x       .copy "in4.dat"

        .sect "code"
start:  STM     #BOS+STKLEN,SP   ;setup stack pointer
        CALL    copy
        CALL    fir

here:   B       here             ;return
```

DSP54.4 - 32

LAB4A.ASM - Solution (Continued)

```
copy:   STM     #a,AR1          ;setup AR1
        RPT     #15             ;copy 16 values
        MVPD    #table,*AR1+
        RET                     ;return

fir:    STM     #a,AR2          ;setup ARs for MAC
        STM     #x,AR3
        MPY     *AR2+,*AR3+,A   ;1st product
        RPT     #14             ;mult/acc 15 terms
          MAC   *AR2+,*AR3+,A
        STL     A,*(y)          ;store result

        RET
```

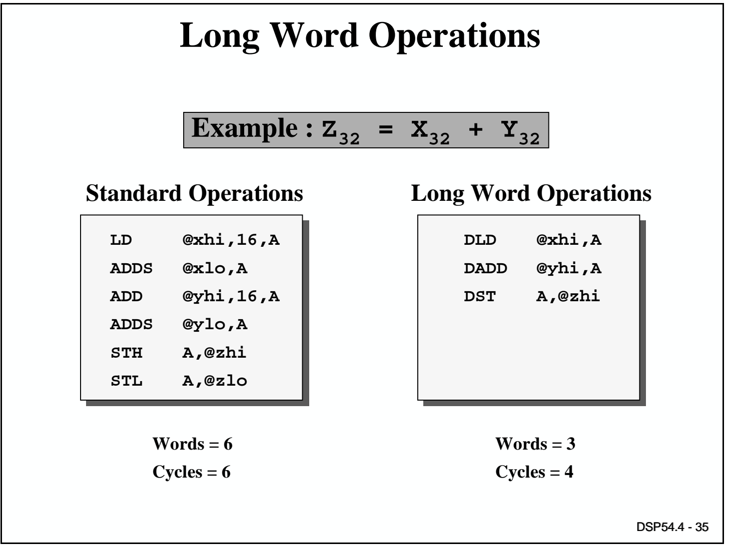
DSP54.4 - 33

LAB4A.CMD : Solution

```
/* file I/O and options       */
vectors.obj
lab4a.obj
-m lab4a.map
-o lab4a.out
-e start

MEMORY {
PAGE 1:        /* Data memory */
  SPRAM:    org = 00060h,  len = 00020h
  DARAM:    org = 00080h,  len = 00400h

PAGE 0:        /* Program memory */
  EPROM:    org = 0F000h,  len = 00F80h
  VECS:     org = 0FF80h,  len = 00080h
}

SECTIONS
{ coeffs         :> SPRAM   PAGE 1
  result         :> DARAM   PAGE 1
  indata         :> DARAM   PAGE 1
  STK            :> DARAM   PAGE 1
  code           :> EPROM   PAGE 0
  init           :> EPROM   PAGE 0
  vectors        :> VECS    PAGE 0
}
```

DSP54.4 - 34

Some Additional Information …



Bus Usage

Instruction Activity	PB	CB	DB	EB
Program Read	A,D			
Program Write	A			D
Data Single Read			A,D	
Data Dual Read		A,D	A,D	
Data Long (32-bit) Read		A,D(ms)	A[1],D(ls)	
Data Single Write				A,D
Data Read / Data Write			A,D	A,D
Dual Read / Coefficient Read	A,D	A,D	A,D	
Peripheral Write				A,D
Peripheral Read*			A,D	

* MMRs only accessible via D Bus, MMR access as Ymem op yields bad data!

DSP54.4 - 36

Numerical Issues

Introduction

Understanding numerical issues is fundamental to getting the best performance from a fixed-point processor. A fixed-point processor generally operates without the benefit of floating point numeric representation, so the programmer must bear in mind overflow during calculations.

Objectives

Learning Objectives

- **Compare/Contrast integers vs. fractions**
- **Use methods for handling multiplicative and accumulative overflow**
- **Discuss other important instructions that handle various numeric types**

DSP54.5 - 2

Module Topics

Numerical Issues **5-1**

Module Topics *5-2*

Integer Multiplication *5-3*

Fractional Multiplication *5-4*

The Fractional Model *5-5*

Handling Accumulative Overflow *5-6*

What's Missing? *5-8*
Bit Compare and Test 5-8
Boolean Operations 5-9
Shift and Rotate Operations 5-9
Some Other Math Operations … 5-10

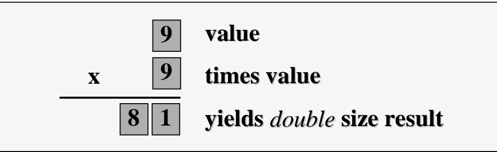
Review *5-11*

Solutions *5-12*

Some Additional Information … *5-13*
Division 5-13
Long Multiplies 5-14
Using Exponents 5-14

Integer Multiplication

Integer Multiplication



- **I * I ≥ I**
- **Which digit should be stored - the upper (8) or the lower (1)?**
  - **- Both must be kept (uses additional resources)**
  - **- Also, the results cannot be used recursively**

Which numerical model solves this problem?

DSP54.5 - 3

Fractional Multiplication

Fractional Multiplication

	. 9	value
x	. 9	times value
	. 8 1	yields *double* size result
	. 8	result to be stored

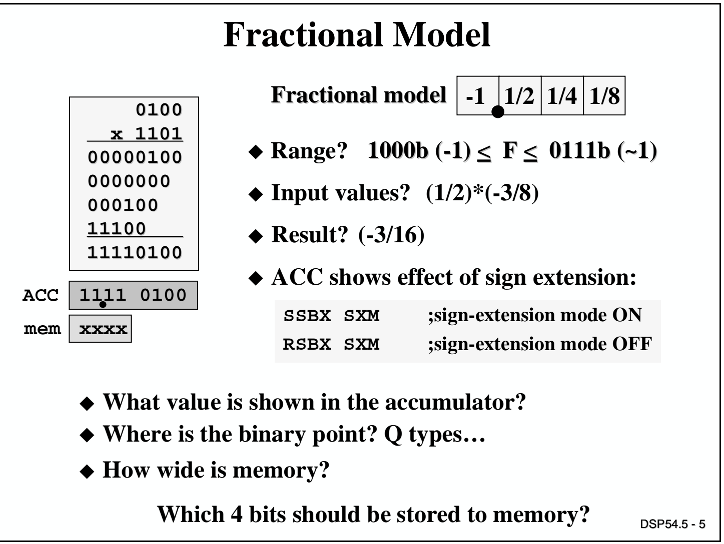
- F * F $\leq$ 1
- Which digit should be stored?
- If only .8 is stored, have we lost important information?
  - No, because the output is as exact as the input
- The accumulator is double-wide to maintain interim results

Let's look at the fractional model in action...

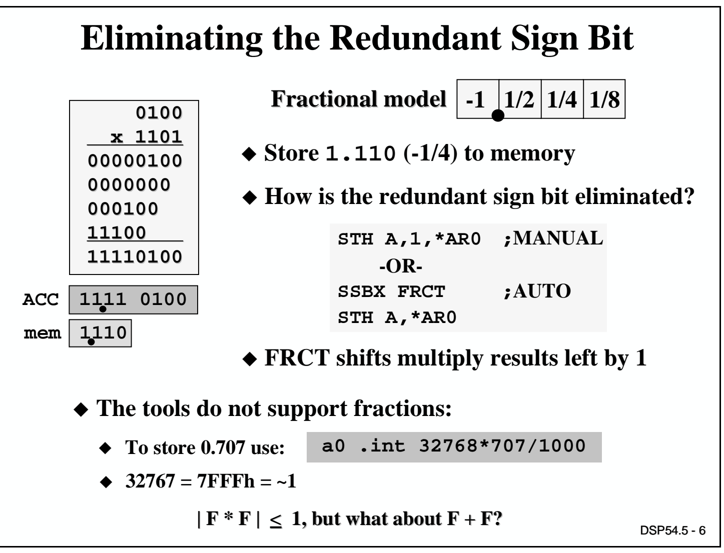
DSP54.5 - 4

We often forget that the accuracy of a calculation can be no greater than the accuracy of its inputs. In the example above, .9 is accurate to tenths. The doublewide result though, contains information precise to100ths, which we know nothing about. The result that we will be storing, .8 represents the most accurate result possible.

The Fractional Model



Q refers to quantization. Even when multiplying decimal numbers, it is a similar process to determine where to put the radix point. Count the total number of places to the right of the radix point in the multiplicands and place the radix point there in the result.



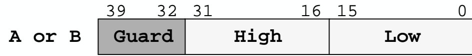
Redundant sign bits are not specific to TI. Anytime you multiply two signed binary number together you will produce two sign bits. The FRCT mode, when selected will shift the multiplier result left by one position. If you happen to need to calculate an integer value like an address while FRCT mode is on, your answer will be twice what you expect.

Handling Accumulative Overflow

Handling Accumulative Overflow

- **F + F could be > 1, so how is this handled?**

1. Use <u>Guard Bits</u> (allow at least 128 signed summations):



Guard bits increase dynamic range from +/-1 to +/-128

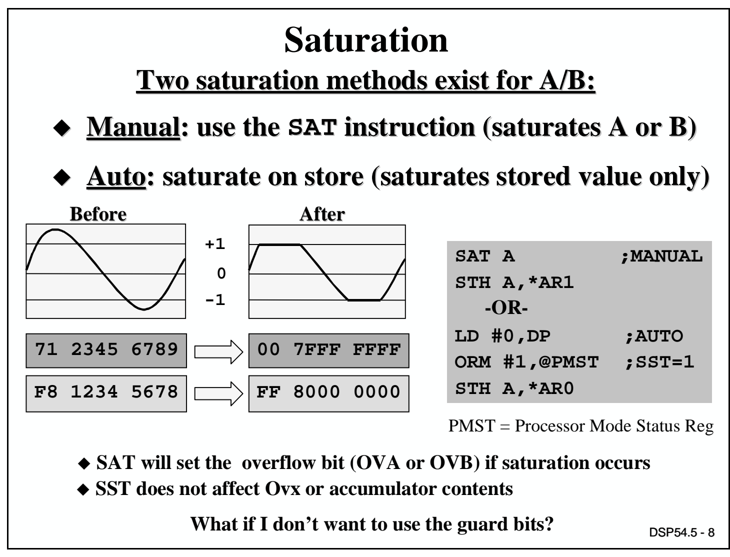
2. In a <u>non-gain system</u> temporary overflow is permitted. The output is guaranteed to remain bounded by the input.

3. In a <u>system with gain</u>, the output is *not* guaranteed to remain bounded (i.e. result is larger than 32-bits).

How do you handle a result larger than 32-bits?

DSP54.5 - 7

How will you write a value larger than 1 or smaller than –1 to a DAC that only understands fractions? The answer is that you can't, anything greater than 1 is 1 and anything smaller than –1 is –1. This process is called saturation.



Overflow Mode

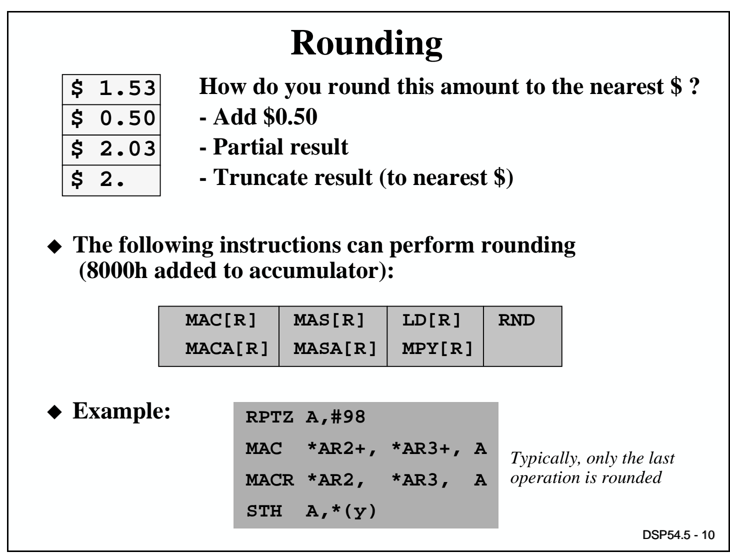
```
SSBX   OVM     ;turn overflow mode ON
```

- **With OVM=1, computations will not exceed 32 bits and are saturated if overflow occurs.**
- **$ST0_{OVA/OVB}$ (overflow bits for A and B) are set and latched if overflow occurs**
- **$ST1_{OVM}$ = 0 at reset**

How do we recover 1/2 bit of accuracy in our result?

DSP54.5 - 9

Some algorithms require precisely 32 bits for their execution. Turning on overflow mode effectively turns off the guard bits.



If you had used MACR in the repeat loop above, you would have added 50 to the final result.

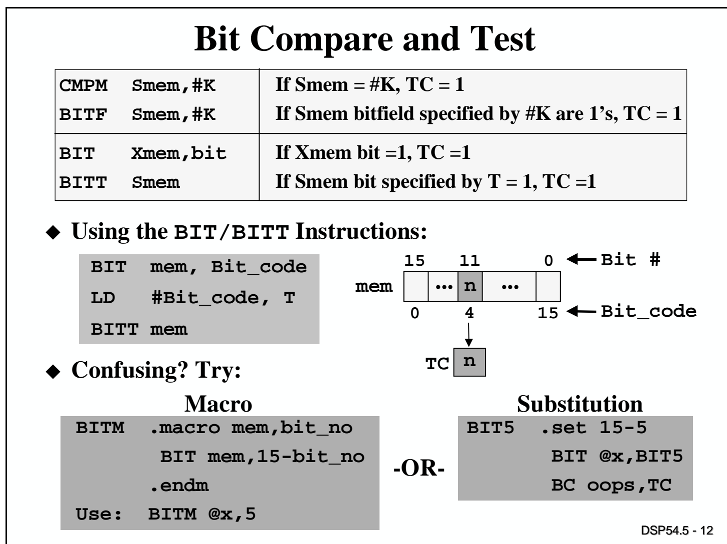
What's Missing?

What's Missing?

- ✓ How do I test and compare bits?
- ✓ What boolean operations can I perform?
- ✓ What shift/rotate operations exist?
- ◆ What other useful math operations could I use?

DSP54.5 - 17

Bit Compare and Test



CMPM compares a location in memory (Smem) to a constant and sets the test condition (TC) bit if they are equal.

BITF tests a 1 to 16 bits of Smem specified by a constant. If they are all ones, TC is set.

BIT writes the value of Smem:bit to the TC.

BITT writes the value of Smem:bit postion specified by the T register to the TC.

Boolean Operations

Boolean Operations

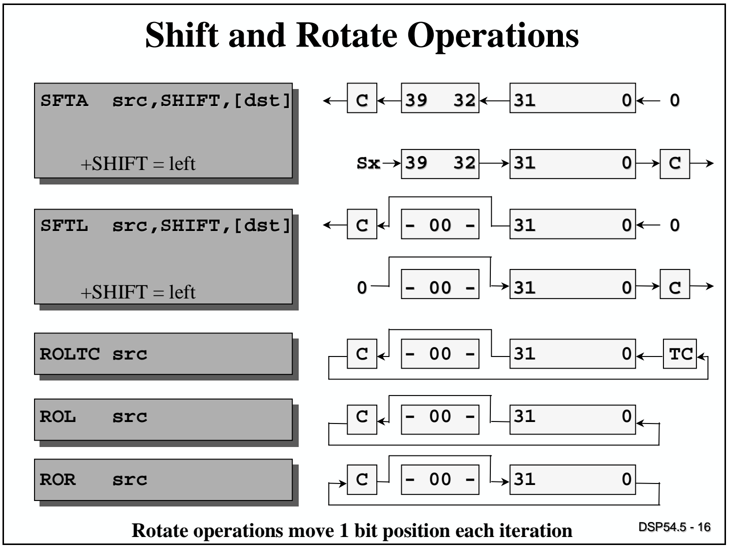
		Cycles
`AND/OR/XOR`	src = src (op) Smem	1
	dst = dst (op) (src << Shift)	2
	dst = src (op) (#K << Shift)	2
`ANDM/ORM/XORM`	Smem = Smem (op) #K	2

- src and dst: A or B accumulators
- ANDM/ORM/XORM perform a read - modify - write

DSP54.5 - 14

ANDM,ORM and XORM work directly to memory.

Shift and Rotate Operations



Some Other Math Operations …

Other Math Operations

`LDU`	**Load Unsigned**
`MPYU`	**Multiply (Unsigned * Unsigned)**
`MACSU`	**MAC (Signed * Unsigned)**
`ABS`	**Absolute Value**
`NEG`	**2's complement**
`CMPL`	**1's complement**
`EXP`	**T = (number of leading 1's or 0's) - 8**
`NORM`	**dst = src << T**
PMST$_{SMUL}$	**If `OVM/FRCT/SMUL` = 1, -1*-1 saturated to 00.7FFFFFFFh**

Unsigned operations are useful for > 16-bit multiplication
EXP/NORM are useful in floating point calculations

DSP54.5 - 18

SMUL is also known as GSM mode.

Review

Review

1. How is multiplicative overflow prevented?
2. How is accumulative overflow handled?
3. What processor bits should be set up for signed fractional math?
4. How does the processor round a number?
5. Do boolean operations only work on the accumulators?
6. What does “bit @y, 5” do?

DSP54.5 - 19

Solutions

Review

1. **How is multiplicative overflow prevented?**
   By using fractional math
2. **How is accumulative overflow handled?**
   Saturation: SAT or SST(bit) OR using a non-gain system OR OVM=1
3. **What processor bits should be set up for signed fractional math?**
   SXM=1 to preserve sign bit, FRCT=1 to eliminate redundant sign bit, OVM=0 to allow use of guard bits
4. **How does the processor round a number?**
   Adds 8000h to accumulator after operation is performed.
5. **Do boolean operations only work on the accumulators?**
   No. ANDM/ORM/XORM operate on memory directly
6. **What does "bit @y, 5" do?**
   Copies bit 10 from the value at address DP:@y into the TC bit

DSP54.5 - 20

Some Additional Information …

Division

Division

- The ‘C54x does *not* have a single cycle 16-bit divide instruction
  - Divide is a rare function in DSP
  - Division hardware is expensive
- The ‘C54x *does* have a single cycle 1-bit divide instruction: conditional subtract or `SUBC`
  - Preceded by `RPT #15`, a 16-bit divide is performed
  - Is *much* faster than without `SUBC`
- The `SUBC` process operates only on *unsigned* operands, thus software must:
  - Compare the signs of the input operands
    - If they are alike, plan a positive quotient
    - If they differ, plan to negate (`NEG`) the quotient
  - Strip the signs of the inputs
  - Perform the unsigned division
  - Attach the proper sign based on the comparison of the inputs

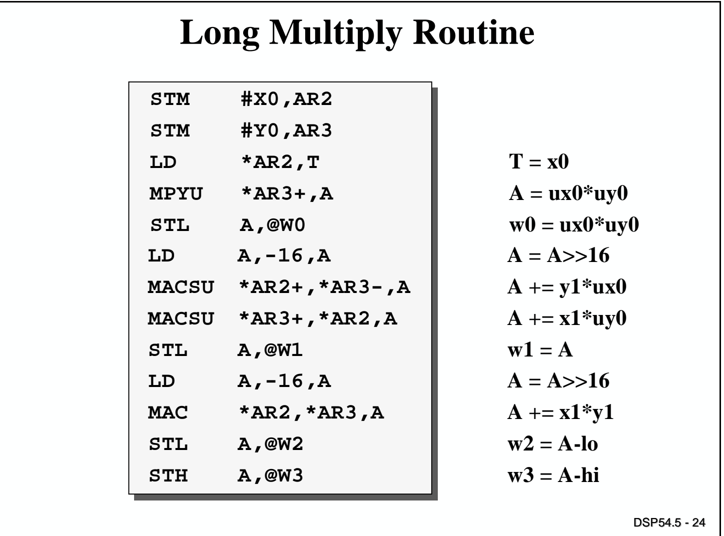
DSP54.5 - 22

Division Routine

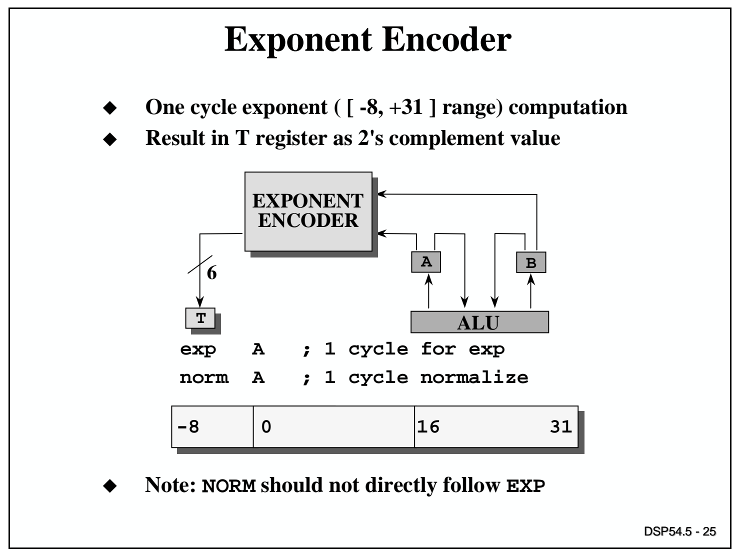
```
LD    @den,16,A
MPYA  @num        B = num*den (tells sign)
ABS   A           Strip sign of denominator
STH   A,@den
LD    @num,A
ABS   A           Strip sign of numerator

RPT   #15         16 iterations
SUBC  @den,A      1-bit divide

XC    1,BLT       If result needs to be negative
NEG   A           Invert sign
STL   A,@quot     Store negative result
```

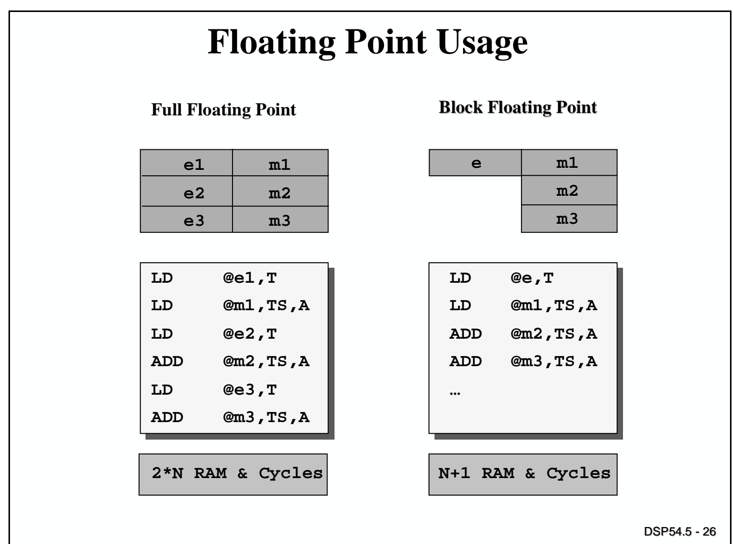
DSP54.5 - 23

Long Multiplies



Using Exponents




Floating Point Usage
Full Floating Point
Block Floating Point
e1
m1
e2
m2
e3
m3
e
m1
m2
m3
LD @e1,T
LD @m1,TS,A
LD @e2,T
ADD @m2,TS,A
LD @e3,T
ADD @m3,TS,A
LD @e,T
LD @m1,TS,A
ADD @m2,TS,A
ADD @m3,TS,A
…
2*N RAM & Cycles
N+1 RAM & Cycles
DSP54.5 - 26

Solving a Block FIR Filter

Introduction

We've already taken a look at the heart of a FIR filter; multiply-accumulates. Now let's extend that concept to include a large block of data. Of primary importance here will be how to manage my pointers and how to efficiently repeat a block of code.

Learning Objectives

Learning Objectives

- **Update pointers using `MAR` and Circular Addressing**
- **Use RPTB to repeat a block of code**
- **Describe how to nest repeat blocks**
- **Learn how to use parallel operations**

DSP54.6 - 2

Module Topics

Solving a Block FIR Filter.. 6-1

Module Topics 6-2

Block FIR Filters 6-3

Repeat Block.. 6-4

Wrapping the Pointers.. 6-5

Circular Addressing 6-6

What Have We Missed?.. 6-7
Single Sample FIR 6-7
Nesting Repeat Loops 6-8
Parallel Instructions.. 6-8

LAB6 – Block FIR 6-9

Objective 6-9

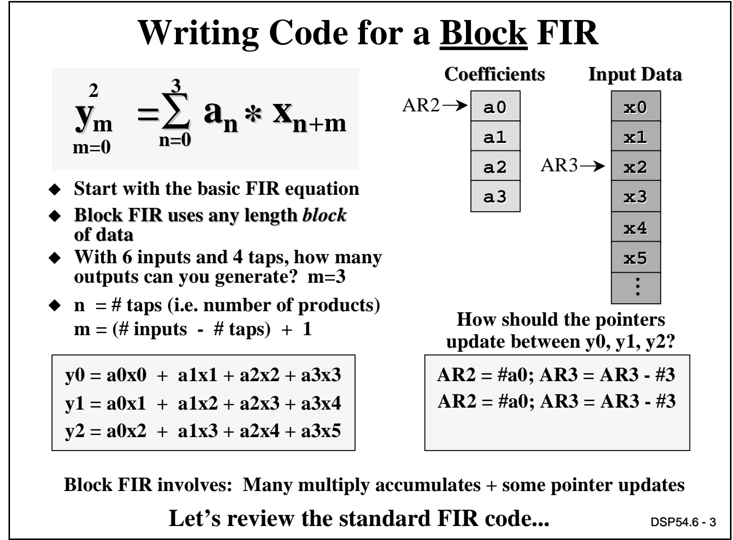
LAB6-A Procedure 6-10
Copy Files, Make Project and Edit LAB6A.CMD.. 6-10
Fractional Math, Repeat Block, Output Buffer – Write/Debug 6-10
Circular Addressing, Pointer Wrap – Write/Debug 6-11
Graph Your Results.. 6-11
Profile Your Code 6-13

LAB6-B Procedure 6-14

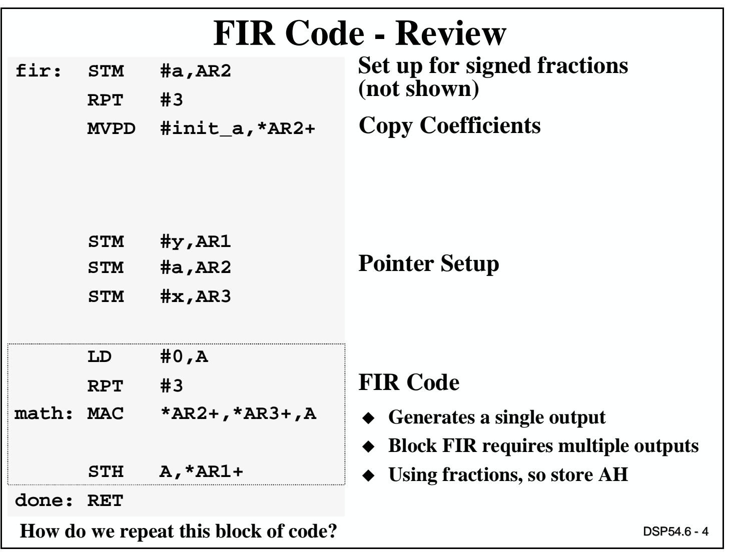
Benchmarking the Labs.. 6-15

Solutions.. 6-16

Block FIR Filters

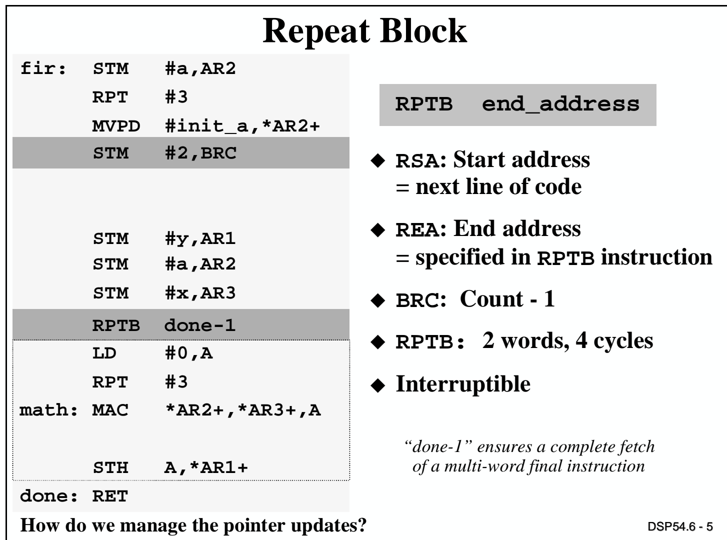


Obviously, this concept can be extended to any length block. You might wonder what happens when the filter reaches the end of the block. We'll cover that later …



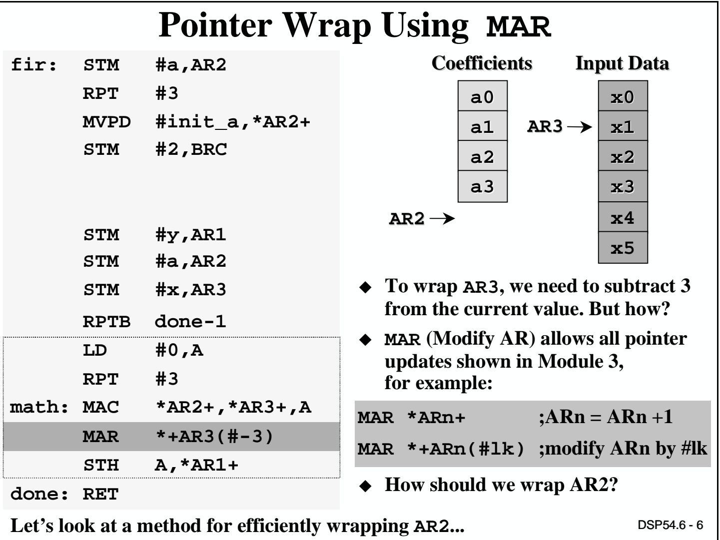
Is this the most efficiently to initialize the A. accumulator?

Repeat Block



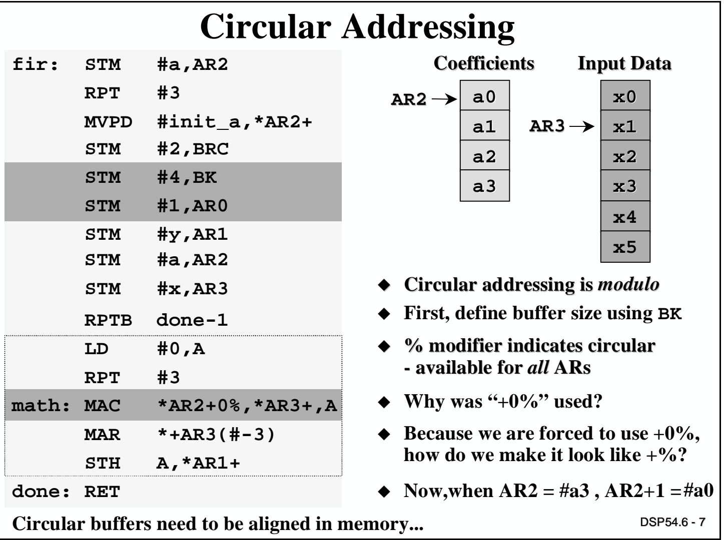
A RPTB (repeat block) may contain any length block up to 64 K. in length.

Wrapping the Pointers

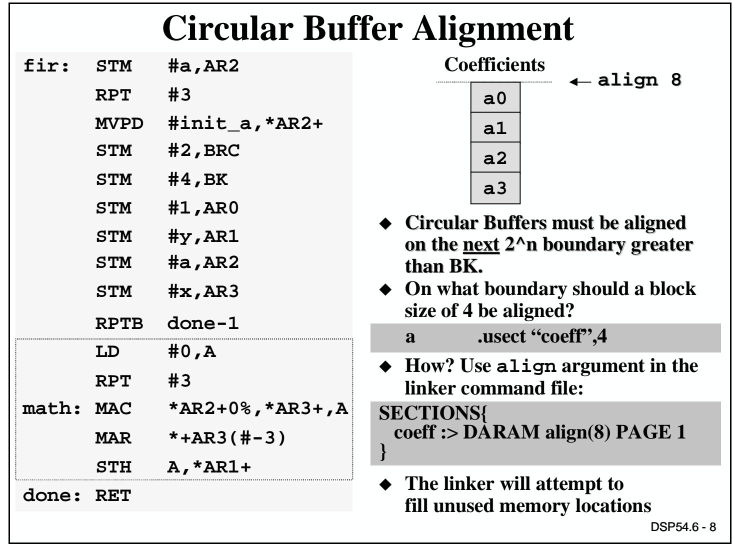


MAR allows you to modify the address register by +1 or by a long constant. AR2 0 always starts back at a0, so its modification can be little different.

Circular Addressing



Always remember to properly align your circular buffers.



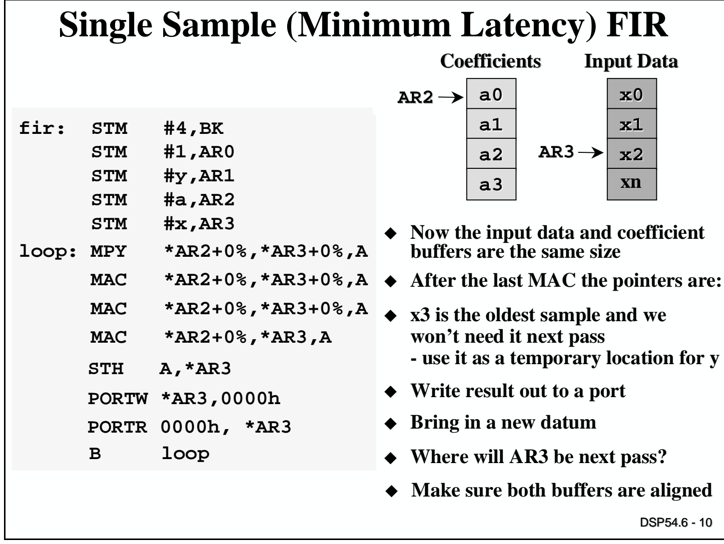
What Have We Missed?

What have we missed?

- ✓ How can I do a single sample FIR?
- ✓ How do I nest repeat operations?
- ◆ How can I optimize code using parallel instructions?

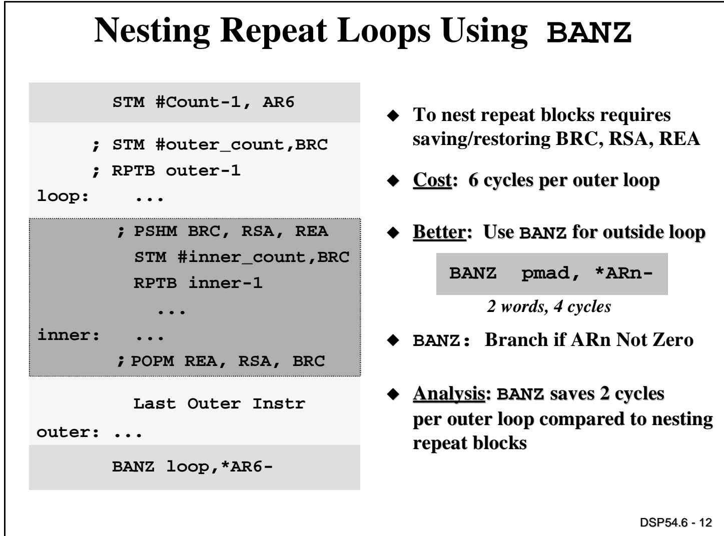
DSP54.6 - 13

Single Sample FIR



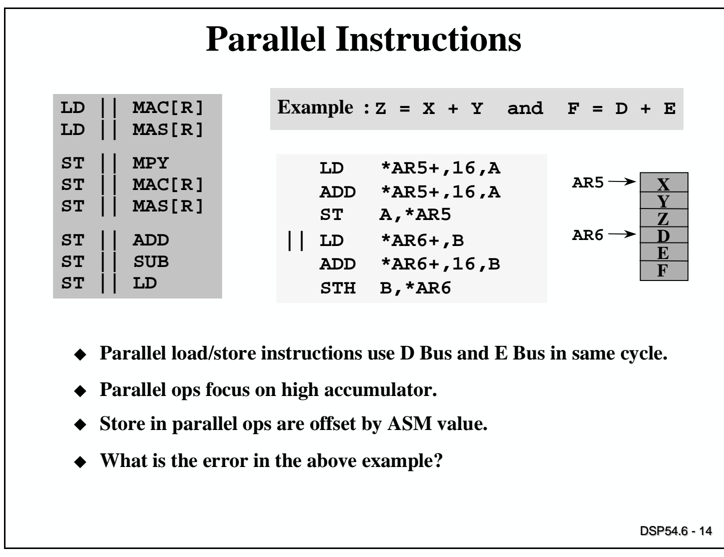
PORTR (port read) and PORTW (port write) operate from a memory location to a port address. In the code above, the oldest data point will not be used again so we can use this memory location as temporary storage. Obviously your code would not look quite like this, since this code will eat 100% of processor bandwidth.

Nesting Repeat Loops



So, while you can nest repeat blocks, it is usually more efficient to use the BANZ instruction.

Parallel Instructions

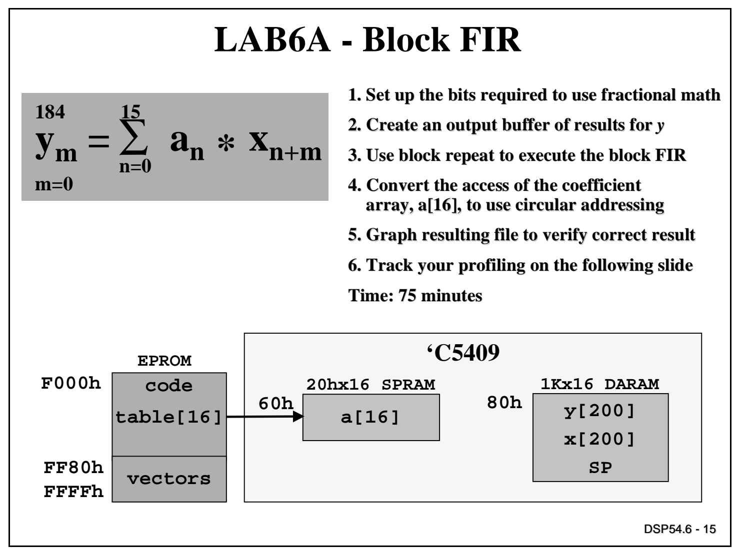


If both parts of a ST || LD instruction pointed to the same memory location the processor would operate as expected, swapping the accumulator and memory values.

LAB6 – Block FIR

Objective

The objective of this lab is to write code to perform a block FIR. The input file, `in6.dat`, contains the input values of the filter (which you plotted in LAB2A). The input file represents the summation of a high frequency and a low frequency sine wave. The objective of the block FIR is to filter out the high frequency component and pass only the low frequency wave. This lab will incorporate the numerical methods you learned in the previous module. The output should look like a low-frequency sine wave. If not, hmmm, debugging is in your future…



LAB6-A Procedure

Copy Files, Make Project and Edit LAB6A.CMD

1. Copy files from the last lab to LAB6A, make a new project called LAB6A and add the appropriate files to it.

2. Edit `LAB6A.CMD` and modify i/o as necessary.

Fractional Math, Repeat Block, Output Buffer – Write/Debug

1. Edit `LAB6A.ASM`.

2. We will be creating an output buffer for the results. So, change the uninitialized data section for your results (*y*) to a length of 200 instead of 1.

3. Change your `.copy` statement to `in6.dat` rather than `in4.dat`.

4. Set up the proper numerical bits to use fractional math. Make sure the following conditions exist in the main routine just before the `CALL` to fir:

   - Overflow Mode is OFF
   - Fractional Mode is ON
   - Sign-extension is ON
   - Set the `SST` bit in the `PMST` register to automatically saturate on store. If you use the `ORM` instruction to set this bit, remember that `ORM` means "OR to Memory", not "OR to Memory-mapped register". This implies that you will need to set the `DP` to the proper page. Which data page is `PMST` located in? Can't remember the format for the `ORM` instruction? Highlight "`ORM`" then hit the `<F1>` key.

3. Set up a repeat block to repeat the 16-TAP FIR code (including the store to *y*) the proper number of times. If you have 200 input values, how many output values should be generated? Use this value to determine how to initialize the block repeat counter. You will need a label on the return instruction to facilitate the block repeat.

4. Look at your store to *y* instruction. This instruction will store all results to ONE memory location. Change this instruction to create an array that contains ALL of the results.

5. Look at the store instruction again. Keeping in mind that you are using fractions, which half of the accumulator contains the correct result? Modify the store command if necessary.

6. Build `LAB6A`.

7. When the simulation opens, press `<F8>` until the `CALL copy` instruction is highlighted. Now, press the `Step Over` button on the vertical tool bar. This will execute the copy routine and return to the next instruction.

8. Open memory windows and verify that the coefficients (*a*) and data values (*x*) are correct.

   You should see the coefficient table in *a* and the input data values in *x*.

9. Open a memory window to view the result array buffer (*y*).

10. Single step your code through the store instruction in the *fir* routine. The first value stored to *y* should be `E404h`. If everything looks like it is operating correctly, move on to the next step.

Circular Addressing, Pointer Wrap – Write/Debug

11. Edit `LAB6A.ASM`. To wrap the coefficient pointer (pointing to *a*) automatically, implement circular addressing. Remember, `+%` is not a supported modifier in a dual-operand `MAC` instruction. What other registers need to be set up to make this work?

12. Circular buffers must be aligned. Make sure the section containing the coefficient table (*a*) is aligned properly in your linker command file. What boundary should an array of 16 values be aligned on?

13. The pointer to the input values (*x*) must also be wrapped back to the proper position. Write an instruction to perform this function just before the store to *y*. To determine the amount to subtract from the pointer, think about where the pointer is after the last `MAC` instruction executed and where the pointer needs to be for the next iteration of the block repeat. You may need to reference Module 3 to see the available modifiers.

14. Build `LAB6A`. When the simulator opens, set a breakpoint on the store instruction in the *fir* routine. Click the Run button on the vertical toolbar to run to the breakpoint. At this point, the first store to *y* has NOT been done yet, but we can see if the pointer wraps have worked properly.

15. Look at the address contained in the pointer to *a*. Is it pointing to the first value in *a*? Now, look at the address contained in the pointer to *x*. Is it pointing to the 2nd value in the data table? If not, debug the problem and re-verify.

16. Click the `Run` button to run through the block repeat again. This will write the first value to *y*. Look at the pointers to *a* and *x* again and verify that they have updated properly.

17. Remove the breakpoint on the store instruction and click `Run`. This will run the block FIR routine to its completion. After a few seconds, click `Halt` to halt at the stop condition.

Graph Your Results

18. Let's take a look again at the input to our filter. On the menu bar click:

    `View → Graph → Time/Frequency`

    Change the following fields:

Graph Title:	Input data
Start Address:	x
Acquisition Buffer Size:	200
Display Data Size:	200
DSP Data Type:	16-bit signed integer
Q Value:	15
Autoscale:	Off
Maximum Y-value:	1

19. Move the window to a convenient spot on your display.

20. Let's set up a display of the output with the following properties:

    Graph Title: Filtered Output
    Start Address: y
    Acquisition Buffer Size: 185
    Display Data Size: 185
    DSP Data Type: 16-bit signed integer
    Q Value: 15
    Autoscale: Off
    Maximum Y-value: 1

21. Explain why the result waveform amplitude is +/- ½.

22. Sometimes you might like to actually view your program in action. Additionally, you may not have a complete buffer of the information available to be graphed. CCS allows you to do this using a feature called "animation". Since this feature stops the program to transfer data, real-time performance may be impacted.

23. Let's wipe out our previous results by filling our result memory with zeros. From the menu bar click:

    `Edit → Memory → Fill.`

    And change the following fields:

    Address: y
    Length: 185

    Click OK. Check your memory display window to verify the fill has occurred.

24. Right click on your filtered output graph and click on `Clear Display`.

25. In `LAB6A.ASM` source window, click on the instruction that stores your result. On the vertical toolbar click on the `Toggle Probe-point` button

26. On the menu bar, click:

    `Debug → Probe-points`

    and highlight the only probe point listed in the bottom window. Take care not to uncheck the box on the left. Change the following fields:

    Probe type: Probe on Data Write at Location
    Connect To: Filtered Output

    Note the different types of probes you can use and that you can use expressions. Click `Add`, then delete the probe point with no connection. Click OK.

27. Restart your project. On the vertical toolbar, click the `Animate` button and watch the display draw as the program runs.

28. Probe points can also run on hardware targets, but RTDX has less impact on real-time operation. Probe points may also be used to connect data i/o files to your program. Look in the online help and see how this is done.

Profile Your Code

29. Once you have a clean graph, it is time to profile your code. Remove the Probe point, then set the 1st profile-point on the `CALL fir` instruction and the 2nd profile-point on the stop condition. Now, restart and profile your code. Write down your time on the sheet provided at the end of this module. It should be about 11470 cycles.

30. Does 11470 seem right? Well, it IS correct, but what did you expect it to be? Estimate the cycle count for the `RPTB` loop by counting up the cycles inside the loop and multiplying by the `RPTB` loop count. Write down your estimate on the sheet provided.

31. Why was the cycle count so much higher than your estimate? Did you forget to run “0ws”?

32. In a real application, you need to setup the external wait state generator to the appropriate values based on your system requirements. However, because we are using the simulator, let’s simply assume that everything external is zero-wait states. At the beginning of `LAB6A.ASM`, type in the following instruction:

    ```
    STM #0, SWWSR
    ```

    Issues associated with wait states and memory interfacing will be covered in later modules.

33. Build and simulate your code again. Now, profile your `fir` routine. What was your cycle count? Write it down on the sheet provided. The cycle count should be around 3700 cycles.

34. One last note: open up `LAB6A.ASM`. Note the values of the coefficients. Notice anything interesting? If you add all 16 coefficients together, they equal almost one (on a fractional scale). What does this imply?

    It implies that the filter has a gain less than one. Therefore, saturation is not required, because the filter is guaranteed (by design) to not overflow because the coefficients added together are less than unity AND every input to the filter is less unity – no gain. Hey, this stuff really works ☺.

35. If you’re done with `LAB6A` and you still have some time, move on to `LAB6B`…

LAB6-B Procedure

If you make any changes to `LAB6A.ASM` or `LAB6A.CMD`, first copy the files to `LAB6B.ASM` and `LAB6B.CMD`. Answers to the following questions are either contained in the on-line documentation or via your instructor.

1. Negate all 16 coefficients. In other words, use the 2's complement of all of the coefficients. Re-run your code. What changed and why?
2. Change your repeat block to use BANZ. Profile your code. Any difference?
3. How do you terminate a block repeat? Can you perform a branch or call inside a block repeat? Hmmm. Look at the BRAF bit. What does it do? Do you think this might assist you in terminating a block repeat?
4. Let's make the entire code interruptible. Replace repeat single with a repeat block and nest the repeat blocks.
5. Look at your code. Can you take advantage of using parallel instructions? Why/why not?
6. Display the contents of REA and RSA during the block repeat loop.

Benchmarking the Labs

Block_FIR Profiling

LAB	Profile	#cycles
6a	MAC, first profile	
6a	MAC, estimate	
6a	MAC, w/SWWSR=0	
8a	FIRS, w/SWWSR=0	

keep this form handy for all the labs to compare your #'s

DSP54.6 - 16

Solutions

LAB6A.ASM - Solution

```
        .mmregs
        .def start

STKLEN  .set    100

a       .usect "coeffs",16,1
y       .usect "result",200
BOS     .usect "STK",STKLEN

        .sect "init"
table   .int 7FCh,7FDh,7FEh,7FFh
        .int 800h,801h,802h,803h
        .int 803h,802h,801h,800h
        .int 7FFH,7FEH,7FDH,7FCH

        .sect "indata"
x       .copy "in6.dat"

        .sect "code"
start:  STM     #BOS+STKLEN,SP   ;setup stack pointer
        STM     #0,SWWSR         ;set ext'l wait state to zero
        LD      #0,DP            ;set SST bit (saturate on store)
        ORM     #1,@PMST
        SSBX    FRCT             ;set FRCT bit (fractional mode)
        RSBX    OVM              ;clr OVM bit (overflow mode)
        SSBX    SXM              ;set SXM bit (sign extension)
```

DSP54.6 - 18

LAB6A.ASM - Solution (Continued)

```
        CALL    copy
        CALL    fir

here:   B       here

copy:   STM     #a,AR1           ;setup AR1
        RPT     #15              ;copy 16 values
        MVPD    #table,*AR1+
        RET                      ;return

fir:    STM     #184,BRC
        STM     #16,BK
        STM     #1,AR0
        STM     #a,AR2           ;setup ARs for MAC
        STM     #x,AR3
        STM     #y,AR4

        RPTB    done-1
          MPY   *AR2+0%,*AR3+,A   ;1st product
          RPT   #14               ;mult/acc 15 terms
            MAC *AR2+0%,*AR3+,A
          MAR   *+AR3(-15)
          STH   A,*AR4+          ;store result

done:   RET                      ;return
```

DSP54.6 - 19

LAB6A.CMD : Solution

```
/* file I/O and options       */
vectors.obj
lab6a.obj
-m lab6a.map
-o lab6a.out
-e start

MEMORY {
PAGE 1:        /* Data memory */
  SPRAM:    org = 00060h,  len = 00020h
  DARAM:    org = 00080h,  len = 00400h

PAGE 0:        /* Program memory */
  EPROM:    org = 0F000h,  len = 00F80h
  VECS:     org = 0FF80h,  len = 00080h
}

SECTIONS
{ coeffs         :> SPRAM   PAGE 1
  result         :> DARAM   PAGE 1
  indata         :> DARAM   PAGE 1
  STK            :> DARAM   PAGE 1
  code           :> EPROM   PAGE 0
  init           :> EPROM   PAGE 0
  vectors        :> VECS    PAGE 0
}
```

DSP54.6 - 20

Pipeline Implications

Introduction

Most microprocessors and DSP's are pipelined in some fashion. The C54x differs in that its pipeline is open. When we write to control type registers it can have a profound and sometimes an out-of-sequence affect on CPU operation. The responsibility for proper pipeline operation rests on the programmer. Fortunately you have both assembler and simulator latency detection tools at your disposal.

Learning Objectives

Learning Objectives

- **Describe the 'C54x pipeline events**
- **Implement delayed operations**
- **Identify and resolve pipeline conflicts**

DSP54.7 - 2

Module Topics

Pipeline Implications **7-1**

Module Topics *7-2*

Delayed Instructions *7-3*

The Pipeline *7-4*

Understanding the Impact on the Pipe *7-5*

Writing Early *7-6*

Determining Latency Cycles *7-7*

Latency Tables *7-8*

Review *7-10*

Exercises *7-11*

LAB7 – Latency Issues **7-12**

Objective 7-12

LAB7-A Procedure *7-13*

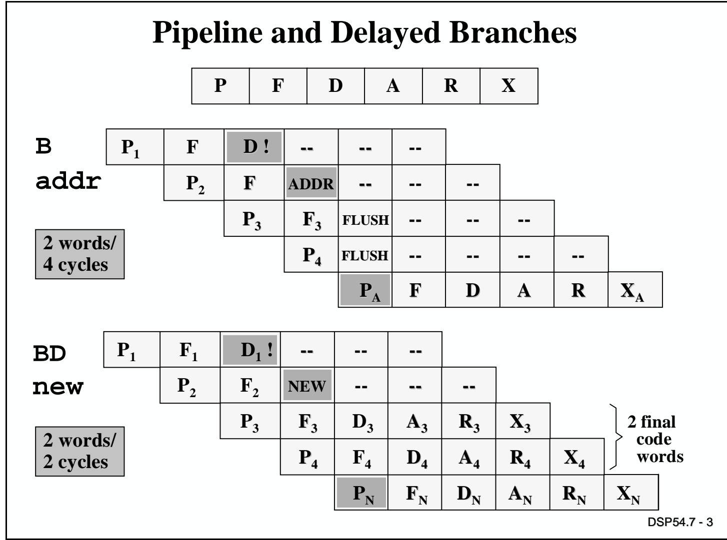
Fix Latencies in `LATENCY.ASM` and `LAB6A.ASM` 7-13

LAB7-B Procedure *7-14*

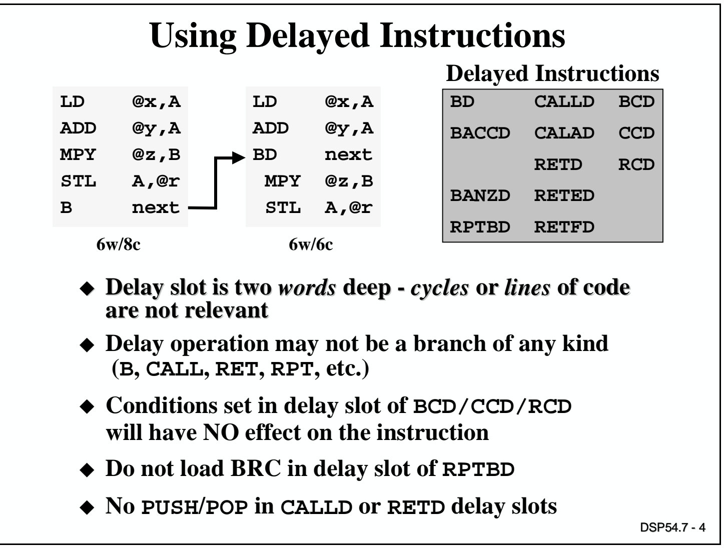
Solutions *7-15*

Additional Information… *7-16*

Delayed Instructions



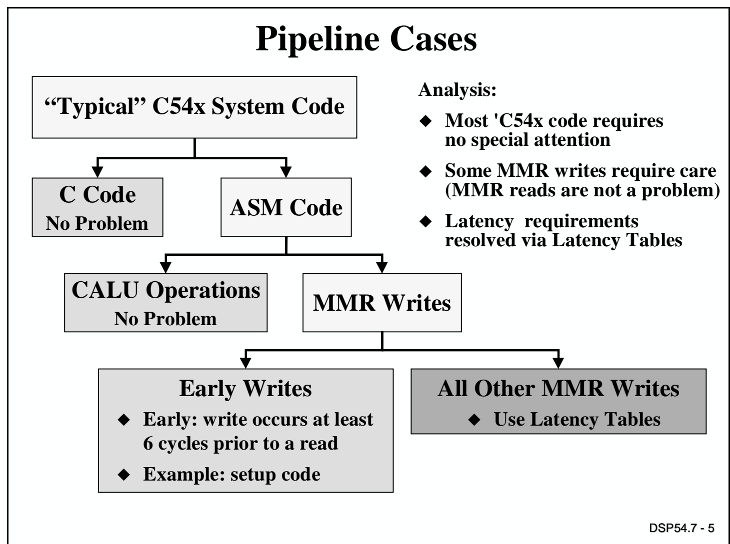
A delayed branch merely allows the 2 words that have already been prefetched to run to completion.

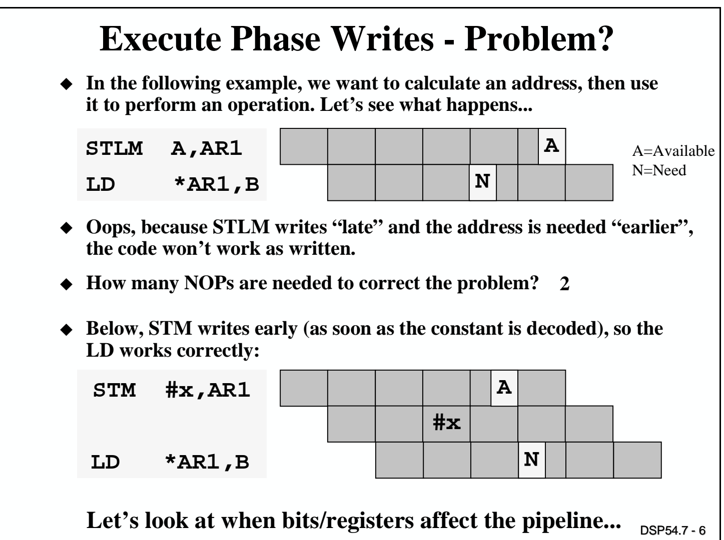


The problem is, some operations may not be in the delay slot hand and since the next two words will be executed before the branch is taken, debugging can be difficult.

Save the use of delayed branches for those situations where saving two cycles is very important.

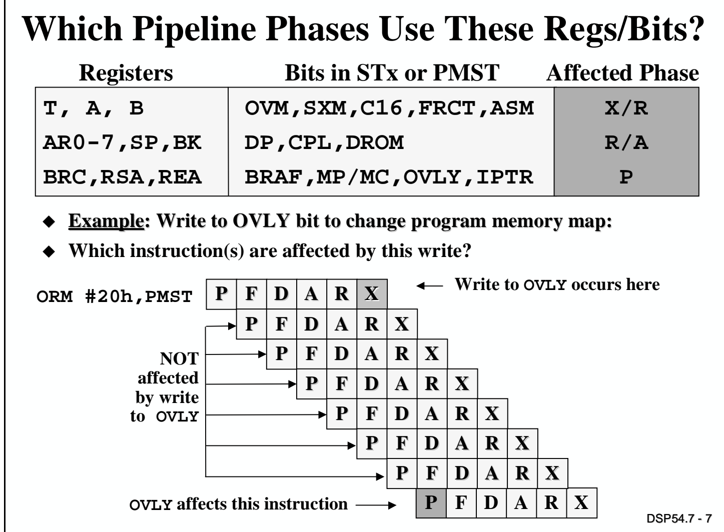
The Pipeline



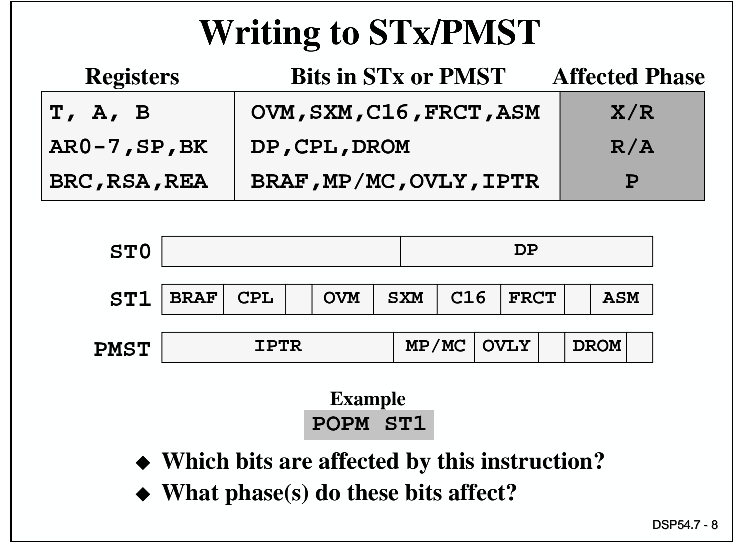


Most instructions complete their operations in the execute phase. Because of potential pipeline latencies, it would be nice to have some instructions that operate early.

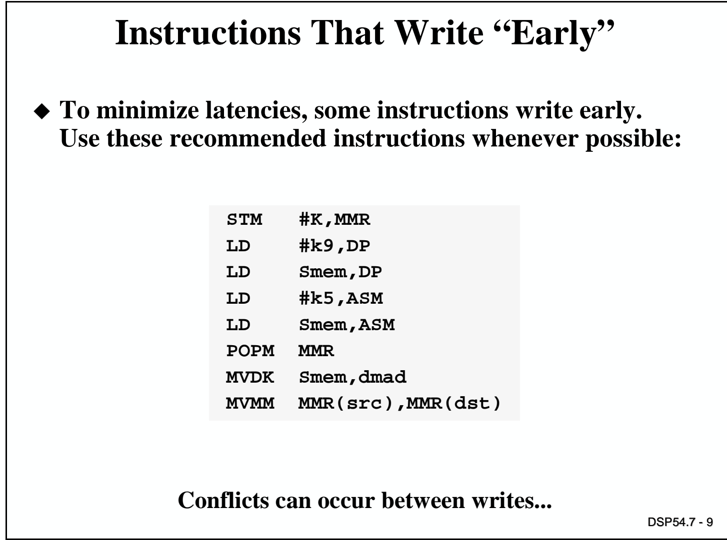
Understanding the Impact on the Pipe



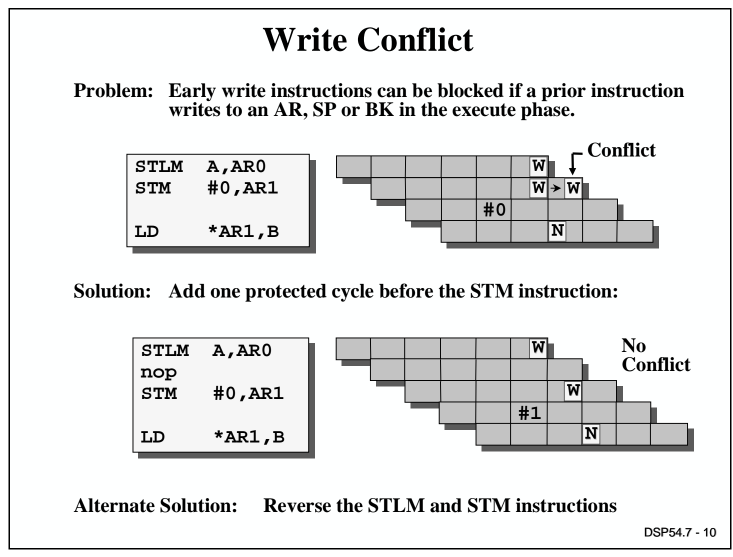
This breaks up the effect of the pipeline into three categories; CALU, data address generation and program address generation.



Writing Early

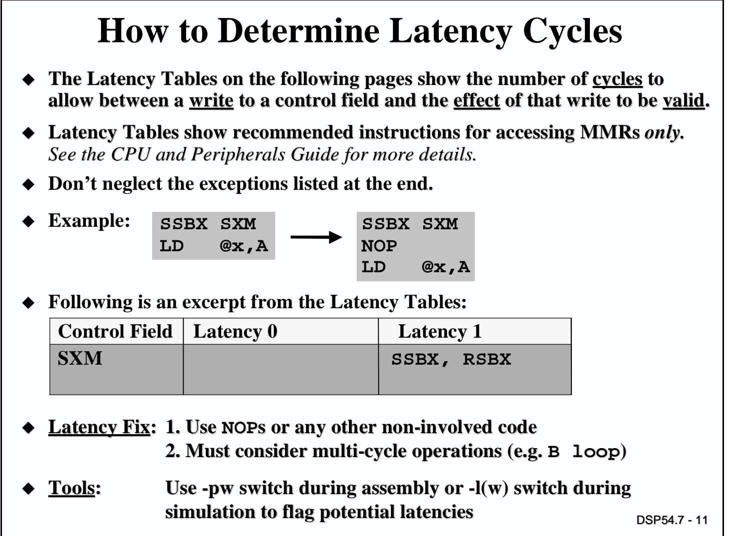


The problem arises when two instructions; one operating in the execute phase and the other operating earlier attempt to write to ARx, SP or BK. If these two writes overlap, they will conflict since there is only one path to these registers.



In the latency chapter of the CPU and Peripherals user guide you'll find a rule stating not to precede an instruction like STM with one that writes to these registers in the execute phase.

Determining Latency Cycles



The following tables do not and cannot show all possible latencies. Please refer to the CPU and peripherals guide for more detail

Latency Tables

Latency Tables - Recommended Instructions

Control Field	Latency 0	Latency 1
T	STM, MVDK LD Smem,T LD Smem,T || ST	STLM, STL, STH, EXP
ASM	LD #k5,ASM LD Smem,ASM	
DP	LD #k9,DP LD Smem,DP	
SXM		SSBX, RSBX
A or B	All *except* ...	Modify accumulator then read as MMR
BRC before RPTB[D]	STM MVDK	STLM, STL, STH

DSP54.7 - 12

Latency Tables - Recommended Instructions

Control Field	Latency 0	Latency 1	Latency 2	Latency 3
ARx	STM MVDK MVMM,MVMD MAR	POPM	STLM,STH,STL	
BK		STM MVDK MVMM,MVMD	POPM	STLM,STH,STL
SP	if CPL = 0 STM MVDK MVMM MVMD	if CPL = 1 STM MVDK MVMM MVMD	if CPL = 0 STLM STH STL	if CPL = 1 STLM STH STL
Implicit SP changes when CPL = 1		FRAME POPM/POPD PSHM/PSHD		

DSP54.7 - 13

Latency Tables - Recommended Instructions

Control Field	Latency 3	Latency 4	Latency 5	Latency 6
DROM	ANDM ORM XORM			
CPL	RSBX SSBX			
BRAF			RSBX SSBX	
OVLY IPTR MP/MC-				ANDM ORM XORM

DSP54.7 - 14

Latency Table Notes

- Do not precede STM,MVDK or MVMD with an instruction (e.g. STLM) that writes to <u>any</u> ARx, BK or SP in the execute phase of the pipeline.
- After altering the BRAF bit, the next 6 cycles must not contain the last instruction word in the RPTB[D] loop.
- SRCCD must be located at least 2 cycles before the last instruction of the RPTB[D] loop.
- When changing OVLY,MP/MC- or IPTR, latency listed is to fetch the first instruction from the newly activated memory space

DSP54.7 - 15

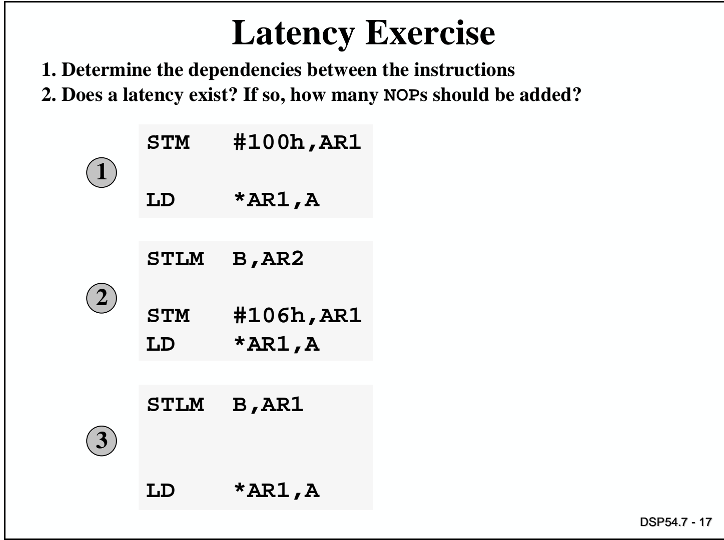
Review

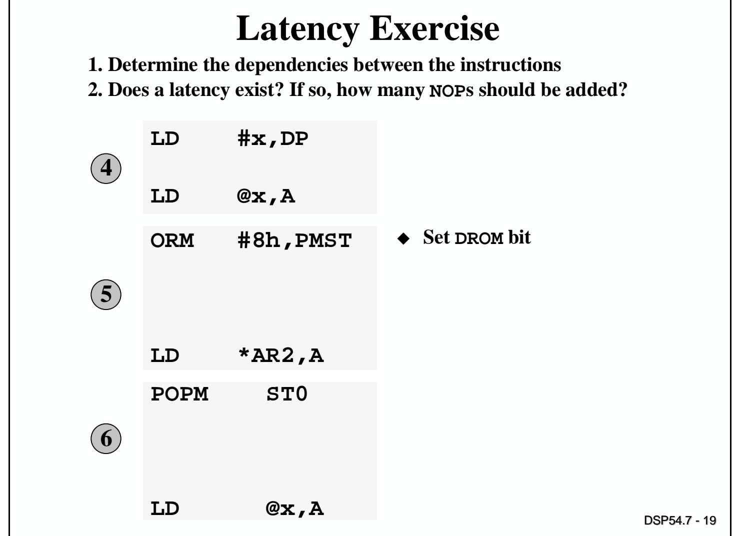
Latency Issues - Review

- *No* latency for CALU operations
- Write to MMRs early whenever possible
- Set status early
- Use latency tables when writing to MMRs
- For debug: *focus on late MMR writes*
- Reference Guide has chapter on pipeline use

DSP54.7 - 16

Exercises





LAB7 – Latency Issues

Objective

The objective of this lab is to find and fix latency issues inside two files: LATENCY.ASM and LAB6A.ASM. Use the latency tables and the `-pw` assembler switch to aide the process.

LAB7A - Latency Issues

1. **Add `-pw` switch to your assembler options.**
2. **Open and view `LATENCY.ASM`.**
3. **Determine the latencies and the #`NOP`s needed to fix the problem. Write a comment next to the instruction of what you expect the solution to be.**
4. **Assemble `LATENCY.ASM` using `-pw` switch and note the warnings. Compare the warnings with your expectations.**
5. **Fix the latency issues and re-assemble.**
6. **Check and fix any latencies in `LAB6A.ASM`. You may or may not have any.**

Time: 45 minutes

DSP54.7 - 21

LAB7-A Procedure

Fix Latencies in LATENCY.ASM

1. Create a new project called `Latency.` Add `LATENCY.ASM` to it.
2. On the menu bar, select:

   `Project → Options`

   Then, select the `Assembler` tab and add `-pw` to the command line switch box at the top of the window. This switch will enable pipeline warnings during assembly. Click `OK`.
3. Open `LATENCY.ASM` for editing. Determine the latencies and the #`NOP`s needed to fix the problems (if they exist). Write a comment next to the instruction describing what you expect the solution to be.
4. Assemble `LATENCY.ASM` by clicking the `Compile` button on the vertical toolbar and note the warnings shown in the output window at the bottom of your screen. Compare the warnings with your expectations.
5. Fix the latency issues and re-assemble.

Fix Latencies in LAB6A.ASM

6. When `LATENCY.ASM` is “clean”, re-load project `LAB6A.MAK`. Check and fix any latencies in `LAB6A.ASM`. You may or may not have any. If you find latencies, explain why your code worked anyway.
7. If you’re done with the above steps and you still have some time, move on to LAB7B…

LAB7-B Procedure

1. Open `LATENCY.ASM`. Type in the following instructions, write down your expectations, assemble and fix the latency issues. If the assembler gives no warning on an instruction which you thought should be a problem, explain why it is not a problem:

   ```
   POPM PMST
   RETF

   STL A,*AR3
   LD *AR3,B

   STL A,*AR3+
   LD  #0,A
   ADD *AR4, *AR5, A

   STL A,*AR3+
   STH A,*AR3
   ADD *AR4, *AR5, A
   ```

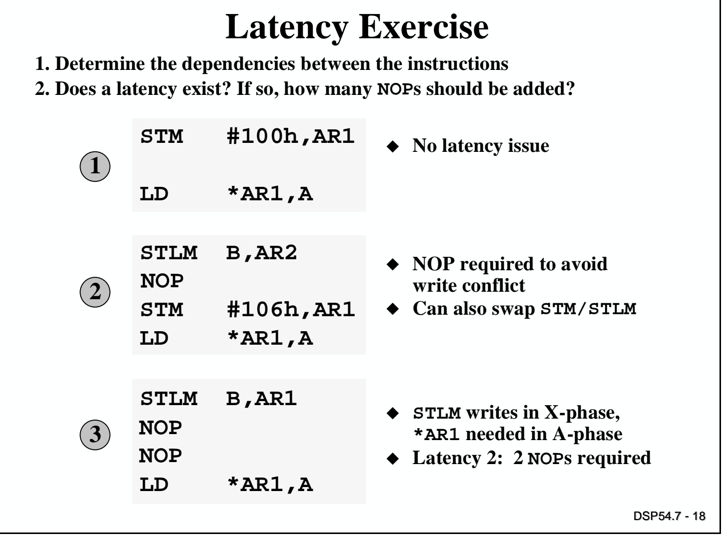
   You might find the additional material at the end of the module helpful.

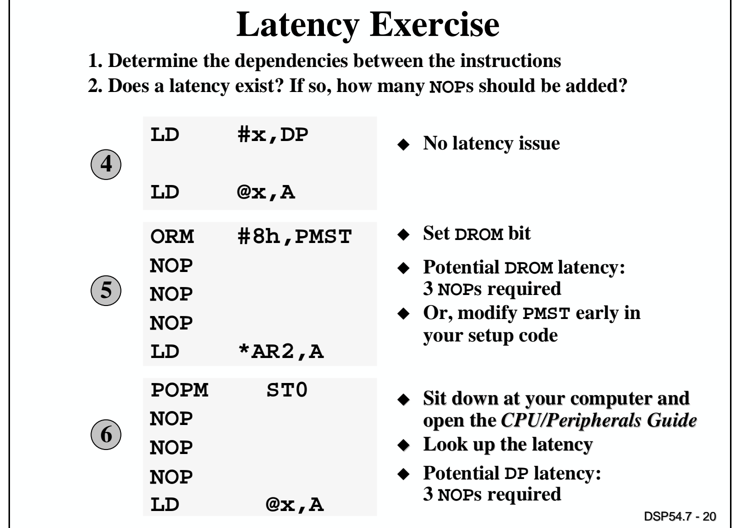
2. Write an execute conditional (`XC`) instruction to perform some task. What are the implications of using this instruction?

3. Delayed operations (like BD) take 2 cycles less than their non-delayed counterparts. Why not use them all the time?

4. Open `LAB6A.ASM` and modify using delayed operations wherever possible. Assemble, link and simulate. Verify your plot. Make sure you have working code because this lab will be copied to later labs.

Solutions



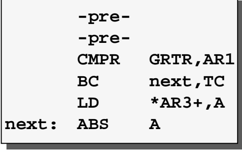


Additional Information…

Conditional Execution: XC

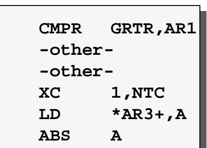
- Allows <u>fast</u> choice of running one or two words of code or substitution of NOPs.
- Condition evaluated early, so must be set <u>two</u> cycles prior.
- Avoid change of condition in last two lines prior to XC, as they can be recognized in event of interrupt prior to XC.

```
XC  n,cnd,cnd,cnd
```



```
       -pre-
       -pre-
       CMPR   GRTR,AR1
       BC     next,TC
       LD     *AR3+,A
next:  ABS    A
```

3 words, 5/4 cycles



```
       CMPR   GRTR,AR1
       -other-
       -other-
       XC     1,NTC
       LD     *AR3+,A
       ABS    A
```

2 words, 2 cycles

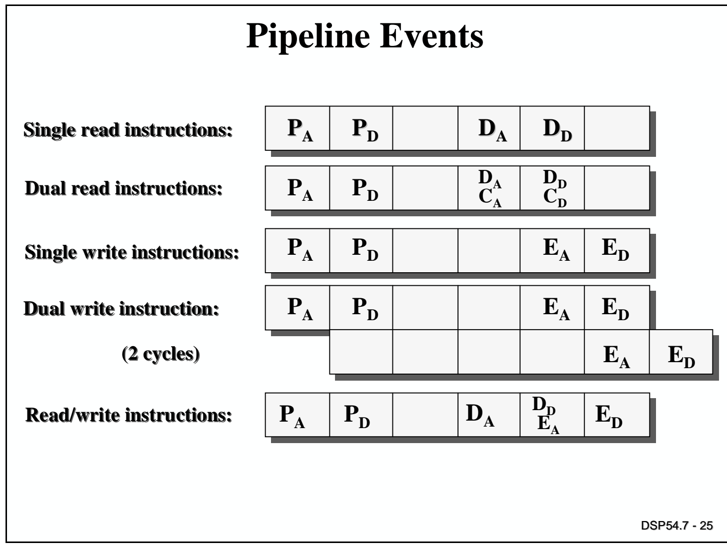
DSP54.7 - 23

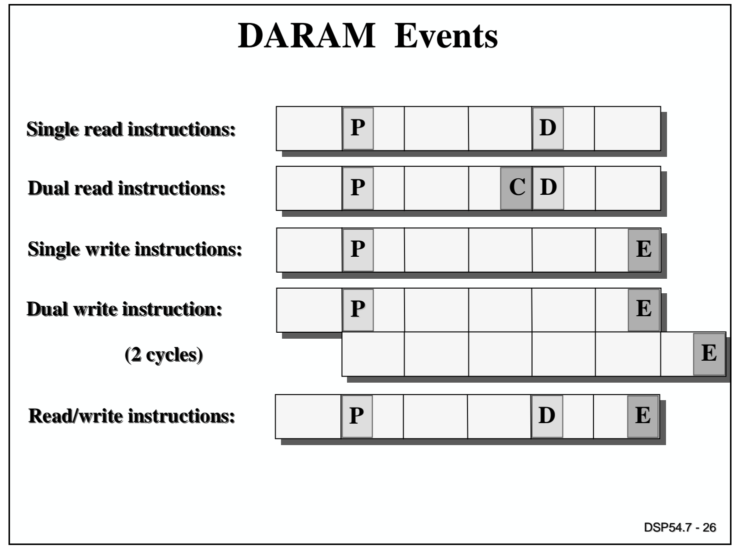
CALU Operations - Analysis

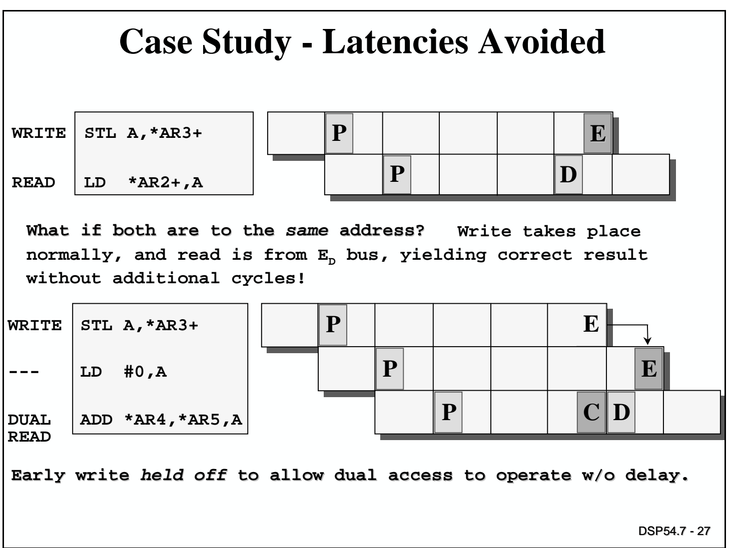
- The 'C54x may need to perform a fetch, two reads, and a write in any given cycle. Depending on the system setup, this event could occur in one cycle or be spread over several cycles. *In no case* are errors generated. Consider the following environments:
  - More than one external access: multiple cycles
  - Each resource in separate memories: single cycle
    - Note: 'C54x memories are broken into blocks.
  - More than one resource in a single 'C54x memory block - Dual Access RAM :

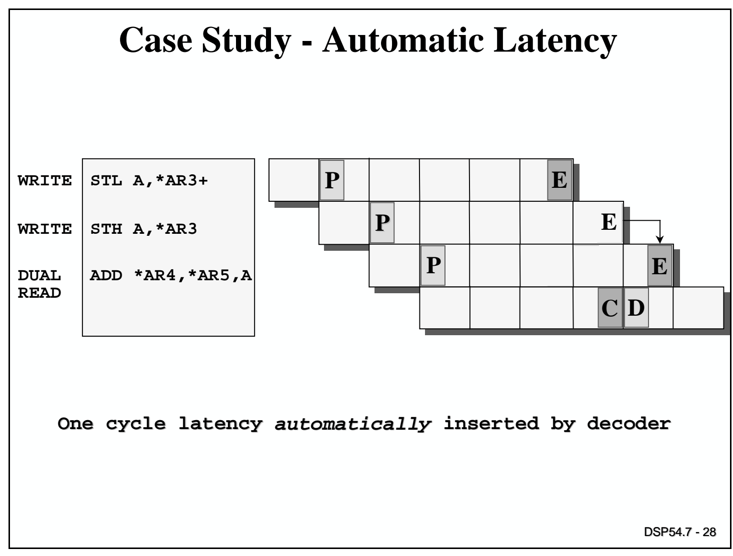
Early phase	P and D
Late phase	C and E

DSP54.7 - 24


Pipeline Events
Single read instructions:
Dual read instructions:
Single write instructions:
Dual write instruction:
(2 cycles)
Read/write instructions:
DSP54.7 - 25


DARAM Events
Single read instructions:
Dual read instructions:
Single write instructions:
Dual write instruction:
(2 cycles)
Read/write instructions:
DSP54.7 - 26


Case Study - Latencies Avoided
WRITE STL A,*AR3+
READ LD *AR2+,A
What if both are to the same address? Write takes place normally, and read is from ED bus, yielding correct result without additional cycles!
WRITE STL A,*AR3+
--- LD #0,A
DUAL READ ADD *AR4,*AR5,A
Early write held off to allow dual access to operate w/o delay.
DSP54.7 - 27



Case Study - Automatic Latency
WRITE
STL A,*AR3+
WRITE
STH A,*AR3
DUAL READ
ADD *AR4,*AR5,A
P
E
P
E
P
E
C
D
One cycle latency automatically inserted by decoder
DSP54.7 - 28

Application Specific Instructions

Introduction

A strictly general-purpose processor can only offer average performance. In order to extract the maximum performance from a device you need instructions that can accelerate certain algorithms. Since the C54x is targeted for the telecom marketplace, there are several powerful instructions that enhance performance for our algorithms typically used in telecom.

Learning Objectives

Learning Objectives

- **Describe the basic operation of specific algorithms**
- **Associate certain instructions to the chosen algorithm**
- **Identify the architectural components that provide advanced performance**

DSP54.8 - 2

Module Topics

Application Specific Instructions.. 8-1

Module Topics.. 8-2

Symmetric FIR.. 8-3

Least Mean Square.. 8-5

Minimum and Maximum.. 8-6

Some Other Useful Instructions.. 8-7

Additional Resources.. 8-8

LAB8 – Block FIR 8-9

Objective.. 8-9

LAB8A - Procedure 8-10

Copy Files, Edit LAB8A.CMD.. 8-10
Edit LAB8.ASM – Write/Debug.. 8-10
Build, Simulate, Verify 8-11

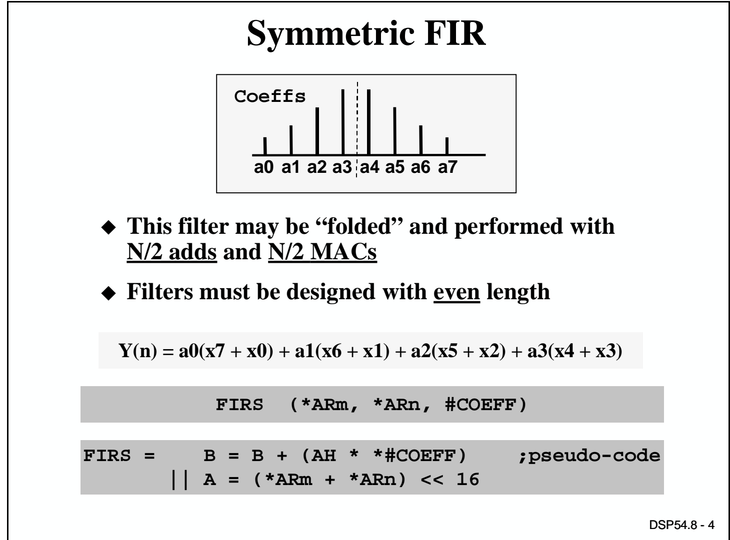
LAB8-B Procedure 8-12

Solutions.. 8-13

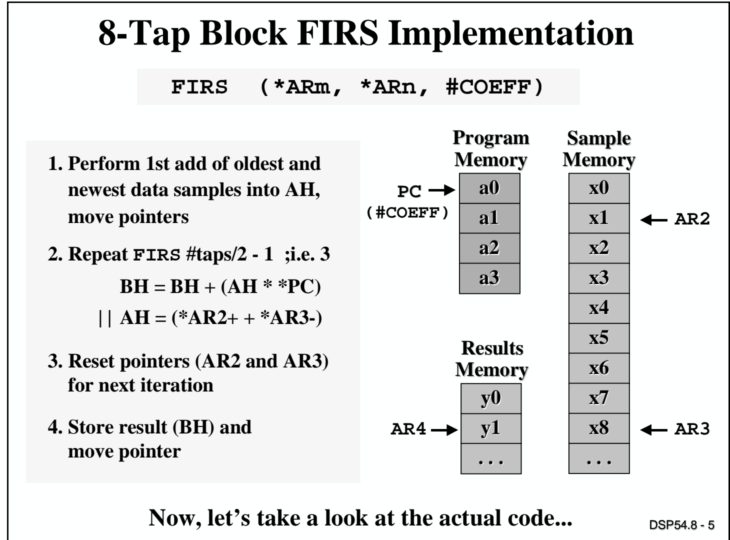
Additional Information.. 8-15

LMS Loading.. 8-15
Codebook Search 8-15
Viterbi Decoding.. 8-16
Determining Metrics 8-17
Polynomial Evaluation.. 8-18

Symmetric FIR



The FIRS instruction performs the multiply and accumulate as well as the summation of the next two data points in a single cycle when in a single repeat.



Your pointers will need to point of both the oldest and the newest samples. You'll also have to perform the first add before entering the repeat single FIRS loop.

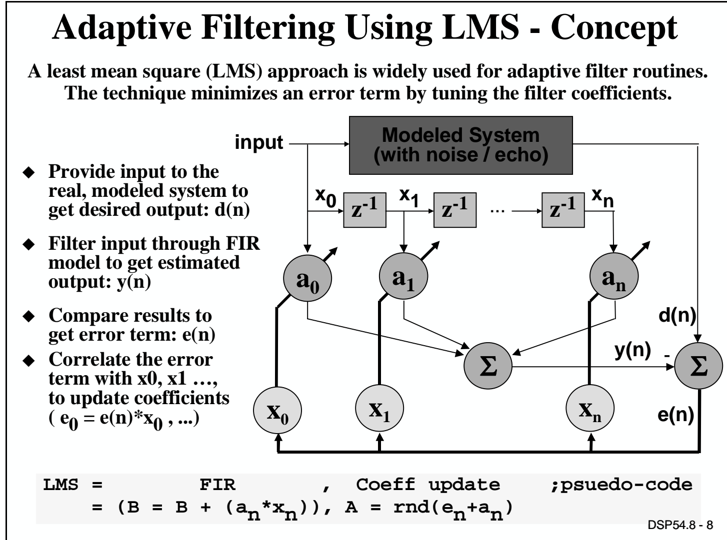
FIRS Code Example

```
        .asg    AR2,old
        .asg    AR3,new
        .asg    AR4,results
        SSBX    FRCT                    ;set FRCT
        STM     #samps-taps+1-1,BRC     ;setup BRC
        STM     #x,old                  ;old = #x
        STM     #x+7,new                ;new = #x+7
        STM     #y,results              ;results = #y
        STM     #-4,AR0                 ;pointer wrap for old

FIR:    RPTB    done-1                  ;Repeat Block FIR
         ADD    *old+,*new-,A           ;First sum for FIRS << 16
         RPTZ   B,#3                    ;B=0, Repeat taps-1
          FIRS  *old+,*new-,#COEFF      ;MAC, next sum
         MAR    *old+0                  ;Wrap old by -4
         MAR    *+new(#6)               ;Wrap new by +6
         STH    B,*results+             ;Store result to y
done:
```

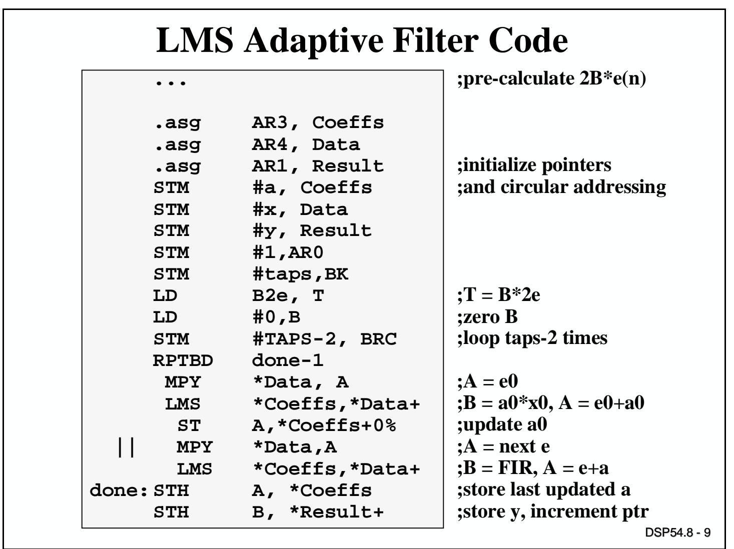
DSP54.8 - 6

Notice how we used AR0 to save a cycle resetting the old pointer.

Least Mean Square



An example of this might be a noise-canceling headset. The modeled system is the earpiece containing your ear, a speaker and a microphone to “listen” to what you are hearing. Add noise from an external source. Now think of the input as the music from your stereo. That signal is what we’d like to hear (monitored by the microphone (d(n))). In order for the signal inside the earpiece to just be music, the FIR filter will adapt itself to produce anti-noise (a signal out-of-phase) with the external noise. If the speaker is driven with this signal, music is all we’ll hear.

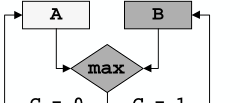



```
      ...                                  ;pre-calculate 2B*e(n)

      .asg    AR3, Coeffs
      .asg    AR4, Data
      .asg    AR1, Result                  ;initialize pointers
      STM     #a, Coeffs                   ;and circular addressing
      STM     #x, Data
      STM     #y, Result
      STM     #1,AR0
      STM     #taps,BK
      LD      B2e, T                       ;T = B*2e
      LD      #0,B                         ;zero B
      STM     #TAPS-2, BRC                 ;loop taps-2 times
      RPTBD   done-1
       MPY    *Data, A                     ;A = e0
       LMS    *Coeffs,*Data+               ;B = a0*x0, A = e0+a0
        ST    A,*Coeffs+0%                 ;update a0
  ||    MPY   *Data,A                      ;A = next e
        LMS   *Coeffs,*Data+               ;B = FIR, A = e+a
done: STH     A, *Coeffs                   ;store last updated a
      STH     B, *Result+                  ;store y, increment ptr
```

Minimum and Maximum

MIN, MAX

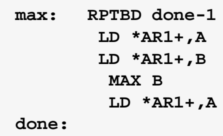
- **Goal: find the max (or min) value in an array**

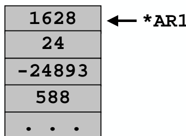



```
MAX dst
MIN dst
```

dst: **A or B**

- **Example: Find the max value in this array**



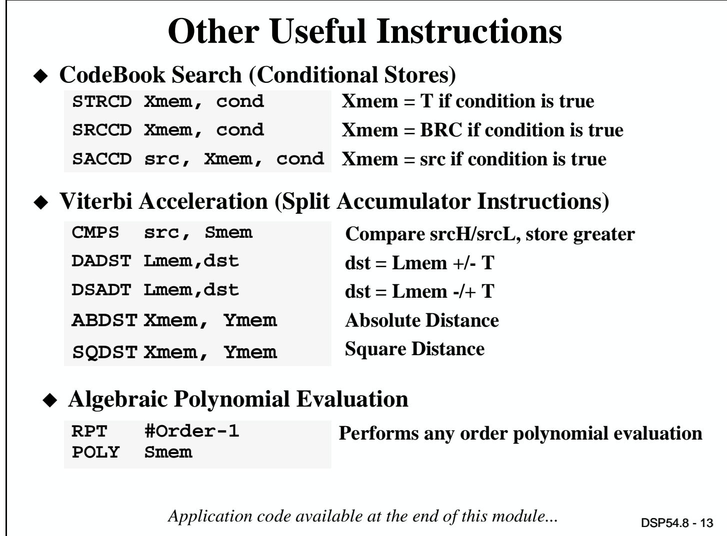


*AR1

- **Benchmark: ~2N cycles to find the min/max of N elements**

DSP54.8 - 11

Some Other Useful Instructions



Viterbi decoding, convolutional encoding and codebook search operations are used extensively in cellular communications. If you are interested in learning more, ask your instructor to give you a short presentation.

Additional Resources

Additional Resources

1. S. M. Redl, M. K. Weber, M. W. Oliphant, “An Introduction to GSM”, Artech House, 1995.

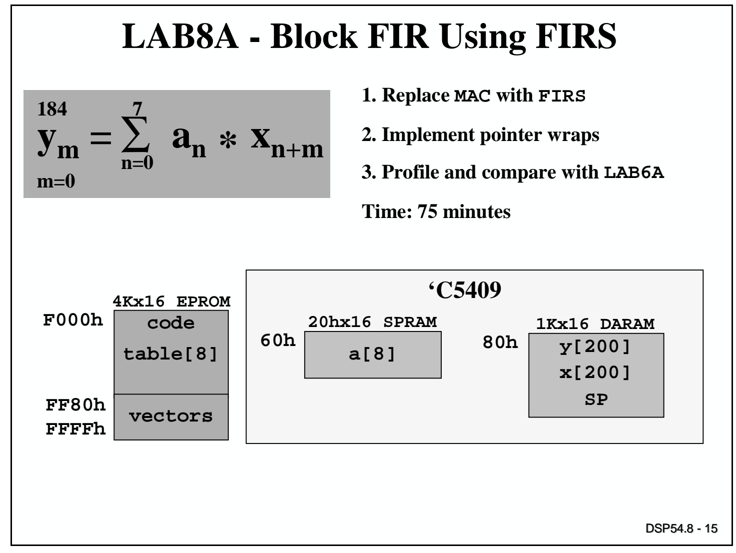
2. H. Hendrix, “Viterbi Decoding Techniques on the TMS320C54x Family”, TI Application Report, 1995.

DSP54.8 - 14

LAB8 – Block FIR

Objective

The objective of this lab is to write code to perform a block FIR using the FIRS instruction using symmetrical coefficients.



LAB8A - Procedure

Copy Files, Edit LAB8A.CMD

1. Make a new project called `LAB8A`.
2. Copy `LAB6A.CMD` to `LAB8A.CMD`. Modify as necessary.
3. Copy `LAB6A.ASM` to `LAB8A.ASM`.
4. Add the appropriate files to `LAB8A`.

Note: This lab will be much less structured and spoon-fed than previous labs – and rightly so. You should be more comfortable with the edit/debug environment, link.cmd files, program vs. data labels, etc., by now. The following instructions will head you in the right direction but will be slightly vague in terms of exact details.

Edit LAB8.ASM – Write/Debug

5. Open `LAB8A.ASM` for editing. Comment out the `CALL` to the `COPY` routine.
6. Replace the `MAC` instruction with `FIRS`. Don't forget that the label used to point to your coefficients is a program label. Also, the pointer wraps will be done manually, so there is no need to use circular addressing. Set up the two ARs to point to the correct data values (refer to the material if you need help).
7. This is a good chance to use the .asg directive for our pointers. Look up its use in the Assembly Language Tools User Guide.
8. Look at your RPT instruction. How does the single repeat count differ when using `FIRS` vs. `MAC`? Also, `FIRS` uses the `B` accumulator to accumulate the result *y*, so how will you initialize `B`?
9. `FIRS` requires the first two data values to be added together in the `A` accumulator prior to using the `FIRS` instruction. Write the proper instruction to add these values. Look up this instruction and note where the results will be. How should the pointers be modified?
10. Add two `MAR` instructions to wrap `AR2` and `AR3` the appropriate amounts after the `FIRS` instruction. Use the value in `AR0` to wrap `AR2` and use a constant `(#lk)` to wrap `AR3`.
11. Look at the store instruction. Which accumulator should be stored?
12. The `RPTB` instruction should remain as is because you still want to generate 185 outputs.

Build, Simulate, Verify

13. Build your project

14. Run the code and verify that results are being written to y. Debug as necessary. Once you think the code is running properly, graph your results.

Note: If, for some reason, you are not able to use symbolic debugging, check your assembler options to see if `-g` is in your switches.

---

15. When you have a clean graph of the filtered sine wave, profile the `FIRS` routine. Set the first profile-point on the `CALL` to your firs routine and the 2nd profile-point on the next instruction in the main routine. Write down your cycle count on the sheet provided. It should be around 3145 cycles. Was your cycle count less using `FIRS` than `MAC`? Why?

16. If you're done with `LAB8A` and you still have some time, move on to `LAB8B`…

LAB8-B Procedure

If you make any changes to `LAB8A.ASM` or `LAB8A.CMD`, first copy the files to `LAB8B.ASM` and `LAB8B.CMD`. Answers to the following questions are either contained in the on-line documentation or via your instructor.

1. Write the kernel to find the minimum value in the x[200] array. Rewrite it the kernel to find the maximum.
2. Edit the “max” kernel to also determine WHICH value was the maximum. In other words, you must find a way to determine the index (from the base x) that locates the max value.
3. Implement an asymmetric FIR using the FIRS instruction. There are several possibilities, but none that are simple.

Solutions

LAB8A.ASM - Solution

```
        .mmregs
        .def start

STKLEN  .set    100

a       .usect "coeffs",16,1
y       .usect "result",200
BOS     .usect "STK",STKLEN

        .sect "init"
table   .int 7FCh,7FDh,7FEh,7FFh
        .int 800h,801h,802h,803h
        .int 803h,802h,801h,800h
        .int 7FFH,7FEH,7FDH,7FCH

        .sect "indata"
x       .copy "in6.dat"

        .sect "code"
start:  STM     #BOS+STKLEN,SP   ;setup stack pointer
        STM     #0,SWWSR         ;set ext'l wait state to zero
        LD      #0,DP            ;set SST bit (saturate on store)
        ORM     #1,@PMST
        SSBX    FRCT             ;set FRCT bit (fractional mode)
        RSBX    OVM              ;clr OVM bit (overflow mode)
        SSBX    SXM              ;set SXM bit (sign extension)
```

DSP54.8 - 25

LAB8A.ASM - Solution (Continued)

```
;       CALL    copy
        CALL    fir
here:   B       here

copy:   STM    #a,AR1                     ;setup AR1
        RPT     #15                       ;copy 16 values
        MVPD    #table,*AR1+
        RET                               ;return

        .asg    AR2,TOP
        .asg    AR3,BOTTOM
        .asg    AR4,RESULTS

fir:    STM     #184,BRC
        STM     #x+15,BOTTOM              ;setup ARs for MAC
        STM     #x,TOP
        STM     #y,RESULTS
        STM     #-8,AR0

        RPTB    done-1
          ADD   *TOP+,*BOTTOM-,A          ;prime FIRS w/add of two data values
          RPTZ  B,#7                      ;execute FIRS 8 times (16 products)
            FIRS  *TOP+,*BOTTOM-,#table
          MAR   *TOP+0
          MAR   *+BOTTOM(#10)
          STH   B,*RESULTS+               ;store result

done:   RET                               ;return
```

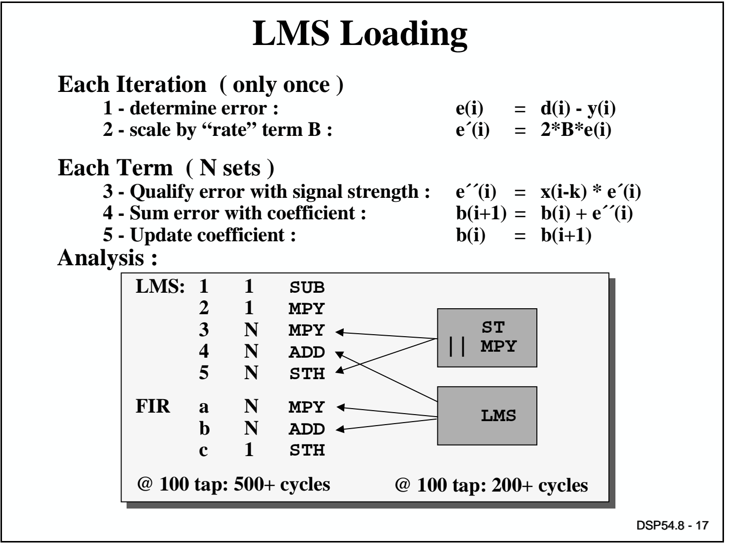
LAB8A.CMD : Solution

```
/* file I/O and options       */
vectors.obj
lab8a.obj
-m lab8a.map
-o lab8a.out
-e start

MEMORY {
PAGE 1:        /* Data memory */
  SPRAM:    org = 00060h,  len = 00020h
  DARAM:    org = 00080h,  len = 00400h

PAGE 0:        /* Program memory */
  EPROM:    org = 0F000h,  len = 00F80h
  VECS:     org = 0FF80h,  len = 00080h
}

SECTIONS
{ coeffs         :> SPRAM   PAGE 1
  result         :> DARAM   PAGE 1
  indata         :> DARAM   PAGE 1
  STK            :> DARAM   PAGE 1
  code           :> EPROM   PAGE 0
  init           :> EPROM   PAGE 0
  vectors        :> VECS    PAGE 0
}
```

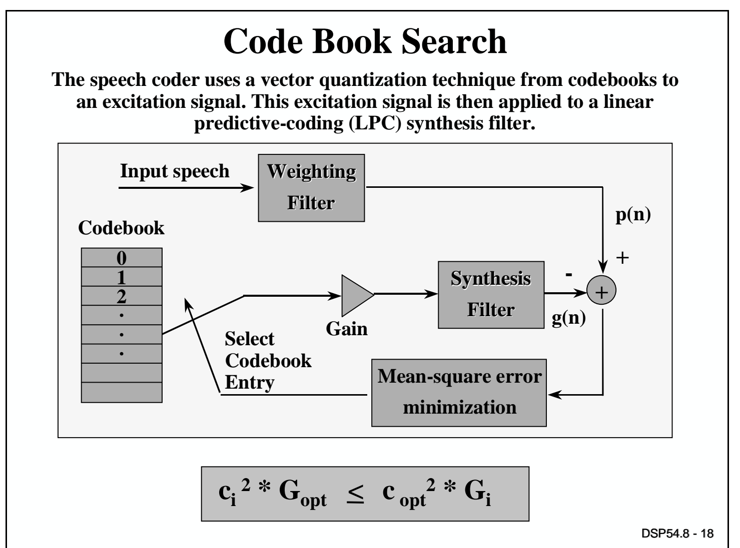
DSP54.8 - 27

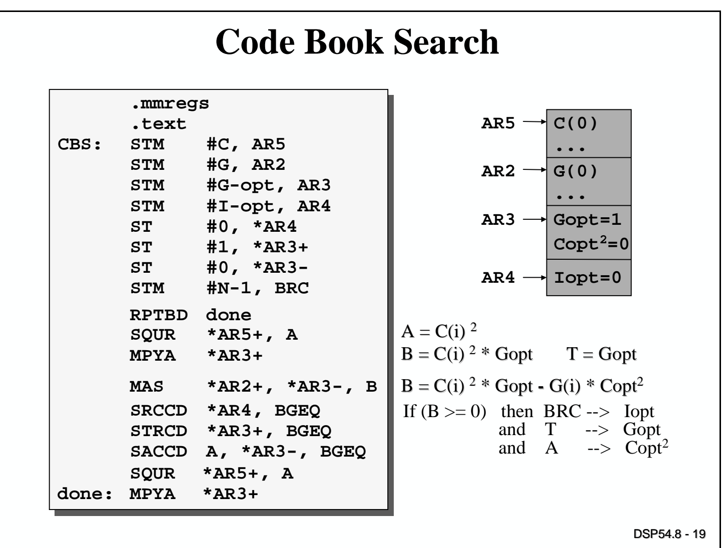
Additional Information

LMS Loading

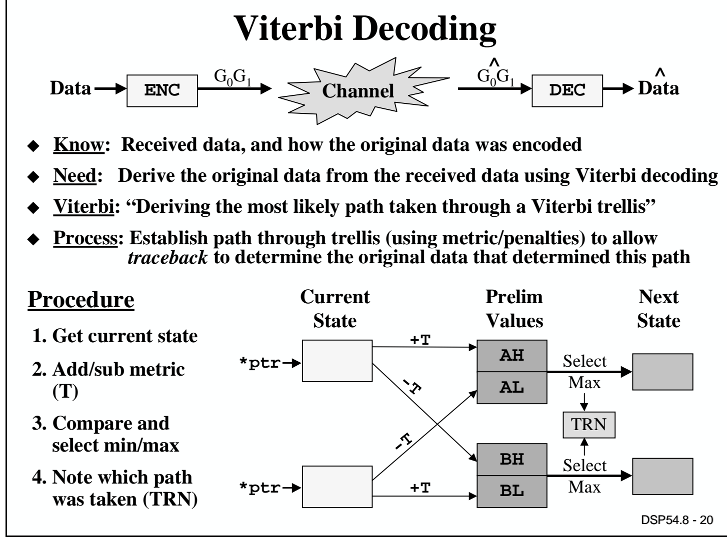


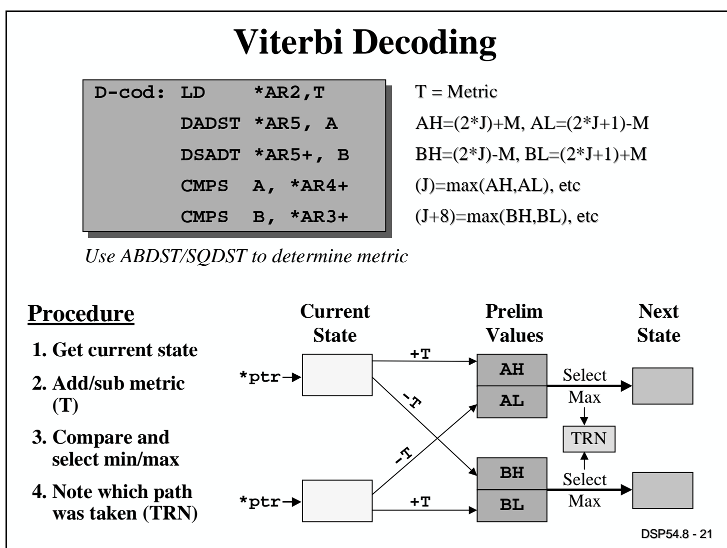
Codebook Search



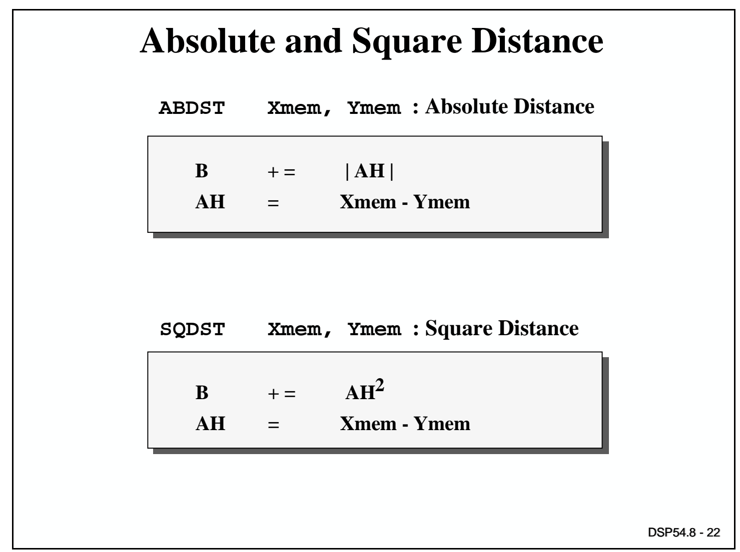


Viterbi Decoding





Determining Metrics



Polynomial Evaluation

Polynomial Evaluation

POLY is used to evaluate real algebraic polynomials of any order.

The general form of a 3rd order polynomial equation can be written as:

$$P(x) = a_3x^{3+} a_2x^2 + a_1x + a_0$$

The equation can be rewritten as:

$$P(x) = [(a_3x + a_2)x + a_1]x + a_0$$

This process can be extended to any order polynomial

DSP54.8 - 23

Polynomial Evaluation

```
   SSBX  FRCT
   SSBX  OVM
   SSBX  SXM

   LD    *AR4+,T
   LD    *AR3+,16,A
   LD    *AR3+,16,B

   RPT   #2
   POLY  *AR3+

   ST    A,*AR2+
|| LD  *AR4+,T
```

POLY operation is affected by these bits

T=X(0)
A=A(order)=PX init
B=A(order-1)

3 times
A=PX=Rnd(B+A*T) B=An<<16

PX=A>>16
T=new x

A parallel load may be added to do iterative POLY operations with no penalty.

Note: The POLY instruction “expects” Q15 numbers!

DSP54.8 - 24

Managing Interrupts

Introduction

Handling interrupts in a timely fashion is what makes a system real-time. You need to know what to set up, how interrupts are recognized and what to do when an interrupt is taken. It is not difficult, but you must complete each step in the process.

Learning Objectives

Objectives

- **Identify interrupt sources.**
- **Identify the requirements for interrupt recognition.**
- **Describe the sequence of events during an interrupt.**

DSP54.9 - 2

Module Topics

Managing Interrupts **9-1**

Module Topics *9-2*

Interrupt Timeline *9-3*

Interrupt Locations *9-4*

Creating VECTORS.ASM *9-5*

Interrupt Mask Register *9-6*

Global Interrupt Bit *9-7*

Interrupt Sources *9-8*

Interrupt Recognition *9-9*

Interrupt Flag Register *9-10*

Post Interrupt Hardware Sequence *9-11*
Context Saves and Restores 9-11
Return Instructions 9-12

Nesting Interrupts *9-13*

Relocating the Vector Table *9-14*

Software Interrupts *9-15*

Hardware State on Reset *9-16*

The Timer *9-17*

Review *9-18*

LAB9 – Managing Interrupts **9-19**

LAB9A - Procedure *9-20*
File Management 9-20
Edit `VECTORS.ASM,LAB9A.ASM` 9-20
Verify that Interrupts Work 9-21
Modify Block FIR Code 9-21
Verify Results 9-21

LAB9-B Procedure *9-22*

Solutions *9-23*

Interrupt Timeline

Interrupt Timeline

I N I T	1. Select interrupt sources 2. Create interrupt vector table 3. Enable individual interrupts 4. Enable Global Interrupts
C P U	5. Valid signal 6. Flag bit set 7. Int enabled? → branch to ISR
I S R	8. Context save/ISR/context restore 9. Return to main program

DSP54.9 - 3

Interrupt Locations

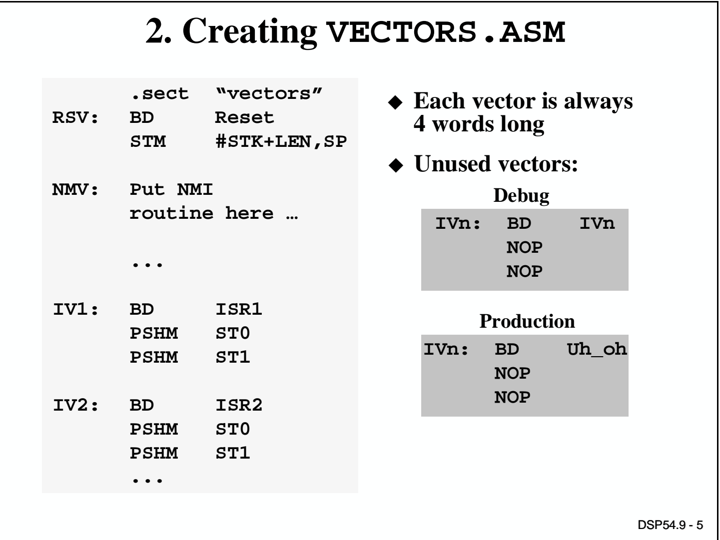
1. 'C5409 Interrupt Locations

Interrupt	Offset (Hex)	Priority	Description
RS	00	1	Reset
NMI	04	2	Nonmaskable Int
SINT17-30	08-3C	--	S/W Int 17-30
INT0	40	3	Ext'l Int #0
INT1	44	4	Ext'l Int #1
INT2	48	5	Ext'l Int #2
TINT	4C	6	Timer Int
BRINT0	50	7	McBSP #0 Rcv Int
BXINT0	54	8	McBSP #0 Xmt Int
BRINT2/DMAC0	58	9	McBSP #2 Rcv Int/DMA Ch0 Int
BXINT2/DMAC1	5C	10	McBSP #2 Xmt Int/DMA Ch1 Int
INT3	60	11	Ext'l Int #3
HINT	64	12	HPI Int
BRINT1/DMAC2	68	13	McBSP #1 Rcv Int/DMA Ch2 Int
BXINT1/DMAC3	6C	14	McBSP #1 Xmt Int/DMA Ch3 Int
DMAC4	70	15	DMA Ch4 Int
DMAC5	74	16	DMA Ch5 Int
Reserved	78-7F	--	Reserved

DSP54.9 - 4

Different devices may have different interrupt tables. Be sure to check your documentation.

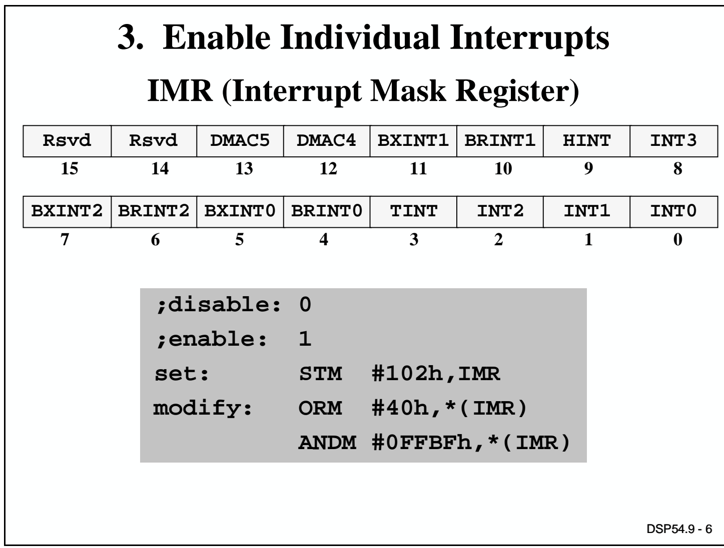
Creating VECTORS.ASM



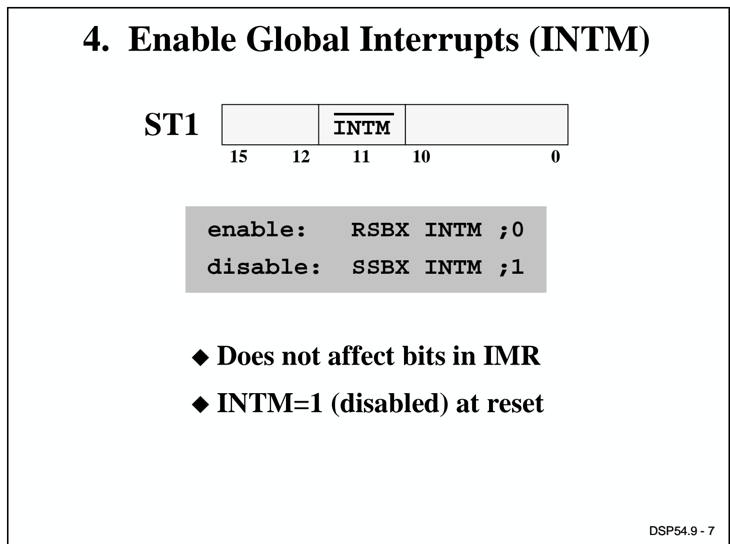
Remember that each location is at a specific address. You must make sure that you precisely locate each vector at the proper address.

Unused vectors can present interesting ways to cost yourself debugging time. If, for example, the solder joint that ties INT2 to a pull-up has a crack in it and the interrupt vector for INT2 contains NOPs ... one day that crack will open, INT2 might get taken and the CPU would execute the NOPs and succeeding code. This probably wouldn't be what you expected!

Interrupt Mask Register



Global Interrupt Bit



The IMR is “the big switch” for interrupts.

Interrupt Sources

5. 'C5409 External Interrupt Pins/Signals

'C5409

- $\overline{\text{INT0-3}}$ ←
- $\overline{\text{NMI}}$ ←
- $\overline{\text{RESET}}$ ←
- $\overline{\text{IACK}}$ →

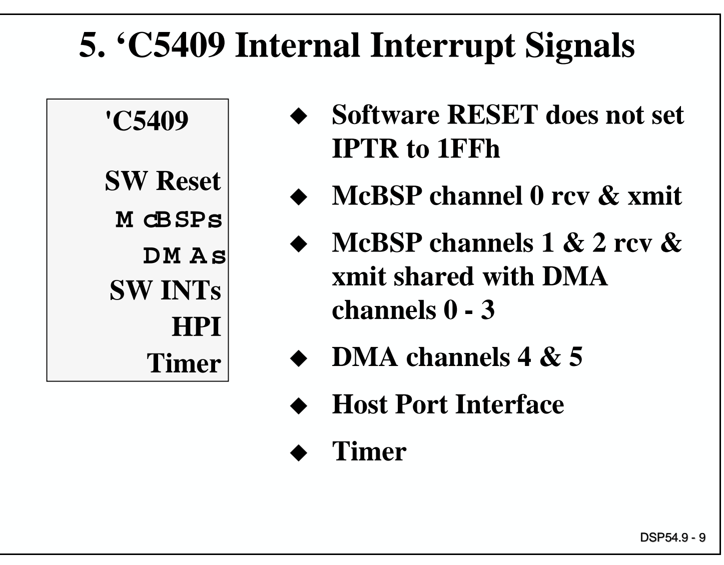
- **4 maskable external interrupts (INT0-3)**
- **2 non-maskable external interrupts (NMI, RESET)**
- **Interrupt acknowledge (IACK)**

Recommendation

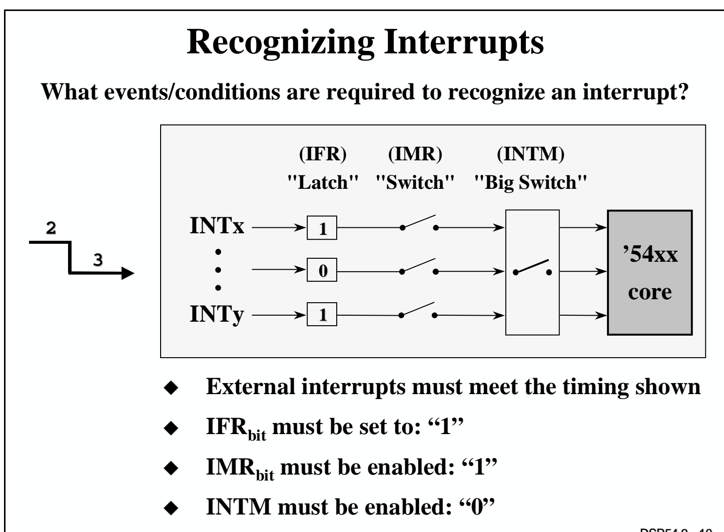
- **Do not use NMI as a high-priority interrupt. Because the state of INTM is not saved, returning to main code from an NMI could result in undesired behavior.**
- **Use NMI only when you do not intend to return to main code.**

DSP54.9 - 8

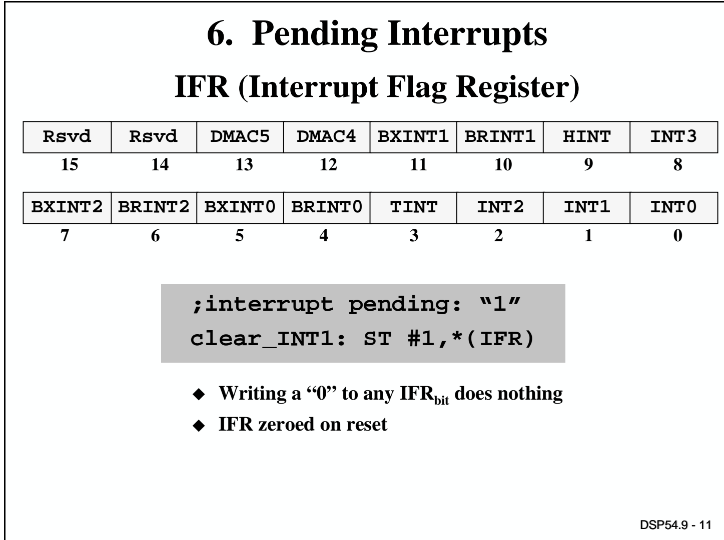
Since NMI doesn't save the state of INTM and it can interrupt main code, ISRs and itself, where to return to can become ambigous.



Interrupt Recognition



Interrupt Flag Register



Notice that the IMR and IFR are identical in layout. That makes coding a little easier.

Post Interrupt Hardware Sequence

7. Post Interrupt Hardware Sequence

CPU Action	Description
1 → $\overline{\text{INTM}}$	Disable global interrupts
PC → - - *(SP)	Push PC onto predecremented stack
Vector(n) → PC	Load PC with int. vector "n" address
0 → $\overline{\text{IACK}}$ pin	$\overline{\text{IACK}}$ signal goes low
0 → IFR (n)	Clear corresponding interrupt flag bit

Minimum interrupt latency is 7 cycles from a synchronous interrupt event to the fetch of the first ISR instruction. Add 2-3 cycles for an external interrupt.

DSP54.9 - 12

The CPU does these things automatically.

Context Saves and Restores

8. Context Save & Restore Instructions

Instruction	Description
PSHM mmr	Push MMR onto Stack SP - 1 → SP
POPM mmr	Pop from Stack to MMR SP + 1 → SP
PSHD Smem	Push Data memory value onto Stack SP - 1 → SP
POPD Smem	Pop top of Stack to Data memory SP + 1 → SP
FRAME K	Modify Stack Pointer SP + K → SP

Restore registers in the opposite order in which they were saved

DSP54.9 - 13

Then, since you have no idea when or where this routine may run, you should save all registers you touch, especially ST0 and 1 which contain important information about the processor state.

Return Instructions

9. Return Instructions

Instruction	Actions	Cycles
RET[D]	*(SP) ++ → PC	RET 5 RETD 3
RETE[D]	*(SP) ++ → PC 0 → INTM -	RETE 5 RETED 3
RETF[D]	RTN → PC 0 → INTM - *(SP) ++	RETF 3 RETFD 1

Using RETF[D]:

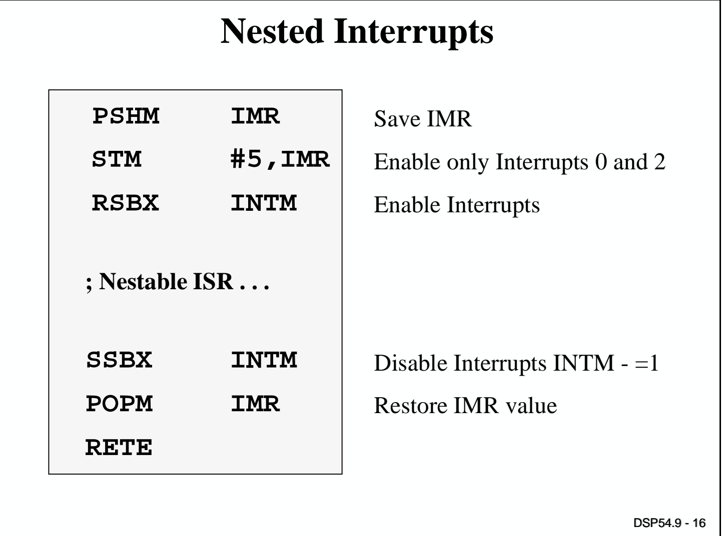
- 3-cycle ISR
- No calls, no nesting

```
RINT0: RETFD
       MVKD   DRR0,*AR7+%
       NOP
```

DSP54.9 - 14

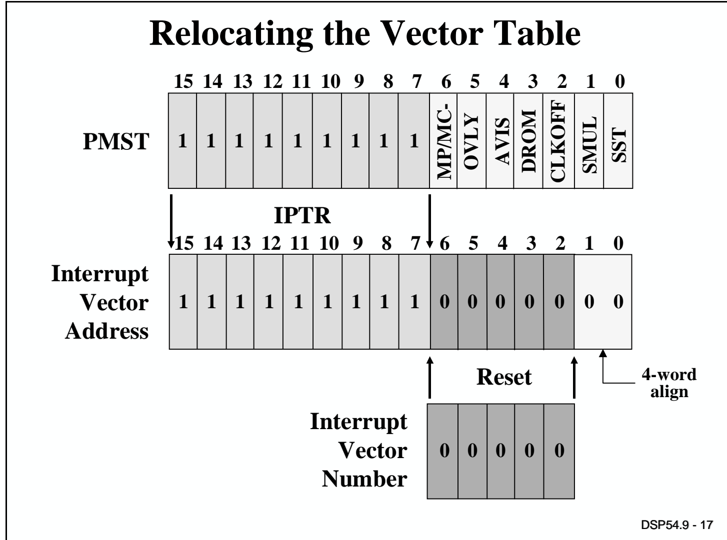
RETF depends on the RTN register. RTN will contain the last PC pushed to the stack, so you can’t do any CALLs or nest any ISRs and still use RETF.

Nesting Interrupts



In this example we are only allowing two interrupts to interrupt this ISR. These ISRs might need to be specially written with the knowledge that they are nested.

Relocating the Vector Table



This is especially useful for bootloads where the interrupt vector table is contained in factory programmed ROM. Chances are that it does not contain the code you'd like. Since you can't reprogram the ROM yourself (without $$'s) you'll need to program the table in available OVLY space and then point IPTR to this location.

Software Interrupts

Software Interrupts

```
INTR   k
TRAP   k
RESET
```

- k: interrupt number (see documentation)
- INTR = TRAP + disables INTM
- RESET instruction performs all tasks that a h/w reset does except it does not set IPTR to all 1's

DSP54.9 - 18

Operating systems make extensive use of software interrupts.

Hardware State on Reset

Hardware Reset

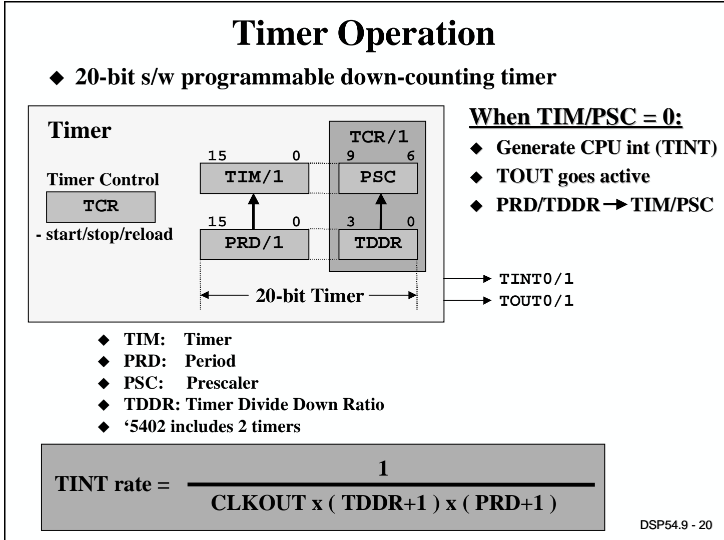
Math	Memory
0 → OVA/OVB/OVM	0 → OVLY,DROM
0 → C16,ASM,FRCT	? → MP/$\overline{\text{MC}}$
1 → SXM,C	1FFh → IPTR

Pins	Misc
1 → XF	0 → BRAF
0 → CLKOFF	0 → DP, CPL
0 → AVIS, HM	1 → INTM

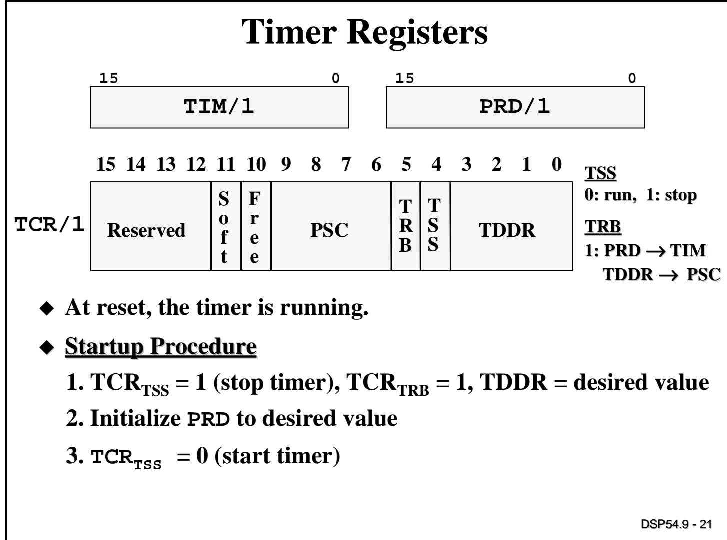
- A[x]: driven to FF80h, D[16]: high impedance
- An internal reset is sent to the peripherals.
- Seven CLKOUT cycles after RS- is released the processor will fetch from 0FF80h

DSP54.9 - 19

The Timer



The timer is running on Reset.



It is usually not important to program TIM since you have no idea "when" you're programming it.

Review

Review

1. What are the interrupt sources?

2. How do you poll for interrupts?

3. What must you set up to respond to an interrupt?

4. What conditions affect interrupt latency?

DSP54.9 - 22

LAB9 – Managing Interrupts

Objective

The objective of this lab is to modify your assembly routine to be an ISR. An infinite loop will be interrupted at specific intervals by the timer and the ISR will write one new result each time. The lab requires you to set up the proper registers to enable interrupts and generate a timer interrupt as well as create an interrupt vector for the timer ISR.

LAB9 - Managing Interrupts

1. **Convert your previous assembly routine to an ISR**
2. **Set up timer registers to generate an interrupt**
3. **Set up interrupt registers to respond to timer interrupt**
4. **Modify `VECTORS.ASM` to add a new vector**
5. **Modify current assembly code to output 1 new result each time the ISR is invoked**
6. **Graph results to verify**

Time: 60 minutes

DSP54.9 - 24

LAB9A - Procedure

File Management

1. Create a project called `LAB9A`.
2. Copy `LAB8A.ASM` to `LAB9A.ASM`.
3. Copy `LAB8A.CMD` to `LAB9A.CMD` and modify as necessary.
4. Add the appropriate files to your project. Double check your tool options and project options to make sure they are set as you like.

Edit `VECTORS.ASM,LAB9A.ASM`

5. Open `VECTOR9.ASM` and copy the contents to your clipboard. Open `VECTORS.ASM` for editing and paste the contents of the clipboard after your code. Delete the first few lines of the pasted information to make sure you have 4 AND ONLY 4 words in each vector. How many words is a "`B start`" instruction? Modify this file to invoke your block FIR routine as an ISR based on the occurrence of `TINT`. Make sure the label `fir` is visible to your program. Save your work. Close `VECTOR9.ASM`.
6. Open `LAB9A.ASM` for editing. Modify your code to make fir visible to `VECTORS.ASM`. Just before your call to the fir routine, write the necessary instructions to set up the timer with the following values:

   `TCR = 30h` (auto-reload TIM/PSC and stop timer)

   `PRD = 30h`

   `TCR = 20h` (start timer)

7. Below your timer setup code (prior to the call to fir), write the necessary instructions to respond to the timer interrupt (`TINT`). Also, write an instruction to clear any pending interrupts prior to turning on global interrupts. This is a good programming practice just in case a spurious interrupt occurred between reset and enabling global interrupts. If one did occur, as soon as you enabled INTM-, you'd service the interrupt.
8. Comment out or remove the `call` to *fir*.
9. Replace your infinite loop "here: B here" with the following instructions:

   ```
   main:    ADD #1,A
            ADD #1,B
            B   main
   ```

   This will be the endless loop that gets interrupted to run your *fir* ISR.

Verify that Interrupts Work

10. Build `LAB9A`. You have not modified the block FIR routine yet to actually give you correct results, but you must ensure that interrupts are working properly before taking this step.

11. Set a breakpoint on the label at the beginning of your fir routine. Then type `RUN` or click the `Run` button on the vertical toolbar to run your code. Does the simulator stop at your *fir* routine? If not, interrupts are NOT working. Debug, rebuild and re-simulate until the debugger stops at *fir*. There is no other way for the code to find your *fir* routine except via the interrupt. Once you have guaranteed that your interrupt setup code is working correctly, you can now modify the *fir* routine itself.

Modify Block FIR Code

12. Now that your interrupt setup code is working, you can make a few more modifications to the FIR code. We want the ISR to write one (1) result each time the interrupt occurs. Also, we want to re-enable interrupts when returning from the ISR.

13. Remove the breakpoint.

14. Delete or comment out the `RPTB` instruction and `RPTB` loop setup code.

15. Modify the `RET` instruction to use "return from an ISR".

16. Do not concern yourself with context save/restore yet. You might have noticed that the main routine that is interrupted is using accumulators A and B and you are likely overwriting these registers in the ISR. This is a no-no that we'll fix in `LAB9B`.

17. What else needs to change? Two issues are left: (1) how do you ensure that only 185 results will be written? Somehow, we need to turn off interrupts when 185 of them have been taken. (2) the setup code for *fir* will be run each time the interrupt occurs. This won't work because the pointers will be reset each time the ISR is invoked. Let's solve the 2nd problem first. The first issue has been left for `LAB9B` (if you make it that far…)

18. Change your *fir* label to point to the first math instruction (the first `ADD` if you're using `FIRS` or the first multiply if you're not using `FIRS`). Use another label at the top of the *fir* setup code and place a `RET` instruction at the end of this setup code. Just before your timer setup code, write a call to the fir setup code. Now, the setup code will only be executed once.

Verify Results

19. Build `LAB9A`. Set a breakpoint on the `STH` instruction at the end of your ISR (the store to *y*). Set your memory window to view the contents of memory starting at the address of *y*. Hit the `Run` button a few times and see the results being written to *y*. Does it look correct? If not, debug, rebuild and simulate.

20. When you think your code is working correctly, remove your breakpoint and reset the simulator. On the command line type: `RUN 4000`

21. This will generate ~185 outputs. Graph your results.

22. If you're done with `LAB9A` and you still have some time left, move on to `LAB9B`.

LAB9-B Procedure

1. Open `LAB9A.ASM` for editing.
2. Determine a way to STOP recognizing interrupts when exactly 185 results have been written to *y*. Run your code (NOT using `RUN 4000`) and verify it worked. Explain your method to the instructor.
3. Now, add the proper context save/restore code to your ISR. Any registers you use or modify in the fir code should be pushed to the stack and then popped just before you return. You might want to put the save of ST0 and ST1 inside the Timer Interrupt Vector. Don't forget to use `BD` and `.mmregs`.
4. How will you deal with the AR registers used in the math code that need to keep updating each time the ISR is executed? Write the code and verify your results. Explain your solution to the instructor.
5. If you've gotten this far, you're hot. Change your code to respond to `INT3-` instead of `TINT.`
6. If you've gotten here, you're REALLY hot. Write your ISR in C and verify your results.

Solutions

Review

1. What are the interrupt sources?

Reset, NMI, Timers, Serial Ports, DMA, External, Software

2. How do you poll for interrupts?

Test the appropriate bit in IFR, then branch to ISR if TC set

3. What must you set up to respond to an interrupt?

INTM-, IMR, SP and a vector

4. What conditions affect interrupt latency?

Higher priority interrupts, IMR_{bit}=0, processor is in hold mode, INTM = 1, memory speed, Not READY, ...

DSP54.9 - 23

LAB9A.ASM - Solution

```
        .mmregs
        .def start,fir

STKLEN  .set    100

a       .usect "coeffs",16,1
y       .usect "result",200
BOS     .usect "STK",STKLEN

        .sect "init"
table   .int 7FCh,7FDh,7FEh,7FFh
        .int 800h,801h,802h,803h
        .int 803h,802h,801h,800h
        .int 7FFH,7FEH,7FDH,7FCH

        .sect "indata"
x       .copy "in6.dat"

        .sect "code"
start:  STM     #BOS+STKLEN,SP   ;setup stack pointer
        STM     #0,SWWSR         ;set ext'l wait state to zero
        LD      #0,DP            ;set SST bit (saturate on store)
        ORM     #1,@PMST
        SSBX    FRCT             ;set FRCT bit (fractional mode)
        RSBX    OVM              ;clr OVM bit (overflow mode)
        SSBX    SXM              ;set SXM bit (sign extension)
```

DSP54.9 - 26

LAB9A.ASM - Solution (continued)

```
        CALL    fir_setup        ;setup pointers for fir ISR

        STM     #30h,TCR         ;auto reload TIM/PSC, stop timer
        STM     #30h,PRD         ;init period to 30 cycles
        STM     #20h,TCR         ;start timer

        STM     #0FFFFh,IFR      ;clear any pending interrupts
        STM     #8,IMR           ;enable TINT bit in IMR
        RSBX    INTM             ;turn ON global interrupts

main:   ADD     #1,A
        ADD     #1,B
        B       main

        .asg    AR2,TOP
        .asg    AR3,BOTTOM
        .asg    AR4,RESULTS

fir_setup:
        STM     #x+15,BOTTOM     ;setup ARs for MAC
        STM     #x,TOP
        STM     #y,RESULTS
        STM     #-8,AR0
        RET
```

DSP54.9 - 27

LAB9A.ASM - Solution (continued)

```
fir:    PSHM    AL                         ;context save
        PSHM    AH
        PSHM    AG
        PSHM    BL
        PSHM    BH
        PSHM    BG

        ADD     *TOP+,*BOTTOM-,A           ;prime FIRS w/add of two data values
        RPTZ    B,#7                       ;execute FIRS 8 times (16 products)
          FIRS  *TOP+,*BOTTOM-,#table
        MAR     *TOP+0
        MAR     *+BOTTOM(#10)
        STH     B,*RESULTS+                ;store result

        POPM    BG                         ;context restore
        POPM    BH
        POPM    BL
        POPM    AG
        POPM    AH
        POPM    AL
        POPM    ST1                        ;pushed in vectors.asm
        POPM    ST0

done:   RETE                               ;return from interrupt
```

DSP54.9 - 28

VECTORS.ASM - Solution

```
        .def    rsv
        .ref    start,fir

        .sect   "vectors"

rsv:    B       start
        RETE
        RETE

        RETE                    ;Non-maskable Interrupt Vector
        RETE
        RETE
        RETE

        . . .

        RETE                    ;Software Interrupt 18 Vector
        RETE
        RETE
        RETE

        BD      fir             ;Timer Interrupt Vector
        PSHM    ST0
        PSHM    ST1
```

DSP54.9 - 29

Setting Up and Using Peripherals

Introduction

Advanced C54x devices have some combination of the following three peripherals on them; the DMA, the EHPI and the McBSP. In this module we'll step through the capabilities of each and delve into getting them set up for use.

Learning Objectives

Objectives

- **Analyze how the DMA operates**
- **Understand the basic setup required to use the DMA to perform a task**
- **Describe the DMA's additional capabilities**

DSP54.10 - 2

Module Topics

Setting Up and Using Peripherals 10-1

Module Topics 10-2

The DMA 10-3
Registers 10-4
Throughput 10-5
Example 10-6
Other DMA Issues 10-6

The McBSP 10-7
Capabilities 10-8
Example 10-8
Sample Rate Generator 10-9
Multi-Channels 10-9
Example 10-10
Other McBSP Capabilities 10-10

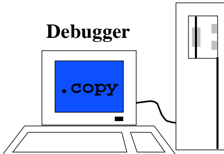
The EHPI 10-11
EHPI Operation 10-12
Other EHPI Issues 10-12

Some Additional Information 10-14
Setting Up a DMA Transfer 10-14

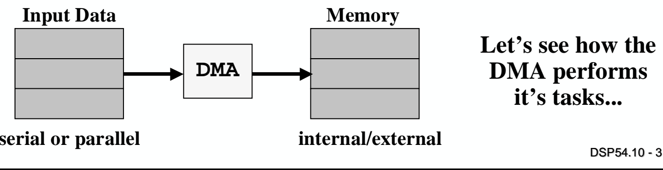
The DMA

DMA - Intro

- We used the ".copy" directive to load the input samples into simulated internal memory.



- In a real system, the DMA would perform this operation via EMIF or a serial port using it's own buses:

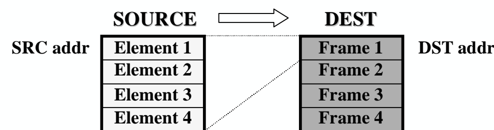


Let's see how the DMA performs it's tasks...

DSP54.10 - 3

Direct Memory Access (DMA)

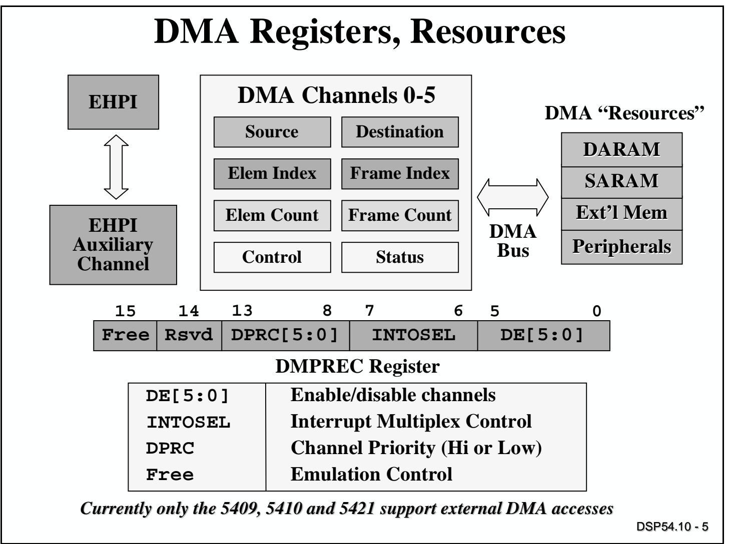
- Performs data transfers without CPU intervention

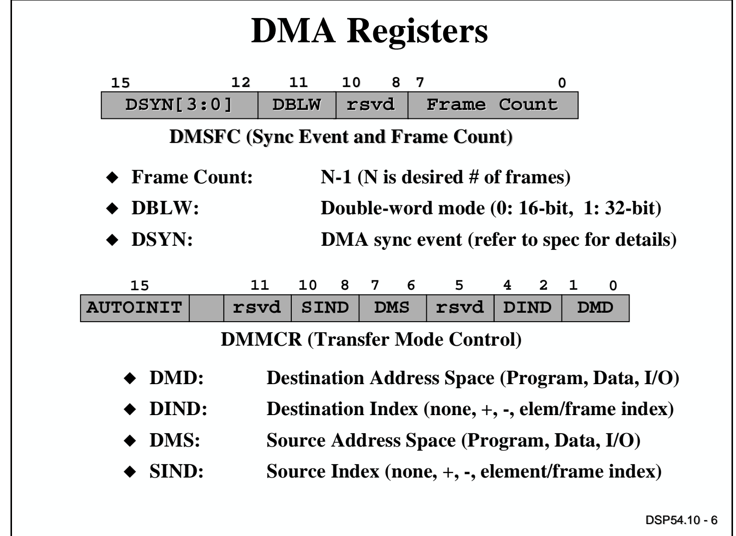


- <u>Terminology</u>
  - Element: basic unit of transfer (1, 2 words)
  - Frame: multiple elements (1-64K)
  - Block: multiple frames (1-256)
- <u>Max Speed:</u> One 16-bit word per 4 CPU cycles (all channels combined)
- <u>Transfer dependent upon:</u>
  - Source/destination address
  - Rotating priority between channels
  - Event sync (different events can be selected)
  - Element/Frame count
  - Index (can select: no mod, inc/dec by 1, element/frame index)

DSP54.10 - 4

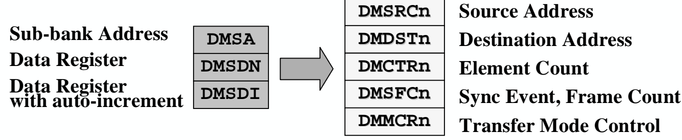
Registers





Writing to the DMA Registers

- Writing to DMA registers is a multi-step process using sub-bank addressing



- Using sub-bank addressing with auto-increment

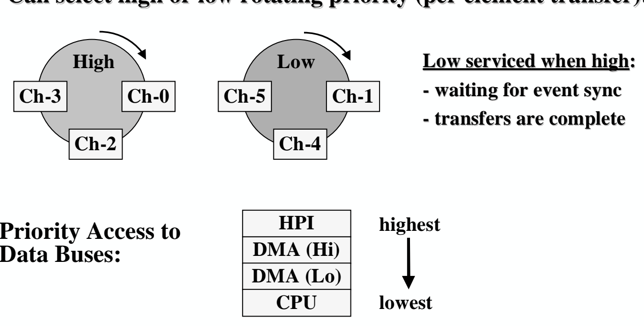
```
DMSRC0 .set 00h
       STM  DMSRC0,DMSA   ;init DMSA to pt to DMSRC0
       STM  #1000h,DMSDI  ;write 1000h to DMSRC0
       STM  #2000h,DMSDI  ;write 2000h to DMDST0
      …etc.
```

DSP54.10 - 7

Throughput

DMA Throughput

- DPRC[5:0]: determines DMA bus priority between channels
  - (hi-1, low-0) Affects access to the 16-bit DMA bus only
  - Can select high or low rotating priority (per element transfer):

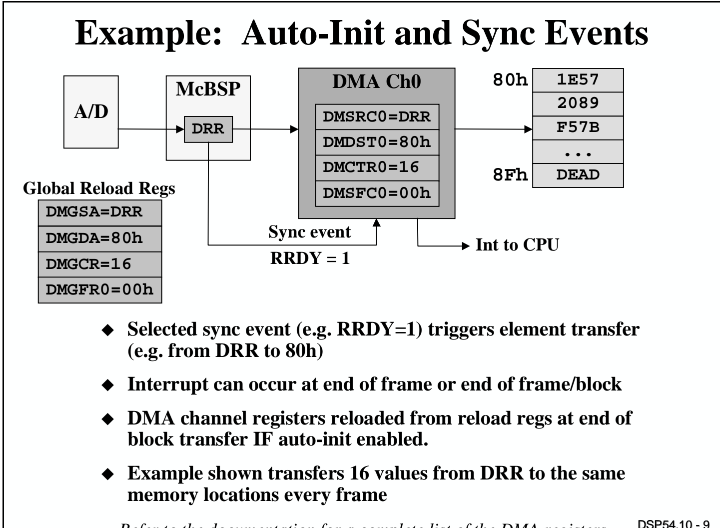


Low serviced when high:
- waiting for event sync
- transfers are complete

- Priority Access to Data Buses:

DSP54.10 - 8

Example



Other DMA Issues

Other DMA Issues

- **Auto-Initialization:**
  - At end of block, DMA copies reload registers to channel registers
  - Only ONE set of re-load registers exist for all 6 channels
  - Reload registers can be altered at any time
  - Every channel can select to use auto-init and reload registers
  - Allows user to provide "continuous" or "repititious" operation
- **Interrupt to CPU** (one interrupt per channel)
  - Triggered at: a) end of frame; b) end of frame and end of block
  - Duplicates C54x ABU when selected (half/full buffer or full buffer)
- **Synchronization Events**
  - DMA ELEMENT transfer is triggered by a specific sync event
  - Example: DRR ready to read tells DMA to ready next value
  - Up to 16 sync events - see datasheet for details

Refer to the documentation for a complete list of the DMA registers

DSP54.10 - 10

The McBSP

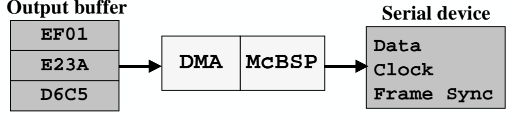
Objectives

- Analyze how the McBSP operates
- Understand the basic setup required to use the McBSP to perform a task
- Describe the McBSP's additional capabilities

DSP54.10 - 11

McBSP - Intro

- In the labs, we graphed the filter output buffer using CCS
- In a real system the output buffer could be transferred to a serial device using the McBSP and DMA:

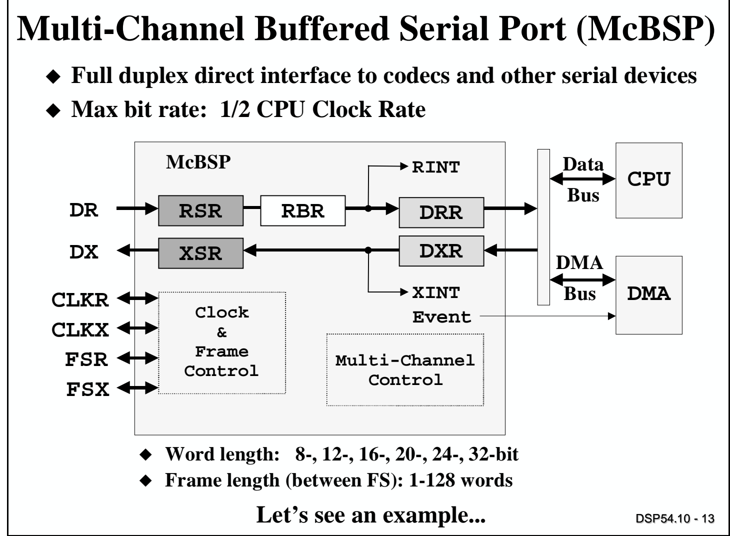


The McBSP isn't buffered without the DMA

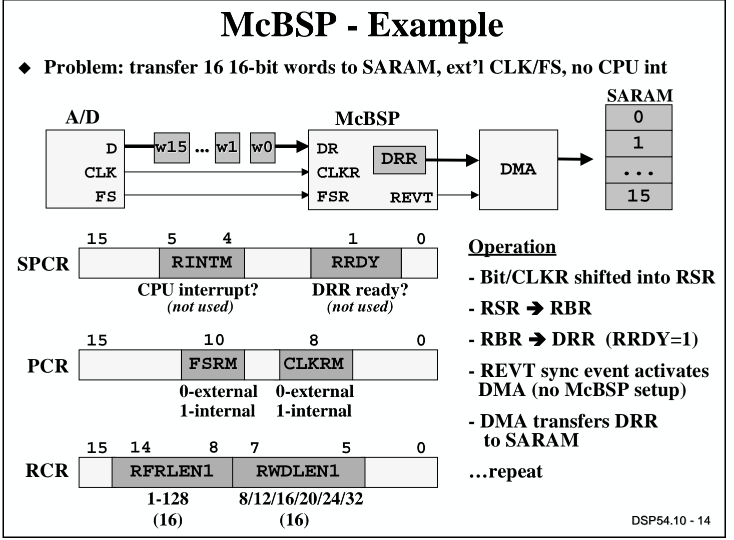
Let's see how the McBSP performs it's tasks...

DSP54.10 - 12

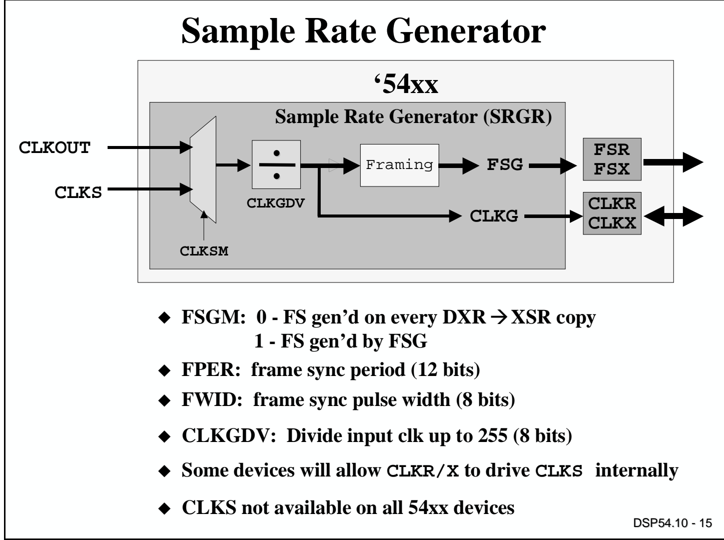
Capabilities



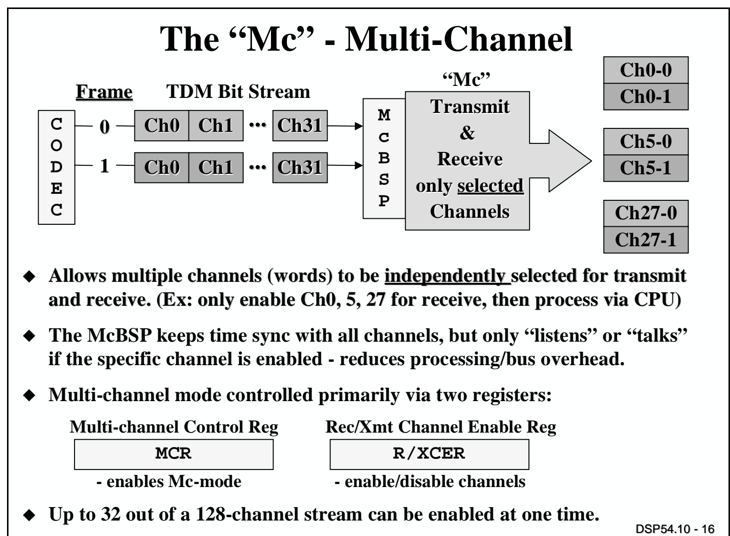
Example



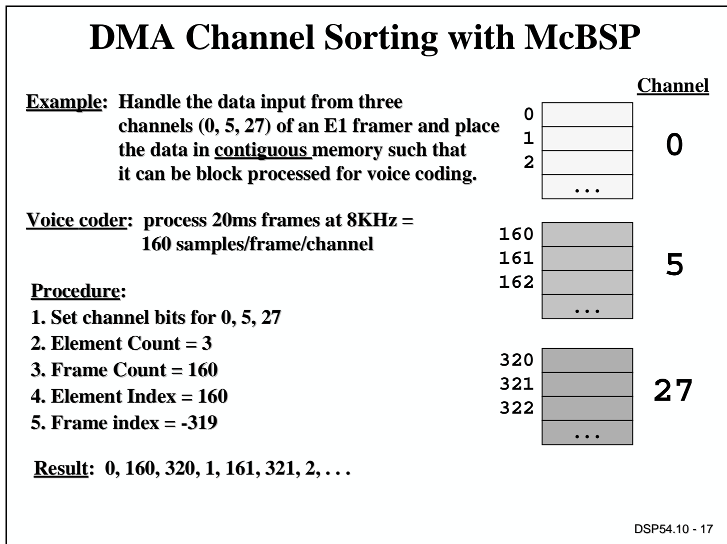
Sample Rate Generator



Multi-Channels



Example



Other McBSP Capabilities

Other McBSP Capabilities

- **Supports direct interface to: T1/E1 framers, MVIP/ST-Bus, IOM-2, AC'97 (multi-phase), IIS, SPI**
- **µ-law/A-law companding in hardware (on receive and transmit)**
- **Internal clock/frame generation using the Sample Rate Generator (SRGR)**
- **Programmable polarity of clock/frame**
- **Digital Loopback (DLB): internally hooks xmt to rcv for debug**
- **CPU interrupts occur when: (R/XRDY=1, end of block/frame, new FS, error)**
- **Can delay first data bit after FS by 0, 1, 2 clk cycles (required by some algorithms, only on transmit)**
- **All typical errors/status are reported**
- **McBSP pins can be configured as general-purpose I/O if desired**

All pins, registers and full explanations of these capabilities can be found in the Peripheral User's Guide

DSP54.10 - 18

The EHPI

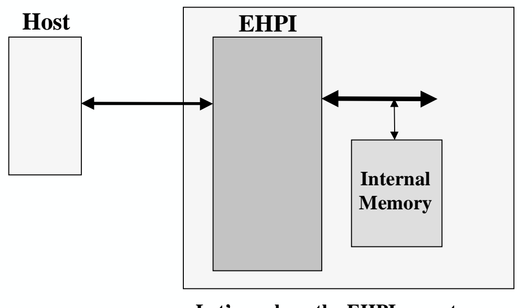
Objectives

- Describe operation of EHPI
- Understand the basic setup required to use the EHPI to perform a task
- Describe additional capabilities

DSP54.10 - 19

Enhanced Host Port Interface (EHPI)

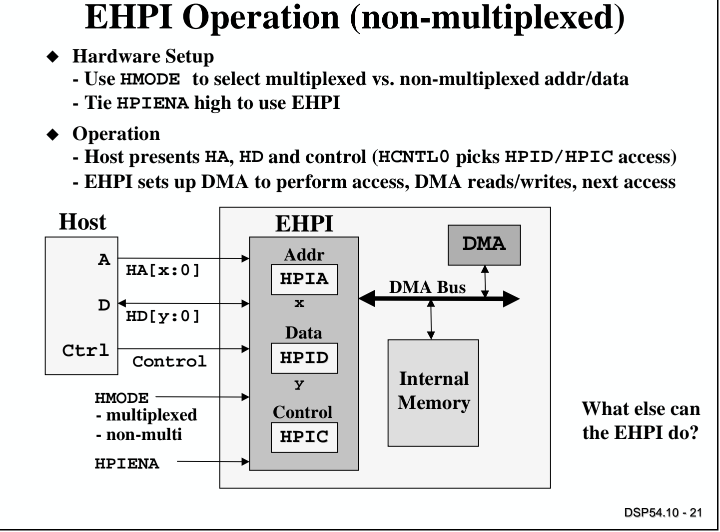
- ACCESS: 8/16-bit port access to 54x's on-chip memory resources
- BOOT: Can boot load DSP's internal memory during reset
- SPEED: 33MBytes/sec @ 100MHz (6 cycles/word, max)
- MODES: Multiplexed Address/Data, Non-multiplexed (A, D separate)



Let's see how the EHPI operates...

DSP54.10 - 20

EHPI Operation

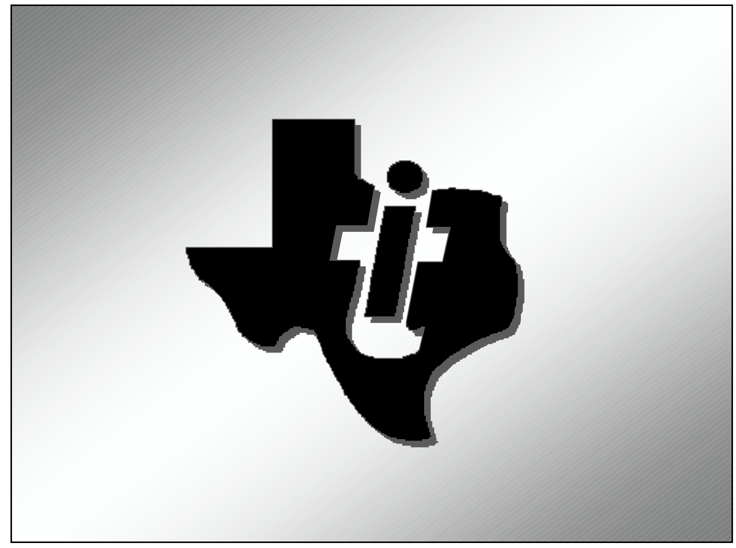


Other EHPI Issues

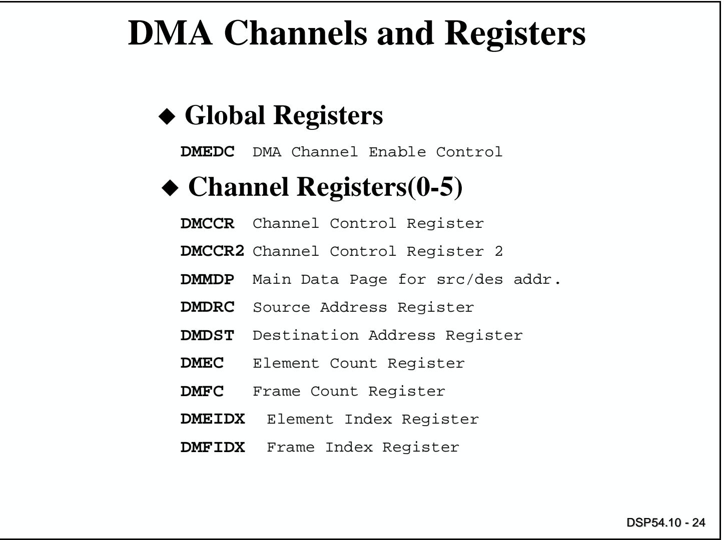
Other EHPI Issues

- **EHPI Bootload during Reset**
  - hold device in reset, transfer data, release reset
  - Host has full access to internal memory
- **Interrupts**
  - DSP to Host
  - Host to DSP
- **Multiplexed Address/Data lines (Intel access)**
  - Uses HD[15/7:0] only, shared address/data bus
- EHPI always has priority to DMA bus

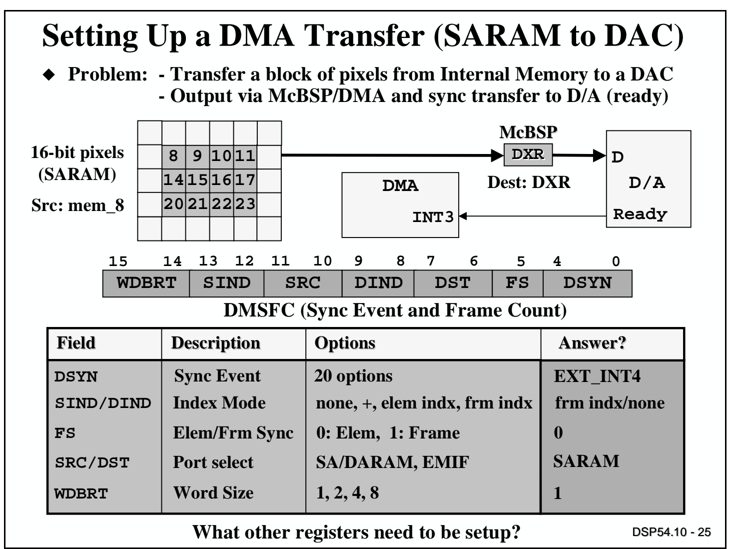
DSP54.10 - 22



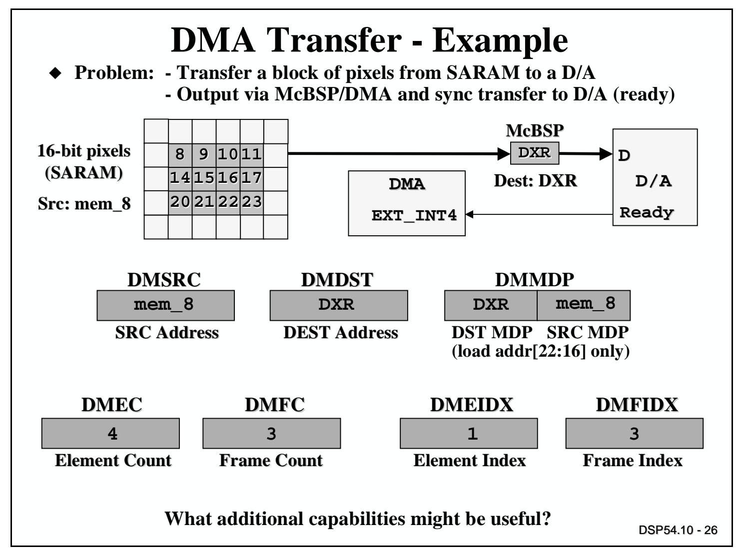
Some Additional Information



Setting Up a DMA Transfer



Field	Description	Options	Answer?
DSYN	Sync Event	20 options	EXT_INT4
SIND/DIND	Index Mode	none, +, elem indx, frm indx	frm indx/none
FS	Elem/Frm Sync	0: Elem, 1: Frame	0
SRC/DST	Port select	SA/DARAM, EMIF	SARAM
WDBRT	Word Size	1, 2, 4, 8	1


DMA Transfer - Example
◆ Problem: - Transfer a block of pixels from SARAM to a D/A
- Output via McBSP/DMA and sync transfer to D/A (ready)
16-bit pixels
(SARAM)
Src: mem_8
8 9 10 11
14 15 16 17
20 21 22 23
McBSP
DXR
Dest: DXR
D
D/A
Ready
DMA
EXT_INT4
DMSRC
mem_8
SRC Address
DMDST
DXR
DEST Address
DMMDP
DXR
mem_8
DST MDP
SRC MDP
(load addr[22:16] only)
DMEC
4
Element Count
DMFC
3
Frame Count
DMEIDX
1
Element Index
DMFIDX
3
Frame Index
What additional capabilities might be useful?
DSP54.10 - 26

Mixing C and Assembly

Introduction

The most important aspect of using C in a Digital Signal Processor is mixing assembly into the C runtime environment Parameter passing and register usage will be covered as well.

Learning Objectives

Objectives

- **Understand the C Environment**
- **Run the Compiler**
- **Describe how to Mix C and Assembly**

DSP54.11 - 2

Module Topics

Mixing C and Assembly **11-1**

Module Topics *11-2*

The C Run-time Environment *11-3*
C Linker Command File 11-3
Compiling and Linking 11-4
The C Environment 11-4
Status Register Expectations 11-5
Func.ASM 11-5

Passing Parameters *11-6*
Accessing MMRS 11-7
Interrupts 11-7
Numerical Types 11-9
C Optimization Levels 11-9
Other C Stuff 11-10

LAB11 – Mixing C and Assembly **11-11**

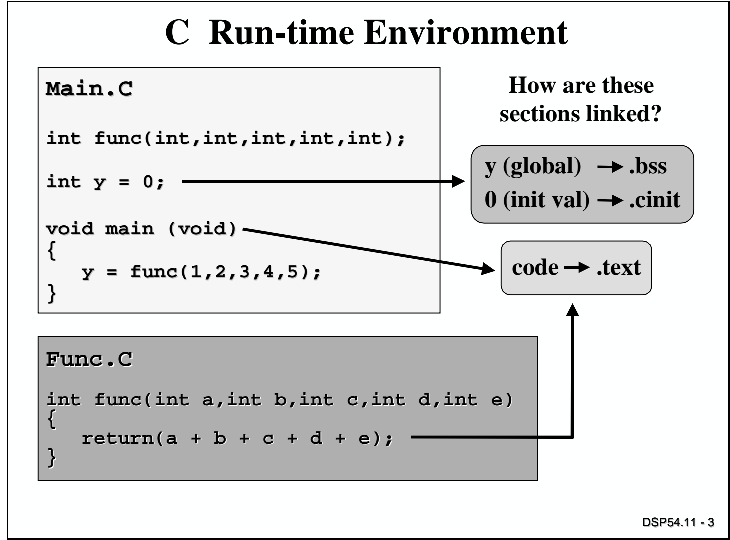
Objective 11-11

LAB11A - Procedure *11-12*
Edit `LAB11A.ASM` 11-12
Build and Simulate 11-13

LAB11B – Procedure *11-15*

Solutions *11-16*

The C Run-time Environment



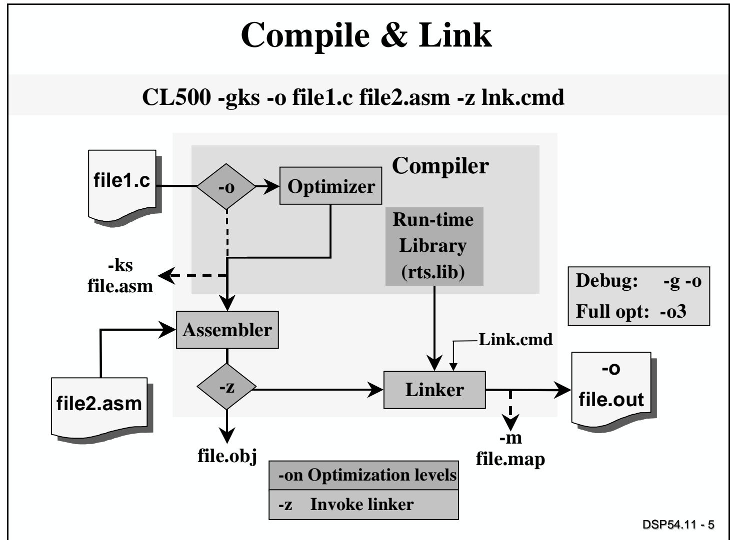
C Linker Command File

C Linker Command File

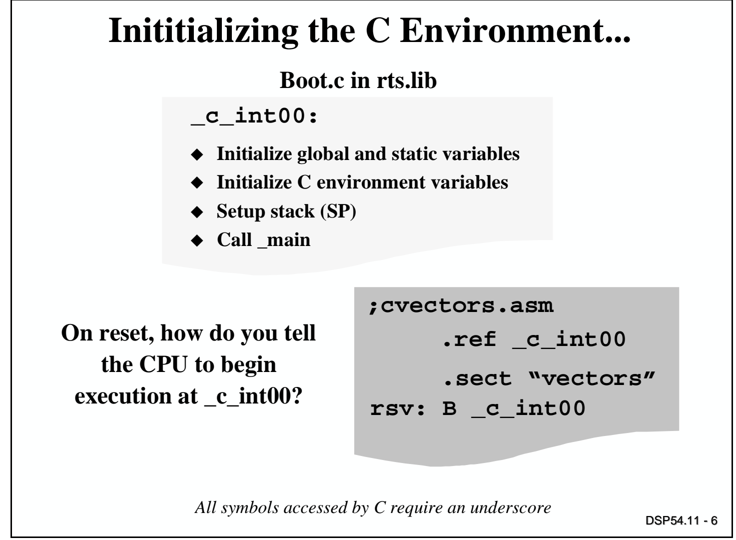
```
MEMORY
{PAGE 0:
  VECS:  org = 0xFF80,    len = 00080h
  EPROM: org = 0xF000,    len = 00F80h
 PAGE 1:
  DARAM: org = 0x0080,    len = 04000h
  CROM:  org = 0x8000,    len = 01000h }
SECTIONS{
.text:     >   EPROM PAGE 0    /* code               */
.cinit:    >   EPROM PAGE 0    /* global inits       */
.bss:      >   DARAM PAGE 1    /* variables          */
.stack:    >   DARAM PAGE 1    /* for SP             */
vectors    >   VECS  PAGE 0    /* vectors            */
.const     >   CROM  PAGE 1    /* const int x=25;    */
.switch    >   EPROM PAGE 0    /* for case stmts     */
.sysmem    >   DARAM PAGE 1    /* heap, dynamic mem  */
}
```

DSP54.11 - 4

Compiling and Linking



The C Environment



Status Register Expectations

Run-time Environment

STx Bits	Name	Presumed Value	Modified
ARP	Auxiliary Reg Ptr	0	Yes
ASM	ACC shift mode		Yes
BRAF	Block Rpt Active Flag		No
C	Carry bit		Yes
C16	Dual 16-bit math	0	No
CMPT	Compatibility mode	0	No
CPL	Compiler mode	1	No
FRCT	Fractional mode	0	No
OVA/B	ACC Overflow flags		Yes
OVM	Overflow mode	0	*
SXM	Sign-extension mode		Yes
SMUL	Saturate/multiply		*
TC	Test Control flag		Yes

- If the user modifies a "presumed value", this value must be restored by the function
- * - for intrinsics only

DSP54.11 - 7

Func.ASM

Writing Func.ASM

Main.C

```
int func(int,int,int);

int y = 0;

void main (void)
{
   y = func(1,2,3);
}
```

Main.C:

- prototypes called function
- calls function

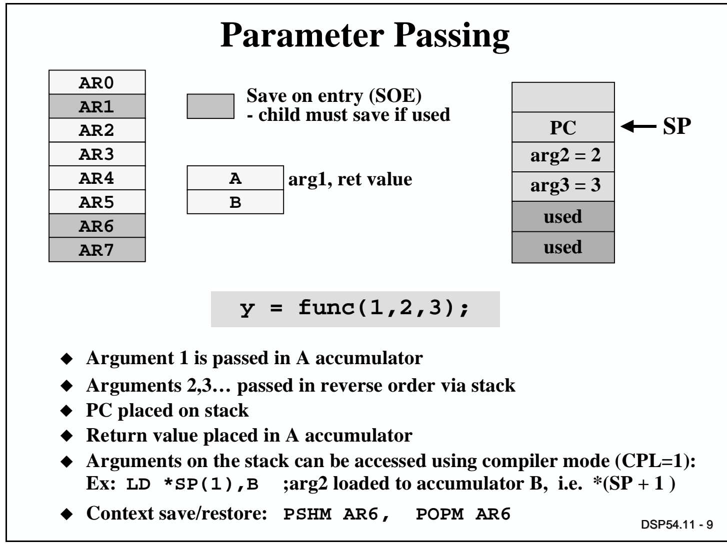
How are the parameters passed to func( ) ?

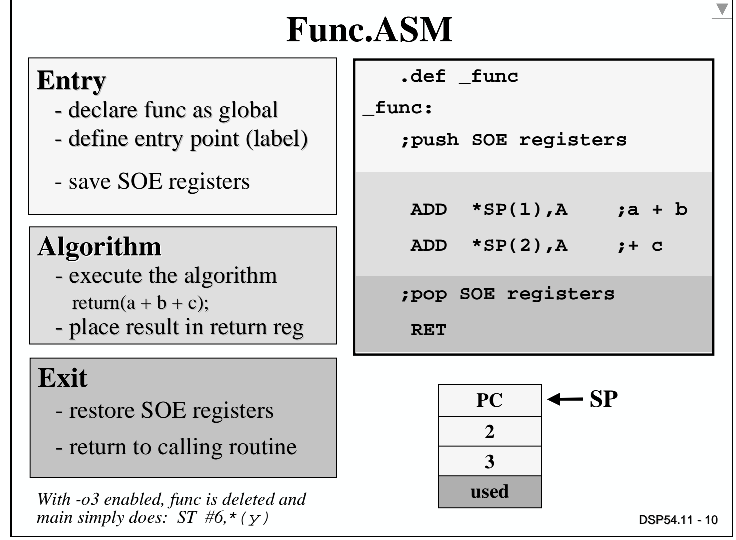
Func.C

```
int func(int a,int b,int c)
{
   return(a + b + c);
}
```

DSP54.11 - 8

Passing Parameters





Accessing MMRS

Accessing MMRs from C

- Using pointers to access Memory-Mapped Registers :
  - Declare the necessary MMR component :

```
extern volatile unsigned SWWSR;
```

  - Set it's address via a forced ASM statement :

```
asm("SWWSR  .set   0x28");
```

  - Read and write to the register as desired :

```
SWWSR = 0x8244;
```

- Volatile modifier :
  - Especially important with optimizer (-o)
  - Tells compiler to always recheck actual memory whenever encountered
  - Otherwise, optimizer might register-base value, or eliminate construct

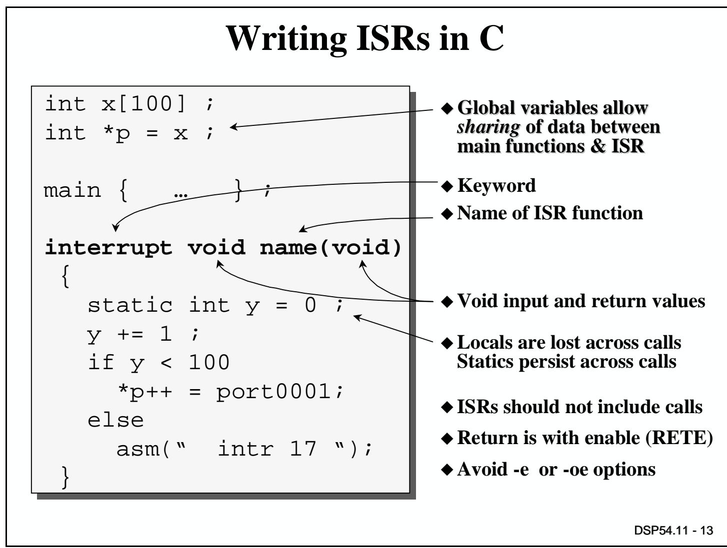
DSP54.11 - 11

Interrupts

Interrupts in C

- Interrupt Service Routine
  - C function to run when interrupt occurs
  - All necessary context save/restore performed automatically
- Interrupt Initialization Code
  - Should be called prior to run-time process
  - Interrupt status may be modified during run-time
- Interrupt Vector Table
  - Written in ASM

DSP54.11 - 12



Initializing Interrupts in C

Setup pointers to IMR & IFR. Initialize IMR, IFR, INTM :

```
volatile unsigned int *IMR = (volatile unsigned int *) 0x0000;
volatile unsigned int *IFR = (volatile unsigned int *) 0x0001;

*IFR = 0xFFFF;
*IMR = 0xFFFF;

asm("       RSBX  INTM ");
```

Create Vector Table :

```
.sect     ".vectors"
…
B         _ISR1
nop
nop
…
```

Compiled ISR Sequence :

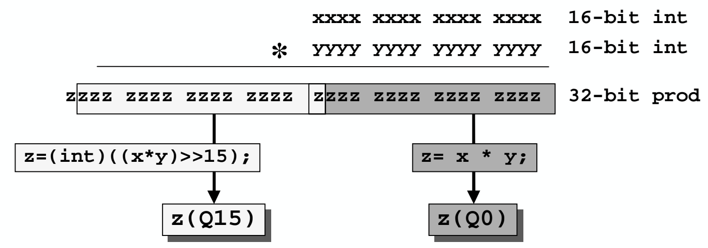
- I$$SAVE performs context save (from RTS.LIB)
- ISR function runs
- I$$RESTORE performs context restore (RTS.LIB)
- RETE - Return with Enable

DSP54.11 - 14

Numerical Types

Numerical Types in C

- Q15 math in C is accomplished by shifting the result:



- an integer is defined as the low portion of the accumulator
- `short`, `char`, etc, occupy full 16-bits of memory
- `float` operations supported via rts.lib (multicycle)

DSP54.11 - 15

C Optimization Levels

C Optimization Levels

Level 0	- allocates variables to registers - simplifies expressions - eliminates unused code
Level 1 "0" + ...	- removes unused assignments and common expressions - single function (local) optimizations
Level 2 "1" + ...	- performs loop optimizations/unrolling - multi-function (global) optimizations
Level 3 "2" + ...	- removes unused functions - in-lines calls to small functions - can perform multi-file optimizations using project mode (assertions) - other options available with Level 3

optimization levels are set via CCS build options

DSP54.11 - 16

Other C Stuff

Other C Stuff...

```
asm(" IDLE");

#include <intrindefs.h>
y = _smacr(x1, x2, x3);

#pragma Data_Section(y,"Var");
int y = 0;

volatile unsigned int *ctrl;
while (*ctrl != 0xFF);
```

- **In-Line Assembly**
  - can disrupt C
- **Intrinsics**
  - ASM instructions in C
  - see C Compiler guide
- **Data/Program Sections**
- **Volatile Keyword**
  - compiler may remove code without volatile keyword

- **CCS allows you to change the default stack and heap sizes in Project : Options : Compiler if desired.**

DSP54.11 - 17

LAB11 – Mixing C and Assembly

Objective

In this lab we'll be changing one of out previous assembly language files to be C callable. Pay careful attention to the passing of parameters in the A accumulator and on the stack

LAB11A - Mixing C and ASM

1. **Review the given file: `MAIN11A.C`**
2. **Modify block FIR routine to be C callable**
3. **Review/modify given linker command file**
4. **Build, profile and verify operations**

Time: 75 minutes

DSP54.11 - 18

LAB11A - Procedure

1. Create a new project called `LAB11A`.

2. Copy `LAB6A.ASM` to `LAB11A.ASM` and add it to the project.

3. Add and inspect the given files below to the project:

   - `LAB11A.CMD`
   - `MAIN11A.C`
   - `CVECTORS.ASM`

4. `IN11.H` is included in the C routine and does not need to be added to the project

Edit `LAB11A.ASM`

4. Make the following changes to `LAB11A.ASM`:

   - Remove the allocations for *a, x, y* and the stack. `MAIN11A.C` and `BOOT.ASM` take care of these allocations for you.
   - Define an entry label for the assembly file using `.def _fir`
   - Reference *a* and *x* using `.ref _`*a*, `_`*x*
   - Change the entry label from `start` to `_fir.` This is where you want your fir code to start when `MAIN11A.C` calls it.
   - Remove the stack allocation instructions.
   - Change your .sect “code” to .text. C places all code in the .text section. Reference `LAB11A.CMD` (the “code” section is not linked).
   - Remove the copy routine, `.copy` of `in6.dat`, stop conditions and all calls.
   - Remove the allocation for the `init` section as well as `table[16]`

5. After the `_fir` label, you need to write some code to access the parameters on the stack. Before writing any code, draw a picture of what the stack looks like prior to calling your assembly routine. If you’d like, comment this in your assembly routine and refer to as you write your code. Your diagram should look something like this: When `_fir` is called, the stack and accumulator look like this:

Return Address ← SP
Results
&y

AL = TAPS

6. So, the stack pointer (`SP`) points to the return address `PC`. An offset of +ONE from SP is the parameter `RESULTS` which needs to be loaded into BRC. An offset of +TWO from SP is the address of *y* which needs to be loaded into `ARn` (whichever AR you used to store the results).

Note: Please note that if you have pushed any registers (like ST0 or ST1), you will need to modify the picture of the stack as well as the following instructions.

---

7. Now, perform the following instructions to load the correct registers from the stack:

   - `STLM   A, BK` ; load BK with TAPS
   - `MVDK   *SP(1), *(BRC)` ; load BRC with #RESULTS-1
   - `MVDK   *SP(2), *(AR1)` ; load ARn with the address of y

   Make sure you remove the STM to the result AR register in your code as well as the STMs to BK and BRC.

8. Inside your `_fir` routine, you set the `FRCT` bit to use fractional mode. C expects `FRCT=0`, so you must `RSBX FRCT` before returning. Change your done: label to set `FRCT` to zero, then follow this instruction with a return.

9. Change any references to *a* or *x* to *_a* and *_x*.

10. Note: in a normal subroutine, you would want to make the single repeat and the pointer wrap based upon the passed parameter TAPS. To make it easy on yourself (to start with), simply hard code the values into the single repeat and pointer wrap. Then, once you get your code working, go back and make the necessary changes to the assembly routine.

Build and Simulate

11. Because we're now working with C, we'll need to check the settings for the C compiler. On the menu bar click:

    `Project` → `Options`

    Under the `Compiler` tab change `-g` to `-gks` in the command line switches box on the top. This will keep the assembly file from the compilation so we can inspect it. Make sure you have Load Program after Build checked under `Options` → `Program Load`.

12. Build the project. Remember that you can double-click on any error to immediately go to it.

13. When the build and load are complete, reset your system. You should see `B _c_int00` in the `CVECTORS.ASM` source file.

14. Type: `go main` on the command line. This will run through the C initialization routine in `BOOT.ASM` and stop at the main routine in `MAIN11A.C`.

15. Single step and check to make sure the proper values are loaded onto the stack in the proper order prior to the call to fir ( ). Then single step through your assembly code and ensure that these parameters are loaded into the proper registers. If all of this works properly, double check your math code. If everything looks good, hit the `Run` button. You do not need to set a breakpoint, because your assembly routine will return back to main and the execution will stop at the C exit routine which is an infinite branch. Graph your results.

16. We don't have much C code here to optimize, but let's see how you use the optimizer …

17. Under `Project` → `Options,` under the `Compiler` tab click on `Optimizer` in the `Category` box. Let's go ahead and pick `Level 3 - File` in the `Level` box. Run your code and graph your results.

18. You are now done with LAB11A. Congrats.

LAB11B – Procedure

As an alternative to processing the entire block in one call, let's decide to use the C language INTERRUPT capabilities and process another output EACH time a new input sample comes from the ADC.

1. Setup the INTERRUPT prototype to call the RINT ISR.
2. Place your FIR code in a subroutine. Call that subroutine with a CALL using the index of the latest sample that became available.
3. See LAB11B files in the solutions directory to see how this works.

Solutions

MAIN11A.C - Solution

```
// Define Sample and Tap sizes for function
#define RESULTS 185
#define TAPS 16

// Initialize Coefficient Table
int a[TAPS] =     {0x7FC, 0x7FD, 0x7FE, 0x7FF,
                   0x800, 0x801, 0x802, 0x803,
                   0x803, 0x802, 0x801, 0x800,
                   0x7FF, 0x7FE, 0x7FD, 0x7FC};

// Specify specific address for the result: y
#pragma DATA_SECTION (y,"yloc");
int y[RESULTS];

// include initialized x array
#include "in11.h"

extern void fir(int taps,int results,int *y);

main()
{
  // set wait states to zero using in-line assembly
  asm(" STM      #0,SWWSR");

  // call assembly FIR routine
  fir(TAPS,RESULTS,y);
}
```

DSP54.11 - 20

LAB11A.ASM - Solution

```
; stack looks like this upon entry to this asm routine:
;
;        RET_ADDR  <-- SP
;        RESULTS
;        &y
;        --
;        --
;
;        AL = TAPS

; allocate label definition here

        .mmregs
        .def _fir
        .ref _a,_x

        .text
_fir:   STLM     A,BK              ;load BK with TAPS (16)
        MVDK     *SP(1),*(BRC)     ;load BRC with RESULTS (185)
        MVDK     *SP(2),*(AR1)     ;load ARn with &y

        LD       #0,DP             ;set SST bit (saturate on store)
        ORM      #1,@PMST
        SSBX     FRCT              ;set FRCT bit (fractional mode)
        RSBX     OVM               ;clr OVM bit (overflow mode)
        SSBX     SXM               ;set SXM bit (sign extension)
```

DSP54.11 - 21

LAB11A.ASM - Solution (continued)

```
        STM     #1,AR0
        STM     #_a,AR2                     ;setup ARs for MAC
        STM     #_x,AR3

        RPTB    done-1
        MPY     *AR2+0%,*AR3+,A             ;1st product
        RPT     #14                         ;mult/acc 15 terms
        MAC     *AR2+0%,*AR3+,A
        MAR     *+AR3(-15)
        STH     A,*AR1+                     ;store result

done:   RSBX    FRCT
        RET                                 ;return
```

DSP54.11 - 22

LAB11A.CMD - Solution

```
main11a.obj
lab11a.obj
cvectors.obj
-o lab11a.out
-m lab11a.map
-c
-stack 0x100
-l c:\dsptls54\c54xcgt\rts.lib

MEMORY {
PAGE 1:         /* Data memory */
  SPRAM:    org = 00060h,  len = 00020h
  DARAM:    org = 00080h,  len = 00400h

PAGE 0:         /* Program memory */
  EPROM:    org = 0F000h,  len = 00F80h
  VECS:     org = 0FF80h,  len = 00080h
}

SECTIONS
{ .text   :> EPROM  PAGE 0
  vectors :> VECS   PAGE 0
  .bss    :> DARAM  PAGE 1
  .stack  :> DARAM  PAGE 1
  yloc    :> DARAM  PAGE 1
}
```

DSP54.11 - 23

Making a C54x System Work

Introduction

Hooking up memory and peripheral devices, programming wait states, relocating code, setting up the clock … these seemingly small items can end up being show-stoppers. In this module we'll take a look at these topics and others so we can smoothly transition our software to a real live system.

We've taken a different approach in this module from the rest of the workshop. Here we've attempted to cover every single topic that someone implementing a DSP system might care about. Obviously there are things that fall outside the scope of this, like choice of algorithm, sampling rates, etc.

There is a lot of detail in this module, probably too much to cover in depth and still stay awake. Your instructor will point out the major decisions and why they were made. Your implementation will be different, but you will still have to go through a similar process.

Module Topics

Making a C54x System Work **12-1**

Module Topics *12-2*

Introduction *12-3*

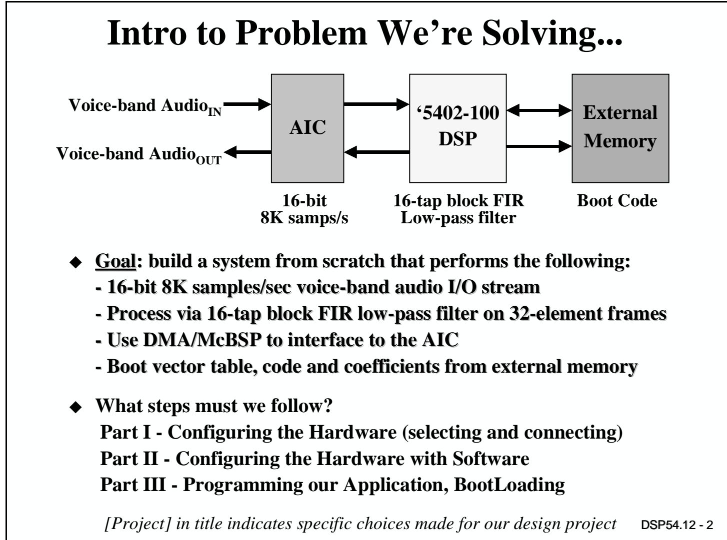
The Hardware *12-4*
Power Considerations 12-4
The Clock 12-5
Memory 12-6
The Analog Interface Circuit (AIC) 12-7
Connecting Unused Pins 12-8
JTAG 12-9
Hardware Troubleshooting 12-9

The Firmware *12-10*
Initial Clock Frequency 12-10
Programming The PLL 12-11
PLL Setup Code 12-11
Wait States 12-12
Waitstate Setup Code 12-13
Bank Switch Control 12-13
BSCR Setup Code 12-14
McBSP/AIC Equations 12-14
Setting Up McBSP0 12-15
McBSP Setup Code 12-17
Setting Up The AIC 12-17
AIC Setup Code 12-18
Setting Up The DMA 12-19
Data I/O 12-21
TImeline Analysis 12-21
DMA Setup Code 12-22
Turning On The Hardware Code 12-22

The Software *12-23*
Link.cmd and Vectors.asm 12-23
The Hardware Setup and The FIR Code 12-24
The Bootloader 12-27
HEX500 12-28
IDLE 12-28
Power Management Hints 12-29
BIOS and RTA 12-29
Need More Information? 12-30

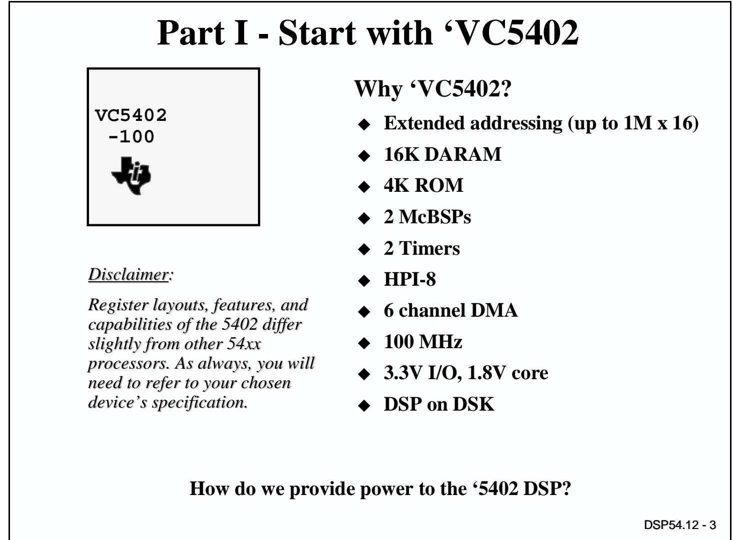
Additional Analog Information *12-32*

Introduction

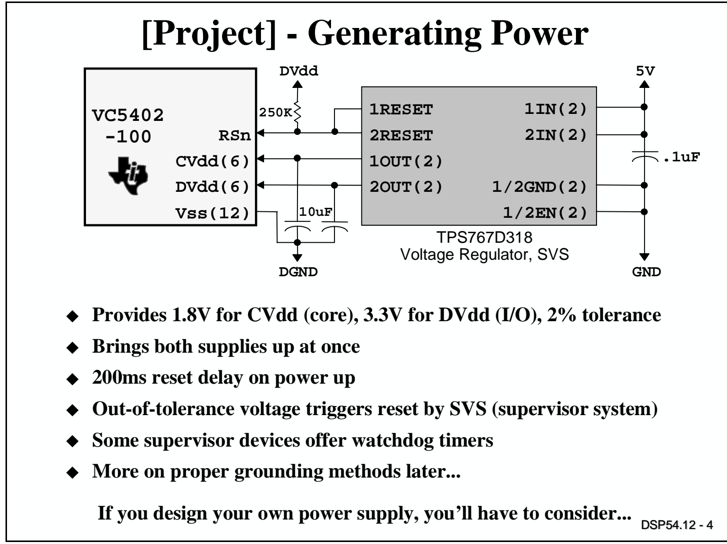


Throughout this module we will use the [Project] in the title of the slide to identify where we have made specific choices for our design project. Other slide may include more general information about concepts or selections that the 5402 may not possess.

The Hardware



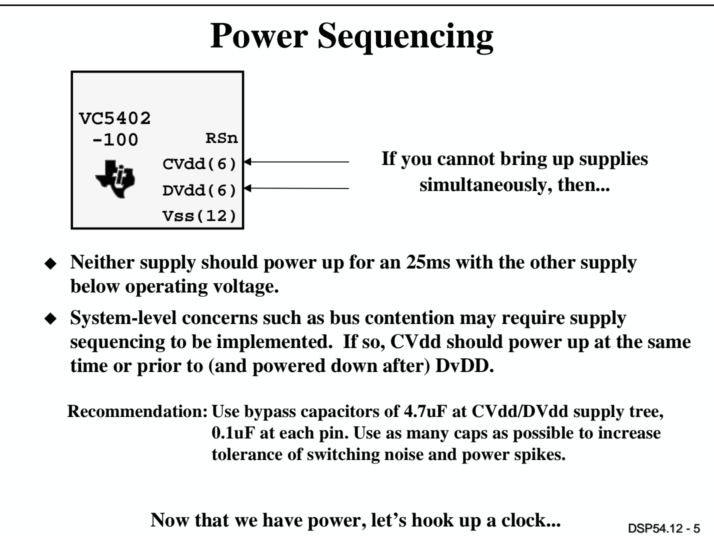
Power Considerations



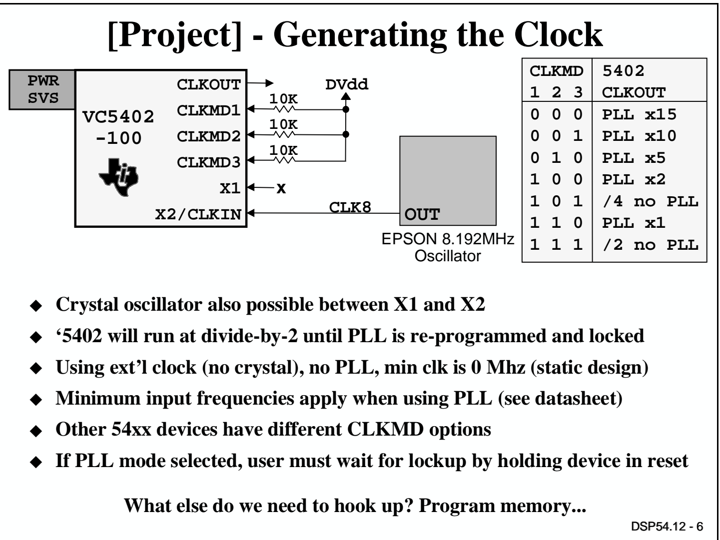
The following pages on the TI web site have excellent selection charts for this and other parts:

www.ti.com/sc/select

www.ti.com/sc/docs/products/msp/index.htm

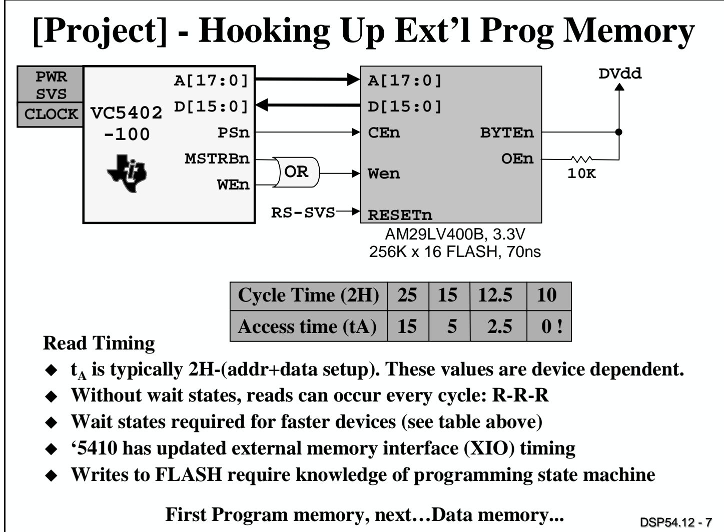


The Clock



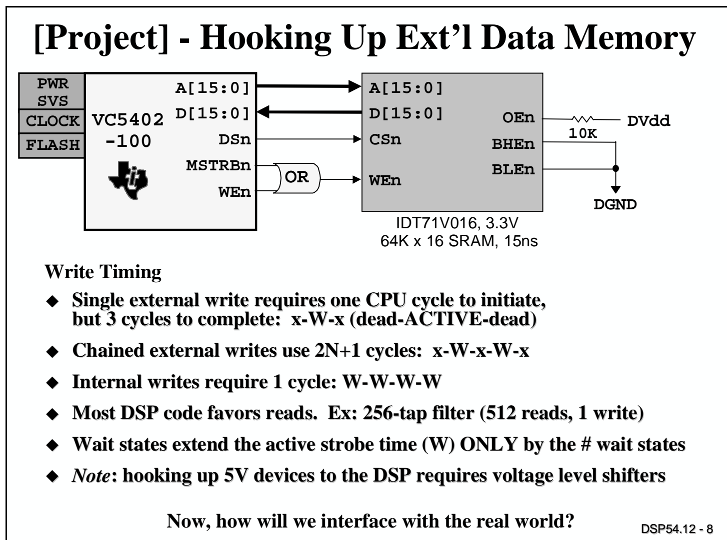
The selection of the clock is driven by a number of factors: PLL options, CPU speeds desired, system circuitry requirements and others. Your system may require more than one clock source to meet all of your goals.

Memory

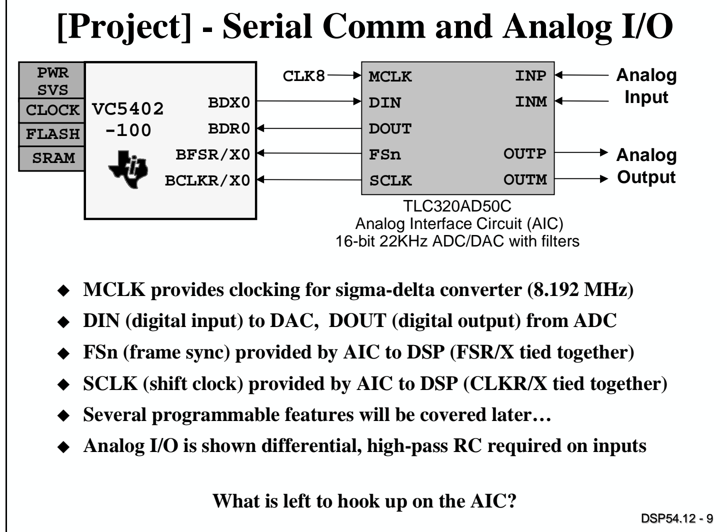


Cycle Time (2H)	25	15	12.5	10
Access time (tA)	15	5	2.5	0 !

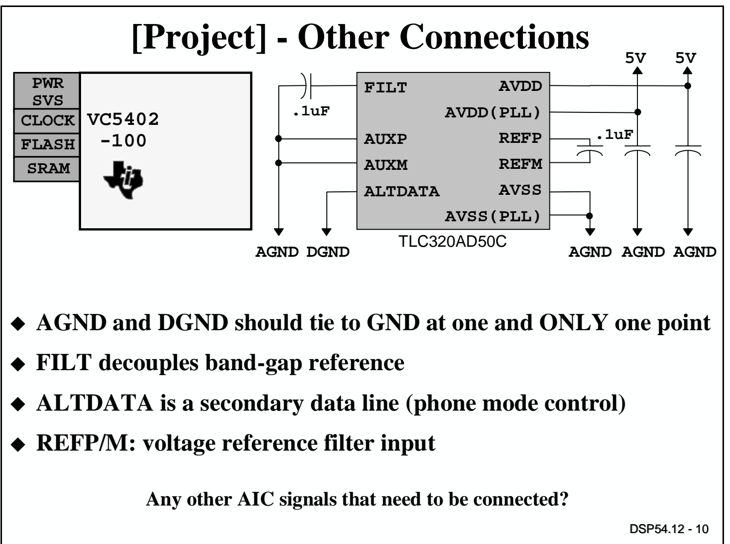
This type of FLASH contains the programming algorithm as an internal state machine. Your code will need to reflect knowledge of the specific algorithm your selected FLASH implements. You can of course use EPROM, ROM, RAM or any other kind of asynchronous memories for both program and data space. You will need to initialize volatile memories if your code maps initialized sections in them.



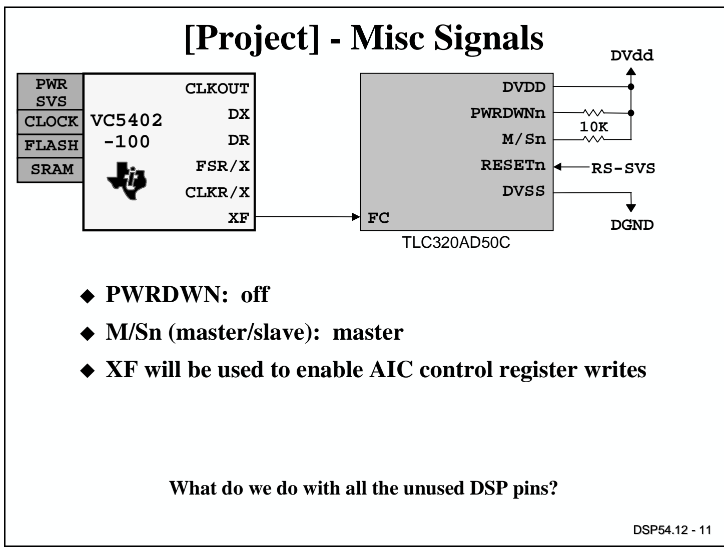
The Analog Interface Circuit (AIC)



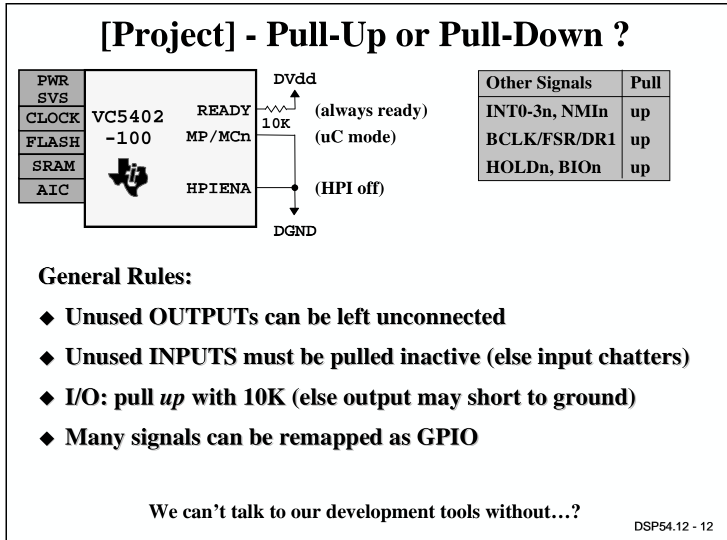
A simple low-pass RC network on the input limits inputs to the Nyquist rate. An RC smoothing filter on the output should also be implemented.



Any circuit with mixed signals (digital and analog) should implement separate digital and analog grounds to reduce noise. Bring these grounds together at one and only one point. Failure to do so will result in ground loop currents between the connections, higher system noise levels and lower effective resolution.

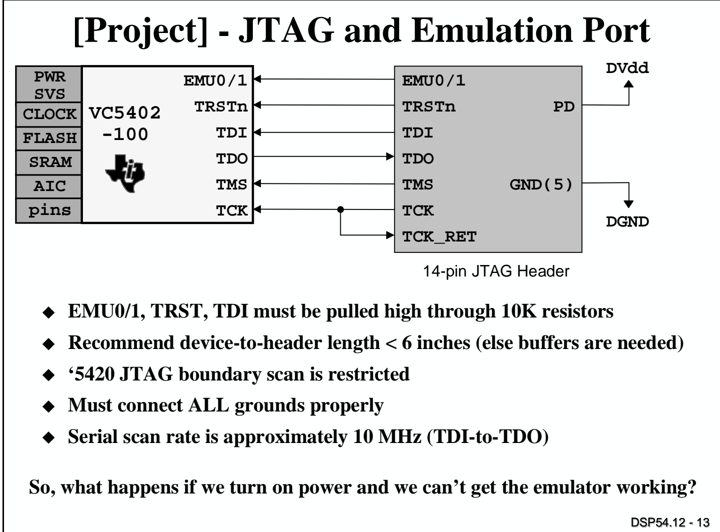


Connecting Unused Pins



CMOS type inputs without a pull-up or pull-down will assume some intermediate voltage. System noise will cause them to go above and below the switching levels causing “chatter” and wasting power.

JTAG



JTAG connections MUST be clean and free of noise. Failure to provide good connections for the emulator results in significant headaches for the designer.

Hardware Troubleshooting

Hardware Troubleshooting

Items to check (in order)

- **Power supply working properly (check DVdd/CVdd/5v)?**
- **Check all grounds for continuity.**
- **Is CLKOUT toggling?**
- **Is RSn working properly (or is the CPU still in reset)? ‘542x devices have 2 reset signals, so make sure they are both working.**
- **Perform an EMURST from the debugger. If it fails …**
  - **- Are TRST and TCK functioning properly?**
  - **- Are the JTAG data signals (TDI, TDO, TMS) toggling?**
  - **- Are EMU0/1 properly pulled up?**
  - **- Is PD connected to DVdd?**
- **Is your debugger (CCS) set up for the correct 54xx device and order of cores if using multi-core device (‘542x)?**
- **For more EMU info, refer to SPRA439 “Emulation Fundamentals ...”**

After verifying that the h/w works, we need to write code to set up the h/w...

DSP54.12 - 14

The Firmware

Part II - Software Part of the Hardware

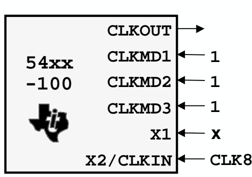
Configuring the Hardware Via Software:

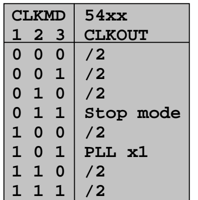
- **Set initial clock frequency and programming the PLL**
- **Set up software and hardware wait states, bank switching**
- **Determine bit settings for all peripherals: McBSP, AIC, DMA**
- **Write code to program the peripherals**

DSP54.12 - 15

Initial Clock Frequency

Setting '54xx Initial Clock Frequency





CLKMD 1 2 3	54xx CLKOUT
0 0 0	/2
0 0 1	/2
0 1 0	/2
0 1 1	Stop mode
1 0 0	/2
1 0 1	PLL x1
1 1 0	/2
1 1 1	/2

- **'5402 supports auto-programming of PLL with CLKMDx pins.**
  - PLL programming not necessary unless change of frequency desired or clock multiplier not sufficient (we're doing a x12 for the project)
  - For all devices, do not change CLKMDx pins after reset
- **Other '54xx devices require the *user* to program the PLL after reset**
- **User will normally power up device in /2 mode (see datasheet).**
- **Early '54x (e.g. 541) devices do not have a software programmable PLL**

So, how do we manually program the PLL to a desired frequency ?

DSP54.12 - 16

See your devices datasheet for its clock mode selections.

Programming The PLL

[Project] - Programming the PLL

- Power up in div-by-2 mode, CLKIN=8.192MHz, CLKOUT= 4.096MHz
- However, we actually want CLKOUT to be ~100MHz

15 12	11	10 3	2	1	0
PLLMUL	PLLDIV	PLLCOUNT	PLLON/OFF	PLLNDIV	PLLSTATUS

Clock Mode Register

1. **Determine Initial Clock Mode Register Value:** (`STM #0B7FCh,CLKMD`)
   a. Select PLLMUL ([x+1] * CLKIN) and PLLDIV (divide: yes/no) to your desired values. Ex: x=11 for ~100MHz CLKOUT, PLLDIV=0 (no divide)
   b. Select PLLCOUNT (Power up CLKOUT cycles * 16 to lock up). CLKOUT=4MHz, PLLCOUNT=FFh, Lock up time = 255*16*250ns = 1ms (Note: worst case time to lock on is 30us)
   c. Select PLLON/OFF. Ex: ON (1) (begins PLL lock-up)
2. Tell processor WHEN to switch to new frequency: (`ORM #2, CLKMD`)
   - if PLLCOUNT is non-zero, no switch will occur until PLLCOUNT = 0.

Let's do the coding...

DSP54.12 - 17

PLL Setup Code

[Project] - Programming the PLL

```
;Setup CLKMD Register
      STM #0B7FCh,CLKMD
;Tell PLL to switch
;when PLLCOUNT = 0
      ORM #2, CLKMD
```

- 5402 includes auto-lock feature
- PLL locks on after 30us to new clock frequency
- Could set PLLCOUNT lower if worst case lockup time is met
- You have 3 options regarding WHEN your system switches from low to high frequency:
  1. Use auto-switch (e.g. '5402 PLL): PLL locks automatically after Xns
  2. If your system is sensitive to the switch time, simply poll PLLSTATUS:
     a. loop until auto-switch occurs
     b. Set PLLNDIV=1 when locked

Next, let's set up the proper wait states...

DSP54.12 - 18

Wait States

‘5402 Software Wait States

15	14–12	11–9	8–6	5–3	2–0
XPA	I/O	Hi Data	Low Data	Hi Prog	Low Prog

SWWSR: Software Wait State Register

SWCR: Software Wait State Control Register

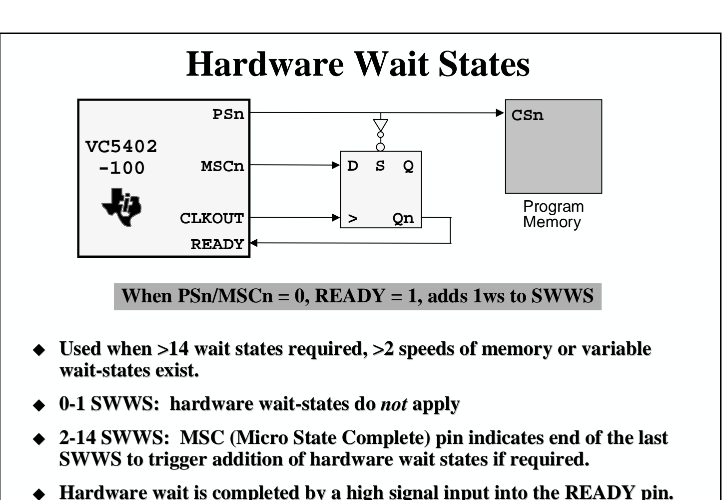
15–1	0
Rsvd	SWSM

- **Wait state: addition of ONE clock cycle to external memory access time**
- **SWWSR (3 bit fields) = 0 to 7**
  SWCR (SWSM) = 1, s/w wait-state multiplier bit doubles SWWSR value
- **On reset, all Program, Data and I/O is 7 wait states (SWSM = 0)**
- **XPA (Extended Program Address Control Bit). 0: every 64K page in program space split in half. 1: Low Prog determines ENTIRE space (varies by device).**
- **On last software wait state (2-14), MSC will go LOW for 1 cycle.**

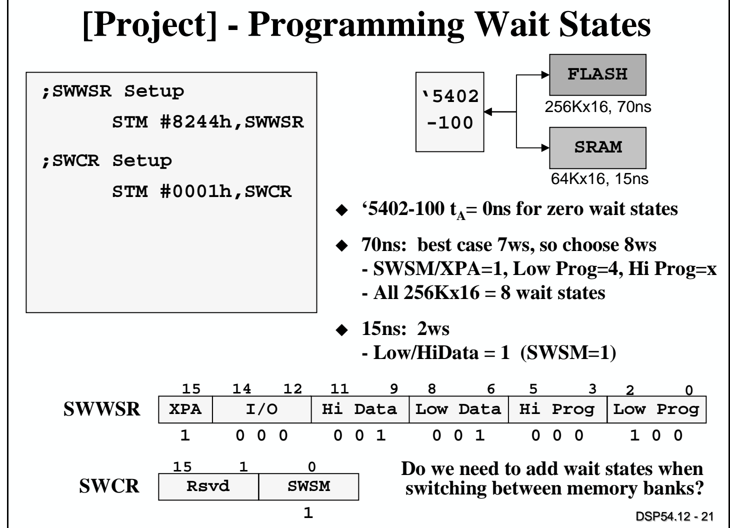
XPA=0: 0 – Low Prog; x8000h – Hi Prog; xFFFFh
XPA=1: 0 – Low Prog; FFFFFh
0 – Low Data; 8000h – Hi Data; FFFFh
0 – I/O; FFFFh

What if this software setup is not sufficient?

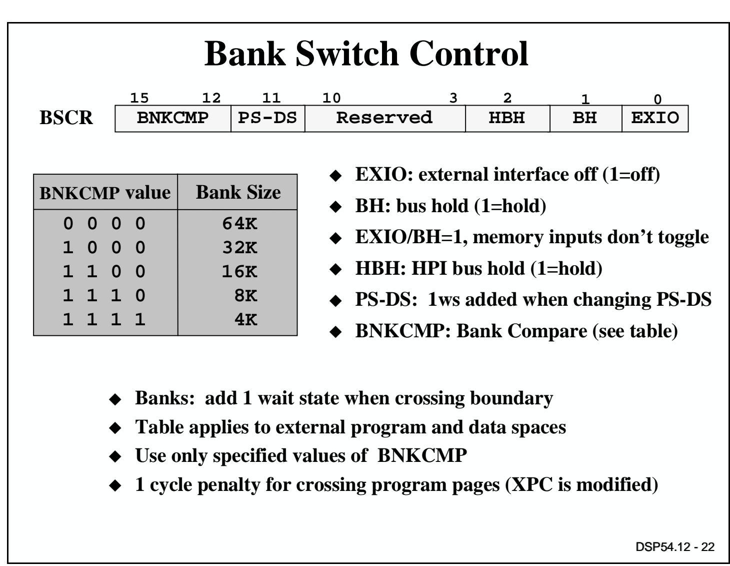
DSP54.12 - 19



Waitstate Setup Code

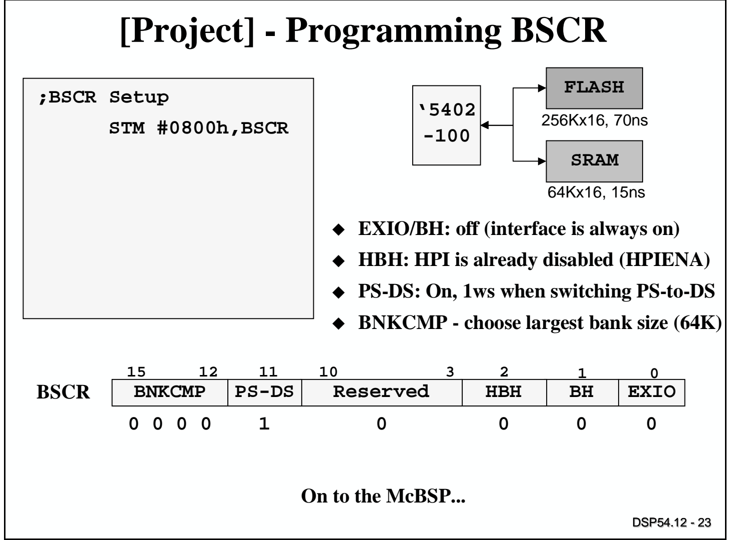


Bank Switch Control

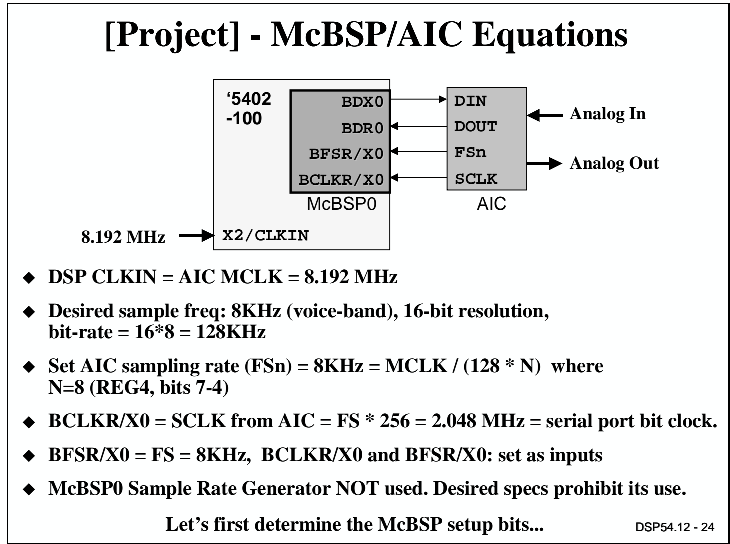


BNKCMP value	Bank Size
0 0 0 0	64K
1 0 0 0	32K
1 1 0 0	16K
1 1 1 0	8K
1 1 1 1	4K

BSCR Setup Code



McBSP/AIC Equations



Setting Up McBSP0

[Project] - McBSP0 Setup

Reg	Bit(s)	Name	Description	Value	Note
SPCR10	15	DLB	Digital Loopback on/off?	0	off
SA-00h	14-13	RJUST	Right justify in DRR?	10	left justify
	12-11	CLKSPP	Clock Stop mode (SPI)	00	no SPI
	10-8	[rsvd]	[reserved]	00	
	7	DXENA	Enable DX delay?	0	no delay
	6	ABIS	A-BIS mode (any bit delay)	0	none
	5-4	RINTM	Rcv interrupt mode	00	interrupt on RRDY
	3-1		Error/status fields	000	not used
	0	RRST	Receiver reset	0	keep in reset
				4000h	FINAL VALUE
SPCR20	15-10	[rsvd]	[reserved]	000000	
SA-01h	9	FREE	Run free w/EMU stop?	0	not FREE running
	8	SOFT	Finish current word?	1	yes
	7	FRSTn	FS logic reset?	0	yes
	6	GRSTn	SRGR reset?	0	yes
	5-4	XINTM	Xmt interrupt mode	00	interrupt on XRDY
	3-1		Error/status fields	000	not used
	0	XRST	Transmit reset	0	keep in reset
				0100h	FINAL VALUE

SPCRxy - Serial Port Control Register x/y (Regx/McBSPy)
SA: Sub address used for programming

DSP54.12 - 25

[Project] - McBSP0 Setup

Reg	Bit(s)	Name	Description	Value	Note
PCR0	15-14	[rsvd]	[reserved]	00	
SA-0Eh	13	XIOEN	DX: GPIO?	0	DX normal
	12	RIOEN	DR/CLKS GPIO?	0	DR/CLKS normal
	11	FSXM	FSX in/out?	0	in: gen'd by AIC
	10	FSRM	FSR in/out?	0	in: gen'd by AIC
	9	CLKXM	CLKX in/out?	0	in: gen'd by AIC
	8	CLKRM	CLKR in/out?	0	in: gen'd by AIC
	7	[rsvd]	[reserved]	0	
	6	CLKS_STAT	value as GPIO	0	not used
	5	DX_STAT	value as GPIO	0	not used
	4	DR_STAT	value as GPIO	0	not used
	3	FSXP	FSX polarity	1	active low, AIC:FS
	2	FSRP	FSR polarity	1	active low, AIC:FS
	1	CLKXP	CLKX polarity	0	xmt on rising edge
	0	CLKRP	CLKR polarity	0	rcv on falling edge
				000Ch	FINAL VALUE

PCRy - Pin Control Register (McBSPy)

DSP54.12 - 26

[Project] - McBSP0 Setup

Reg	Bit(s)	Name	Description	Value	Note
RCR10	15	[rsvd]	[reserved]	0	
SA-02h	14-8	RFRLEN1	Rcv Frame Length 1	00h	1 word/frame
	7-5	RWDLEN1	Rcv Word Length 1	010	16-bit
	4-0	[rsvd]	[reserved]	00000	
				0040h	FINAL VALUE
RCR20	15	RPHASE	Rcv: 1/2 phases?	0	1 phase
SA-03h	14-8	RFRLEN2	Rcv Frame Length 2	00h	not used
	7-5	RWDLEN2	Rcv Word Length 2	000	not used
	4-3	RCOMPAND	Rcv Compand mode	00	not used
	2	RFIG	Rcv Frame Ignore	0	don't ignore
	1-0	RDATDLY	FSR-DR delay (0,1,2-bit)	00	no delay
				0000h	FINAL VALUE

RCRxy - Receive Control Register x/y (Regx/McBSPy)
Dual-phase frames used by AC'97

DSP54.12 - 27

[Project] - McBSP0 Setup

Reg	Bit(s)	Name	Description	Value	Note
XCR10	15	[rsvd]	[reserved]	0	
SA-04h	14-8	XFRLEN1	Xmt Frame Length 1	00h	1 word/frame
	7-5	XWDLEN1	Xmt Word Length 1	010	16-bit
	4-0	[rsvd]	[reserved]	00000	
				0040h	FINAL VALUE
XCR20	15	XPHASE	Xmt: 1/2 phases?	0	1 phase
SA-05h	14-8	XFRLEN2	Xmt Frame Length 2	00h	not used
	7-5	XWDLEN2	Xmt Word Length 2	000	not used
	4-3	XCOMPAND	Xmt Compand mode	00	not used
	2	XFIG	Xmt Frame Ignore	0	don't ignore
	1-0	XDATDLY	FSX-DX delay (0,1,2-bit)	00	no delay
				0000h	FINAL VALUE

XCRxy - Transmit Control Register x/y (Regx/McBSPy)
Dual-phase frames used by AC'97

DSP54.12 - 28

[Project] - McBSP0 Setup

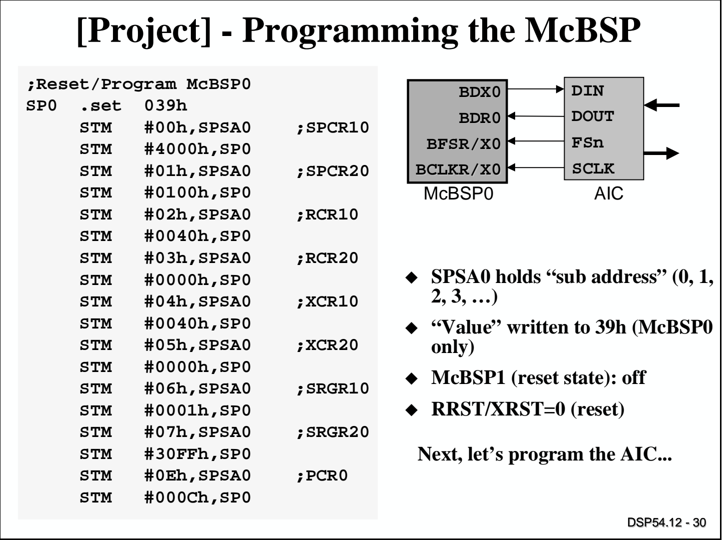
Reg	Bit(s)	Name	Description	Value	Note
SRGR10	15-8	FWID	Frame width	00h	not used
SA-06h	7-0	CLKGDV	CLKG Divider (1-256)	01h	default, not used
				0001h	FINAL VALUE
SRGR20	15	GSYNC	re-sync to FSG or free run?	0	free run, not used
SA-07h	14	CLKSP	CLKS polarity	0	no CLKS on 5402
	13	CLKSM	input CLKOUT/CLKS?	1	CLKOUT, not used
	12	FSGM	FSX=FSG or DXR-XSR ?	1	FSG, not used
	11-0	FPER	Frame period, 1-4096	0FFh	256, not used
				30FFh	FINAL VALUE

SRGRxy - Sample Rate Generator Register x/y (Regx/McBSPy)

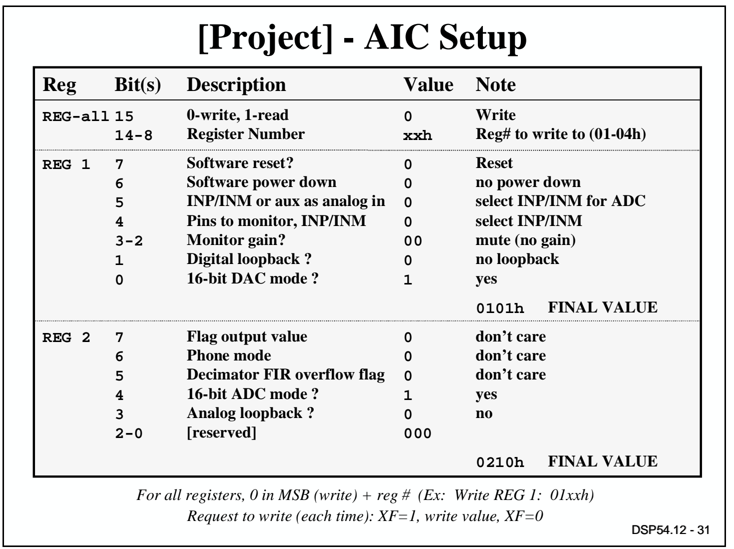
Multi-channel Registers (MCR10/20, RCERA/B, XCERA/B) not used. Default mode on reset is "non-multi-channel mode"

DSP54.12 - 29

McBSP Setup Code



Setting Up The AIC



Reg	Bit(s)	Description	Value	Note
REG-all	15	0-write, 1-read	0	Write
	14-8	Register Number	xxh	Reg# to write to (01-04h)
REG 1	7	Software reset?	0	Reset
	6	Software power down	0	no power down
	5	INP/INM or aux as analog in	0	select INP/INM for ADC
	4	Pins to monitor, INP/INM	0	select INP/INM
	3-2	Monitor gain?	00	mute (no gain)
	1	Digital loopback ?	0	no loopback
	0	16-bit DAC mode ?	1	yes
				0101h FINAL VALUE
REG 2	7	Flag output value	0	don't care
	6	Phone mode	0	don't care
	5	Decimator FIR overflow flag	0	don't care
	4	16-bit ADC mode ?	1	yes
	3	Analog loopback ?	0	no
	2-0	[reserved]	000	
				0210h FINAL VALUE

For all registers, 0 in MSB (write) + reg # (Ex: Write REG 1: 01xxh)
Request to write (each time): XF=1, write value, XF=0

[Project] - AIC Setup

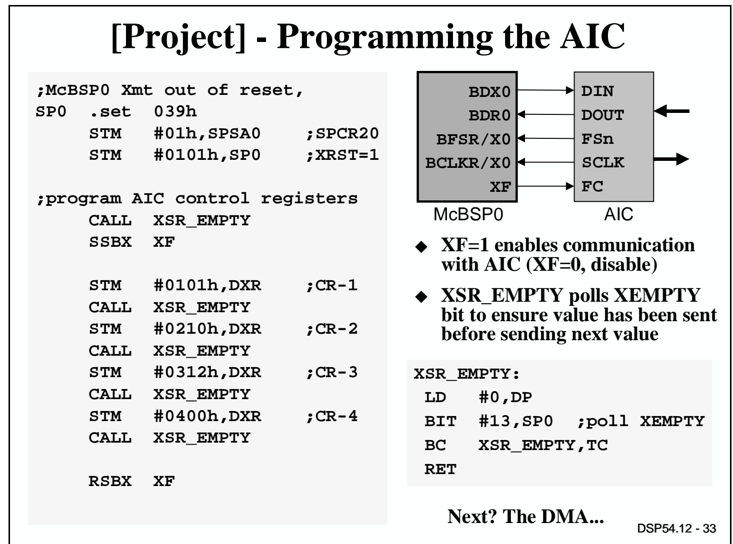
Reg	Bit(s)	Description	Value	Note
REG 3	7-6	# slave devices ?	00	None
	5-0	Frame sync delay timing	12h	not used (FSD)
			0312h	FINAL VALUE
REG 4	7	External sample clock y/n ?	0	no
	6-4	Sample frequency select	000	Equation (N = 8)
	3-2	Analog input gain ?	00	0db gain
	1-0	Analog output gain ?	00	0db gain
	3	Analog loopback ?	0	no
			0400h	FINAL VALUE

For all registers, 0 in MSB (write) + reg # (Ex: Write REG 1: 01xxh)

Request to write (each time): XF=1, write value, XF=0

DSP54.12 - 32

AIC Setup Code



Setting Up The DMA

[Project] - DMA Setup (Main)

Reg	Bit(s)	Name	Description	Value	Note
DMPREC	15	FREE	Run free w/EMU stop?	0	not FREE running
	14	[rsvd]	[reserved]	0	
	13	DPRC5	Ch5 priority? hi/low?	0	low
	12	DPRC4	Ch4 priority? hi/low?	0	low
	11	DPRC3	Ch3 priority? hi/low?	0	low (output to AIC)
	10	DPRC2	Ch2 priority? hi/low?	1	hi (input from AIC)
	9	DPRC1	Ch1 priority? hi/low?	0	low
	8	DPRC0	Ch0 priority? hi/low?	0	low
	7-6	INTOSEL	Interrupt selector	01	Select Ch2, Ch3, timer
	5	DE5	Enable Ch5?	0	no
	4	DE4	Enable Ch4?	0	no
	3	DE3	Enable Ch3?	1	yes
	2	DE2	Enable Ch2?	1	yes
	1	DE1	Enable Ch1?	0	no
	0	DE0	Enable Ch0?	0	no
				044Ch	FINAL VALUE

DMPREC - Channel Priority and Enable Control Register
INTOSEL: '5402 has 3 choices for interrupt selection, we chose to use channels 2/3 and timer. Also forces IMR/IFR to bits 7/10/11 to TIMER1/CH2/CH3 respectively
DEx: Programming these bits START the transfer

DSP54.12 - 34

[Project] - DMA Channel 2 Setup

Reg	Bit(s)	Name	Description	Value	Note
DMSRC2	15-0		16-bit Src Address (Ch2)	#DRR10	input buffer src
DMDST2	15-0		16-bit Dest Address (Ch2)	#80h	input buffer dst
DMCTR2	15-0		16-bit Elem Count (Ch2)	#1Fh	32 elements/frame
DMSFC2	15-12	DSYN	Sync Event	0001	REVT0 (RRDY=1)
	11	DBLW	Double word?	0	no (16-bit element)
	10-8	[rsvd]	[reserved]	000	
	7-0	FRMCNT	Frame Count	02h	#frames = 3
				1002h	FINAL VALUE
DMMCR2	15	AUTOINIT	Auto init on/off?	1	yes
	14	DINM	DMA interrupts on/off?	1	on
	13	IMOD	When to gen interrupt	1	int @ end of frm/blk
	12	CTMOD	ABU/multi-frame mode	0	multi-frame
	11	[rsvd]	[reserved]	0	
	10-8	SIND	Source index	000	no modification (DRR)
	7-6	DMS	Source space select	01	data space
	5	[rsvd]	[reserved]	0	
	4-2	DIND	Destination index	001	post increment (80h)
	1-0	DMD	Destination space select	01	data space
				0E045h	FINAL VALUE

DSP54.12 - 35

[Project] - DMA Channel 3 Setup

Reg	Bit(s)	Name	Description	Value	Note
DMSRC3	15-0		16-bit Src Address (Ch3)	#2000h	output buffer src
DMDST3	15-0		16-bit Dest Address (Ch3)	#DXR10	output buffer dst
DMCTR3	15-0		16-bit Elem Count (Ch3)	#1Fh	32 elements/frame
DMSFC3	15-12	DSYN	Sync Event	0010	XEVT0 (XRDY=1)
	11	DBLW	Double word?	0	no (16-bit element)
	10-8	[rsvd]	[reserved]	000	
	7-0	FRMCNT	Frame Count	00h	#frames = 1
				2000h	FINAL VALUE
DMMCR3	15	AUTOINIT	Auto init on/off?	0	no
	14	DINM	DMA interrupts on/off?	0	off
	13	IMOD	When to gen interrupt	0	not used
	12	CTMOD	ABU/multi-frame mode	0	multi-frame
	11	[rsvd]	[reserved]	0	
	10-8	SIND	Source index	001	post increment (2000h)
	7-6	DMS	Source space select	01	data space
	5	[rsvd]	[reserved]	0	
	4-2	DIND	Destination index	000	no modification (DXR)
	1-0	DMD	Destination space select	01	data space
				0141h	FINAL VALUE

DSP54.12 - 36

[Project] - DMA Setup (Misc, Reload)

Reg	Bit(s)	Name	Description	Value	Note
DMSRCP	6-0		7-bit Src Prog Page Addr	0h	not used
DMDSTP	6-0		7-bit Dest Prog Page Addr	0h	not used
DMIDX0	15-0		Element Index 0	0h	not used
DMIDX1	15-0		Element Index 1	0h	not used
DMFRI0	15-0		Frame Index 0	0h	not used
DMFRI1	15-0		Frame Index 1	0h	not used
DMGSA	15-0		Global Src Addr Reload	#DRR10	auto-init, Ch2 src
DMGDA	15-0		Global Dest Addr Reload	#80h	auto-init, Ch2 dst
DMGCR	15-0		Global Elem Cnt Reload	001Fh	32 elements
DMGFR	7-0		Global Frm Cnt Reload	02h	Ch2 - 3 frames

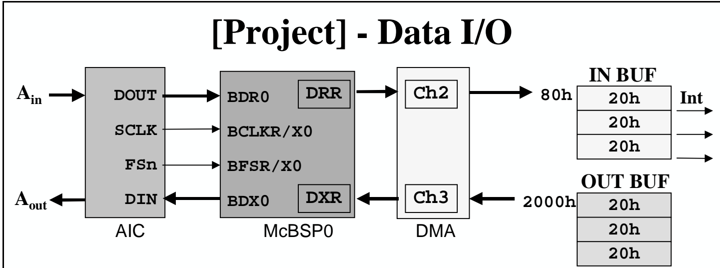
- No transfers to/from program space
- Not using indexing
- Global reload registers used to auto-initialize Channel 2 (input buffer)

Let’s step back and review the data flow...

DSP54.12 - 37

Data I/O

[Project] - Data I/O

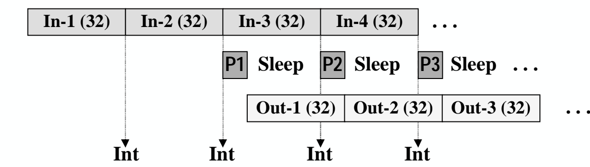


- System Goal: Process a 16-tap block FIR filter on incoming data
- Design choices made
  - McBSP0: aribitrary (could have chosen McBSP1)
  - Ch2/3: based on INTOSEL limitations for '5402
- '5402 has two blocks of 8Kx16 DARAM. To avoid memory conflicts, code/coeffs will be placed at 1000/3000h respectively
- Number of buffers chosen to ease programming boundary conditions
- Size of buffers chosen to limit I/O latency

Let's make sure dataflow <u>timing</u> works... DSP54.12 - 38

TImeline Analysis

[Project] - Timeline Analysis



- Input first 32 words - ignore first interrupt, (32 words @ 8KHz) = 4000us
- Input 2nd 32 words - interrupt starts processing of In-1
  - In-1 processing requires 47 input samples (15 from In-2) to complete
- Processing takes ~22 cycles/result * 32 results (@100MHz) = 7.04us
- CPU writes results to Out-1 and reprograms DMA3
- Begin first transfer to AIC
- Latency from first input to first output: ~8ms
- Go to power-down state between end of processing and input interrupt

Let's finish programming the DMA... DSP54.12 - 39

DMA Setup Code

[Project] - Programming the DMA

```
;DMA Channel 2 Setup
  STM   #0Ah,DMSA
  STM   #DRR,DMSDI      ;DMSRC2 w/auto-inc
  STM   #80h,DMSDI      ;DMDST2
  STM   #1Fh,DMSDI      ;DMCTR2
  STM   #1002h,DMSDI    ;DMSFC2
  STM   #0E045h,DMSDI   ;DMMCR2

;DMA Channel 3 Setup
  STM   #0Fh,DMSA
  STM   #2000h,DMSDI    ;DMSRC3 w/auto-inc
  STM   #DXR,DMSDI      ;DMDST3
  STM   #1Fh,DMSDI      ;DMCTR3
  STM   #2000,DMSDI     ;DMSFC3
  STM   #0141h,DMSDI    ;DMMCR3

;DMA Global Reload Reg Setup
  STM   #24h,DMSA
  STM   #DRRh,DMSDI     ;DMGSA w/auto-inc
  STM   #80h,DMSDI      ;DMGDA
  STM   #1Fh,DMSDI      ;DMGCR
  STM   #02h,DMSDI      ;DMGFR
```

- save DMPREC code until we are ready to begin transfers
- Registers not used - not programmed

DSP54.12 - 40

Turning On The Hardware Code

[Project] - Turning it ON...

```
;Check to ensure PLL is locked
       LD   #0,DP
loop:  BIT  CLKMD,#15-0
       BC   loop,NTC

;En/Disable DMA (Ch2-on, Ch3-off)
       STM  #0444h,DMPREC

;McBSP0 (rcv) out of reset
       ORM  #1,SPCR10

;AIC out of reset
       CALL  XSR_EMPTY
       SSBX  XF
       STM   #0181h,DXR    ;CR-1
       CALL  XSR_EMPTY
       RSBX  XF

;turn on interrupts
       STM   #0400h,IMR   ;DMA-INT CH2
       STM   #0FFFFh,IFR  ;Clr IFR
```

- Analog path now active
- INTM left inactive until just before main code begins
- First interrupt will not occur for 4ms (400,000 cycles)

DSP54.12 - 41

The Software

Part III - Software

Programming the bootloader and application:

- Determine vector table and linker command options
- Write fir_ISR and application setup code
- Use HEX500 to create a boot table, discuss bootload options
- Discuss power down options (IDLE)
- Review power management hints
- See how BIOS and RTA can assist us

DSP54.12 - 42

Link.cmd and Vectors.asm

[Project] - Link.cmd, Vector Table

```
project.obj
vectors.obj
-o project.out

MEMORY {
PAGE 1: INBUF:  org = 00080h, len = 00060h
        CODE:   org = 00100h, len = 00400h
        VECS:   org = 00500h, len = 00100h
        OUTBUF: org = 02000h, len = 00060h
        COEFF:  org = 02060h, len = 00200h
       DARAM2:  org = 03000h, len = 01000h
PAGE 0: EPROM:  org = 0F000h, len = 00F80h
}
SECTIONS
{in_bufs  :> INBUF PAGE 1
 out_bufs :> OUTBUF PAGE 1
 coeffs   :> LOAD=EPROM PAGE 0, RUN=COEFF PAGE 1
 code     :> LOAD=EPROM PAGE 0, RUN=CODE PAGE 1
 vectors  :> LOAD=EPROM PAGE 0, RUN=VECS PAGE 1
 STK      :> DARAM2 PAGE 1
 vars     :> COEFF PAGE 1
}
```

- The following sections are booted:
  - coeffs
  - code
  - vectors
- Bootloader moves code from Program space to Data Space. Bootloader sets OVLY bit to one.
- Vectors.ASM:

```
;unused: RETE
;Ch2 Int @68h
DMAC2:
  B  fir_isr
```

Let's now review ALL of the code...

DSP54.12 - 43

The Hardware Setup and The FIR Code

[Project] - Project.ASM

```
;** .set statements **
        .mmregs
DMPREC  .set    54h             ;Channel Priority and Enable Control
DMSA    .set    55h             ;DMA sub-address
DMSDI   .set    56h             ;DMA write without indexing
DMSDN   .set    57h             ;DMA write with indexing
SPSA0   .set    38h             ;McBSP0 sub-address
SP0     .set    039h            ;Write for McBSP0 sub-addressed regs
DRR10   .set    21h             ;Data Receive for McBSP0
DXR10   .set    23h             ;Data Transmit for McBSP0
SWCR    .set    2bh             ;Software Wait State

;** allocate aligned circular buffers for input and output **
x       .usect  "in_bufs",96
bos     .usect  "STK",128
FLAG1   .usect  "vars",3        ;signal first time thru input routine
FLAG2   .set    FLAG1+1         ;signal first time thru output routine
COUNT   .set    FLAG1+2         ;which buffer is being processed? 1,2,3
y       .usect  "out_bufs",96

;** allocate 16 initialized coeffs of 1/16th each **
        .sect   "coeffs"
a       .int    800h,800h,800h,800h
        .int    800h,800h,800h,800h
        .int    800h,800h,800h,800h
        .int    800h,800h,800h,800h
```

DSP54.12 - 44

```
        .sect   "code"
;*******************   H/W Setup Code   *****************

;** PLL **
start:  STM     #0B7FCh,CLKMD  ;Setup CLKMD Register
        ORM     #2,CLKMD       ;Tell PLL to switch when PLLCOUNT = 0

;** SWWSR/SWCR/BSCR **
        STM     #8244h,SWWSR   ;SWWSR Setup
        STM     #0001h,SWCR    ;SWCR Setup
        STM     #0800h,BSCR    ;BSCR Setup

;** Reset/Program McBSP0 **
        STM     #00h,SPSA0     ;SPCR10
        STM     #4000h,SP0
        STM     #01h,SPSA0     ;SPCR20
        STM     #0100h,SP0
        STM     #02h,SPSA0     ;RCR10
        STM     #0040h,SP0
        STM     #03h,SPSA0     ;RCR20
        STM     #0000h,SP0
        STM     #04h,SPSA0     ;XCR10
        STM     #0040h,SP0
        STM     #05h,SPSA0     ;XCR20
        STM     #0000h,SP0
        STM     #06h,SPSA0     ;SRGR10
        STM     #0001h,SP0
        STM     #07h,SPSA0     ;SRGR20
        STM     #30FFh,SP0
        STM     #0Eh,SPSA0     ;PCR0
        STM     #000Ch,SP0
```

DSP54.12 - 45

```
;** McBSP0 Xmt out of reset **
       STM    #01h,SPSA0           ;SPCR20
       STM    #0101h,SP0           ;XRST=1

;** program AIC control registers **
       CALL   XSR_EMPTY
       SSBX   XF
       STM    #0101h,DXR10         ;CR-1
       CALL   XSR_EMPTY
       STM    #0210h,DXR10         ;CR-2
       CALL   XSR_EMPTY
       STM    #0312h,DXR10         ;CR-3
       CALL   XSR_EMPTY
       STM    #0400h,DXR10         ;CR-4
       CALL   XSR_EMPTY
       RSBX   XF

;** DMA Channel 2 Setup **
       STM    #0Ah,DMSA
       STM    #DRR10,DMSDI         ;DMSRC2 w/auto-inc
       STM    #80h,DMSDI           ;DMDST2
       STM    #1Fh,DMSDI           ;DMCTR2
       STM    #1002h,DMSDI         ;DMSFC2
       STM    #0E045h,DMSDI        ;DMMCR2

;** DMA Channel 3 Setup **
       STM    #0Fh,DMSA
       STM    #2000h,DMSDI         ;DMSRC3 w/auto-inc
       STM    #DXR10,DMSDI         ;DMDST3
       STM    #1Fh,DMSDI           ;DMCTR3
       STM    #2000,DMSDI          ;DMSFC3
       STM    #0141h,DMSDI         ;DMMCR3
```

DSP54.12 - 46

```
;** DMA Global Reload Reg Setup **
       STM    #24h,DMSA
       STM    #DRR10,DMSDI         ;DMGSA w/auto-inc
       STM    #80h,DMSDI           ;DMGDA
       STM    #1Fh,DMSDI           ;DMGCR
       STM    #02h,DMSDI           ;DMGFR

;** Set PMST Register to proper value   **
;**                                     **
;** IPTR=500h (bootloaded vector table) **
;** MP/MC  = 0 (should be 0 already)    **
;** OVLY   = 1 (should be 1 already)    **
;** AVIS   = 0 (off)                    **
;** DROM   = 0 (off)                    **
;** CLKOFF = 1 (off)                    **
;** SMUL   = 0 (off)                    **
;** SST    = 1 (on)                     **

       STM     #0525h,PMST

;** Check to ensure PLL is locked **
       LD     #0,DP
plloop:
       BITF   @CLKMD,#1            ;loop until PLLSTATUS=1
       BC     plloop,NTC           ;(PLL Locked)

;** Enable DMA Channels 2 & 3 **
       STM    #044Ch,DMPREC
```

DSP54.12 - 47

```
;** McBSP0 (rcv) out of reset **
       STM    #00h,SPSA0     ;SPCR10
       STM    #4001h,SP0

;** AIC out of reset **
       CALL   XSR_EMPTY
       SSBX   XF
       STM    #0181h,DXR10   ;CR-1
       CALL   XSR_EMPTY
       RSBX   XF

;** enable DMA Ch2 interrupt, Clear IFR **
       STM    #0400h,IMR     ;DMA-INT CH2
       STM    #0FFFFh,IFR    ;Clr IFR

;** fir_isr setup code **
       LD     #FLAG1,DP      ;FLAG1,FLAG2 and COUNT on same DP
       ST     #0,FLAG1       ;assure FLAG1 (for in_bufs) is zero
       ST     #0,FLAG2       ;assure FLAG2 (for out_bufs) is zero
       ST     #0,COUNT       ;assure COUNT is zero
       STM    #31,BRC        ;generate 32 results
       STM    #96,BK         ;Moe, Larry, Curly input and
                             ;Tom, Dick, Harry output buffers
       STM    #1,AR0         ;emulate post inc by 1
       STM    #0Fh,DMSA      ;0Fh is DMSRC3 (for all DMSRC3 writes)
       RSBX   OVM            ;clear overflow mode
       SSBX    FRCT          ;set fractional mode
       SSBX   SXM            ;set sign extension
       RSBX   INTM           ;enable global interrupts last
```

DSP54.12 - 48

```
;************************
;**      Main Loop      **
;************************

main:
       IDLE   1       ;When DMA2 interrupts main, fir_isr runs and
       NOP            ;execution returns to this code. We then go
       NOP            ;back into IDLE mode and wait for the
       NOP            ;next interrupt.
       NOP
       B      main

;************************
;**    XSR Empty Test   **
;************************

XSR_EMPTY:
       LD     #0,DP
       BITF   @SP0,2h              ;poll XEMPTYn flag
       BC     XSR_EMPTY,TC
       RET
```

DSP54.12 - 49

```
;****************   FIR ISR (DMA Ch2 Int)   ****************

fir_isr:
       LD     #FLAG1,DP      ;Ignore First DMA Interrupt
       CMPM   @FLAG1,#0Fh    ;
       ST     #0Fh,@FLAG1    ;
       BC     done,NTC       ;
       ADDM   #1,@COUNT      ;COUNT holds 1,2,3 for Moe,Larry,Curly
       CMPM   @COUNT,#1      ;1st pass (Moe)?
       BC     loopinit,TC    ;if so, setup ARs and set output SRC
       CMPM   @COUNT,#2      ;2nd pass (Larry)?
       BC     test3,NTC      ;NO, go to test3
       STM    #y+32,DMSDN    ;DMA3 SRC = out_buf #2
       B      math

test3: CMPM   @COUNT,#3      ;3rd pass (Curly)?
       BC     fourth,NTC     ;NO, must be fourth, reset counter
       STM    #y+64,DMSDN    ;DMA3 SRC = out_buf #3
       B      math

fourth: ST    #1,COUNT       ;reset COUNT, reload ARs as 1st pass

loopinit:
       STM    #x,AR3         ;setup ARs for MAC
       STM    #y,AR4
       STM    #y,DMSDN       ;DMA3 SRC = out_buf #1

math:  ;...
```

DSP54.12 - 50

```
math:  STM    #a,AR2                ;always re-init coeff pointer
       RPTB   tstflg2-1
        MPY   *AR2+,*AR3+0%,A       ;1st product, AR3 circles on 96
        RPT   #14                   ;mult/acc 15 terms
         MAC  *AR2+,*AR3+0%,A
        MAR   *+AR3(-15)%           ;modify AR3 by -15 circularly
        STH   A,*AR4+               ;store result

tstflg2:
       CMPM   @FLAG2,#0Fh           ;Write dummy DXR to initiate
       ST     #0Fh,@FLAG2           ;first DMA3 transfer IF the FIRST
       XC     2,NTC                 ;out_buf is ready
       STM    #0,DXR10

done:  RETE                         ;return with enable
```

Now that we've written all of the code,
how does it get loaded into the system?

DSP54.12 - 51

The Bootloader

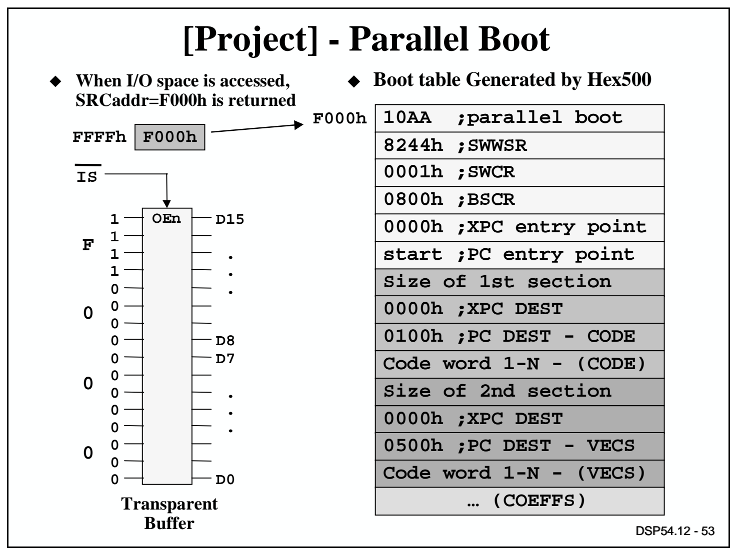
'5402 Boot Loader - Options

Boot Mode	Description	Trigger
No Boot	MP/MC=1, begin execution @RS vector	None
HPI-8	Host transfers code to DARAM. PC = dest.	INT2n low
Parallel	Boot Loader xfrs code. Src = 8/16-bit async mem dest = int/ext'l RAM. PC = entry point specified.	FFFFh in I/O = src src = 8AA or 10AAh
I/O	Boot Loader transfers code via I/O addr 0h. Handshake via XF/BIO.	FFFFh in data = src src = 8AA or 10AAh
Serial	Boot Loader configures SP and reads 1st word Src = 8/16-bit. Dest = int/ext'l RAM.	8AA or 10AAh rcvd?
Serial EEPROM	Bootloader configures SP in SPI-mode	INT3n low

Each 54xx device has specific options and modes. Refer to the boot-loader specification for your chosen device for more details.

'5402 bootloader can copy to extended program space

DSP54.12 - 52



HEX500

[Project] - Using HEX500

`HEX500 firmware.cmd`

```
                   /* FIRMWARE.CMD */
project.out          /* input file                    */
-e start             /* set entry (execution) point   */
-i                   /* select Intel format           */
-map project.mxp     /* map file for HEX500           */
-o project.hex       /* output file                   */
-memwidth 16         /* DSP accesses mem as 8/16-bit  */
-romwidth 16         /* physical mem width, 8/16-bit  */
-boot                /* make all sections bootable    */
-bootorg 0xF000      /* location of boot table        */
```

- **Must assemble .OUT file using -v548 if using ‘548/9/02/10 devices**
- **Programming formats: 16-bit ASCII hex, Tektronix, Intel MCS-86, Motorola S (16/24/32-bit addresses), 16-bit TI-Tag**
- **For more info, see the Assembly Language Tools User Guide**

Now, let’s look at our power-down options...

DSP54.12 - 54

IDLE

IDLE Options

IDLE n n =	1	2	3
Effect :			
CPU Halted	x	x	x
Peripherals Halted, CLKOUT Stopped		x	x
System Clock Stopped			x
Resume on:			
Reset	x	x	x
External Interrupt	x	x	x
Internal Interrupt	x		

- **A 10ns minimum pulse on any external interrupt pin will initiate the wakeup sequence**
- **INTM=0 : go to ISR, INTM=1 : continue in-line main code**
- **PLL requires a locking time for restart**
- **McBSP, DMA and HPI are considered ‘external’ devices in above table**

What other methods can be used to reduce power?

DSP54.12 - 55

Power Management Hints

Power Management Hints

Some design techniques for minimizing power consumption :

- Minimize external trace lengths and their associated capacitance
- Set Address Visibility (AVIS) = 0
- When not being used, make sure the timer and serial ports are in reset and MCM = 0
- Assure all input pins are grounded or pulled high
- Set SWWSR to 0 wait states when possible
- Use circular addressing instead of DELAY's
- Use internal instead of external memory accesses
- Minimize the clock frequency to match the task required
- Implement power down modes where possible

DSP54.12 - 56

BIOS and RTA

BIOS and RTA

- BIOS and CCS offer static configuration of most of the peripherals. We set up each peripheral manually via code, but BIOS will allow the user to control each peripheral from CCS.
- TI Analog group is creating plug-in tools to CCS to configure ADC/DAC devices
- RTA (real time analysis) allows the user to ensure that the real-time constraints are met. In our case, that means the in_buff/processing/out_buf timing works within spec.

DSP54.12 - 57

Need More Information?

Analog Information & Product Resources

- For the [Project], we used the following sources to select our conversion and power devices:
  - www.ti.com/sc/select
  - www.ti.com/sc/docs/products/msp/index.htm
- Evaluation modules can be found at:
  - www.ti.com/sc/docs/tools/analog/index.html
- Samples can be ordered on the TI Web
- Mixed-Signal and Analog Literature
  - Designers Guide (SLYU001B)
  - Analog Overview (SSDV004D)

DSP54.12 - 58

Don't Forget!

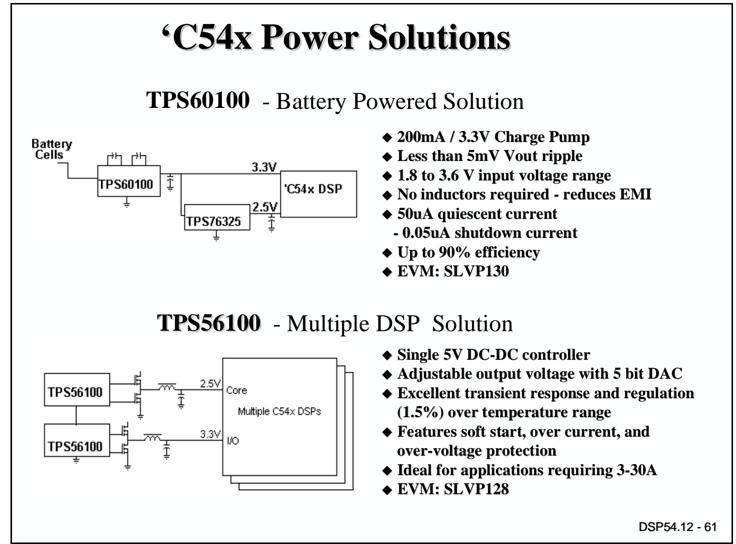
- Take your workbook, databook(s) and CDs with you
- Fill out the evaluation form (in pencil)
- Fill out and mail the registration form if you desire
- Copy c:\dsp54\labs and c:\dsp54\solutions to the disk in your workbook

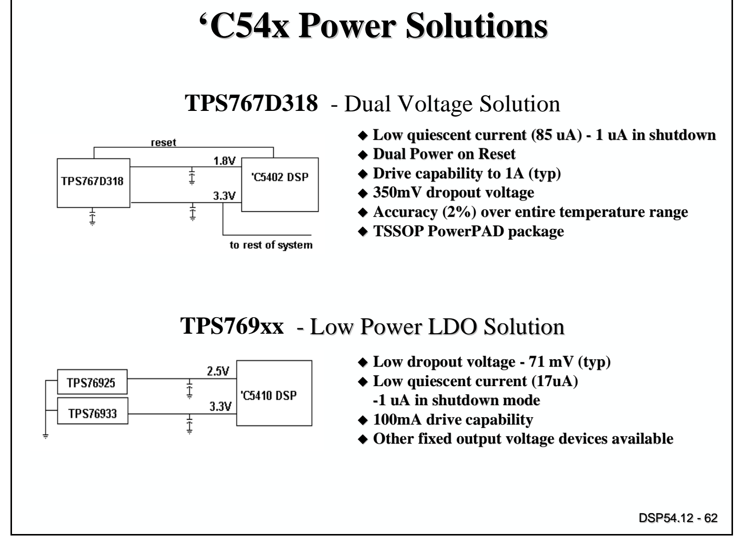
Thank you for attending.

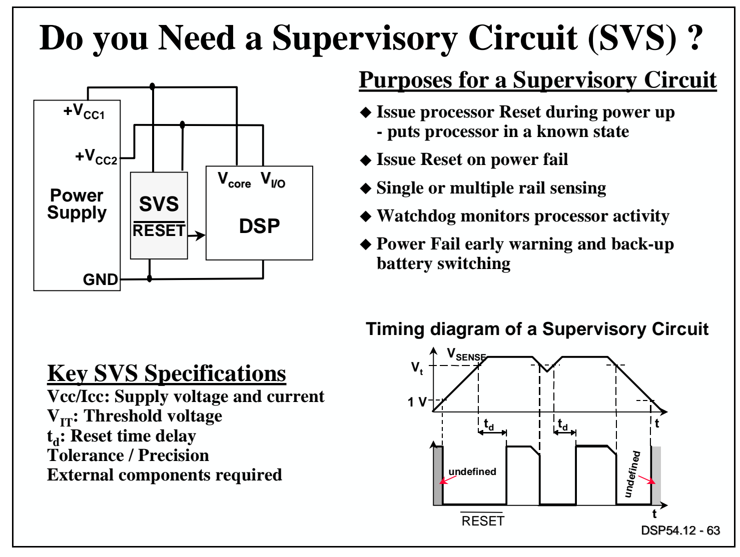
Have a safe trip home.

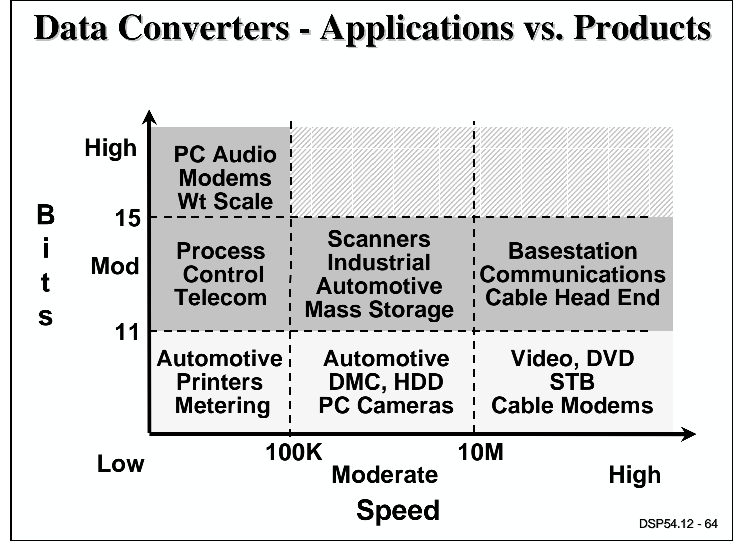
DSP54.12 - 59

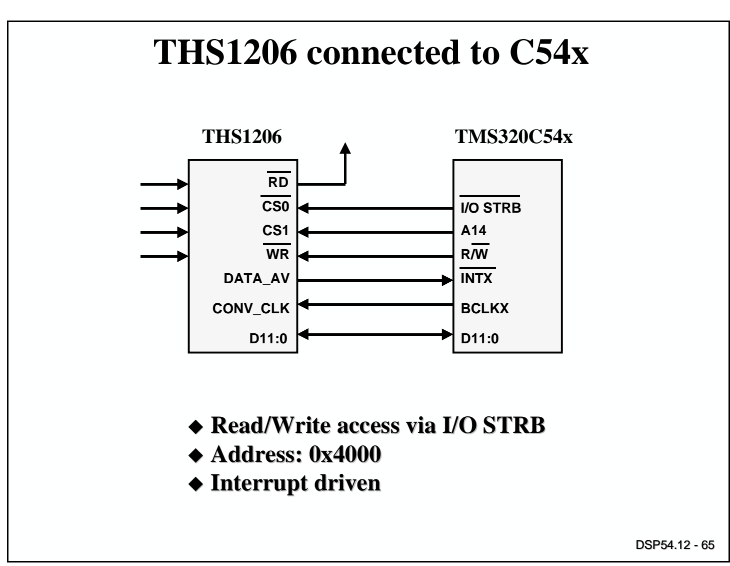
Additional Analog Information

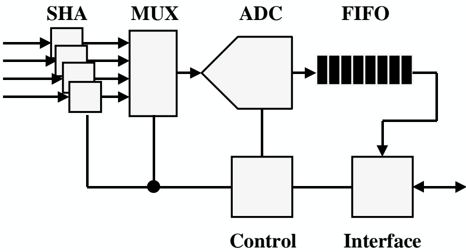





Do you Need a Supervisory Circuit (SVS) ?
+VCC1
+VCC2
Power Supply
GND
SVS
RESET
Vcore VI/O
DSP
Purposes for a Supervisory Circuit
◆ Issue processor Reset during power up - puts processor in a known state
◆ Issue Reset on power fail
◆ Single or multiple rail sensing
◆ Watchdog monitors processor activity
◆ Power Fail early warning and back-up battery switching
Key SVS Specifications
Vcc/Icc: Supply voltage and current
VIT: Threshold voltage
td: Reset time delay
Tolerance / Precision
External components required
Timing diagram of a Supervisory Circuit
VSENSE
Vt
1 V
td
td
t
undefined
undefined
RESET
t
DSP54.12 - 63


Data Converters - Applications vs. Products
Bits
High
15
Mod
11
Low
PC Audio Modems Wt Scale
Process Control Telecom
Scanners Industrial Automotive Mass Storage
Basestation Communications Cable Head End
Automotive Printers Metering
Automotive DMC, HDD PC Cameras
Video, DVD STB Cable Modems
100K
10M
Moderate
High
Speed
DSP54.12 - 64


THS1206 connected to C54x
THS1206
TMS320C54x
RD
CS0
CS1
WR
DATA_AV
CONV_CLK
D11:0
I/O STRB
A14
R/W
INTX
BCLKX
D11:0
◆ Read/Write access via I/O STRB
◆ Address: 0x4000
◆ Interrupt driven
DSP54.12 - 65

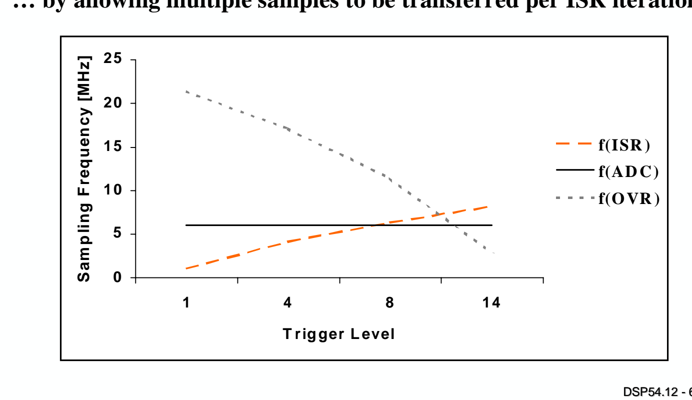


- **12-Bit, 6MSPS**
- **parallel interface**
- **FIFO to improve throughput**

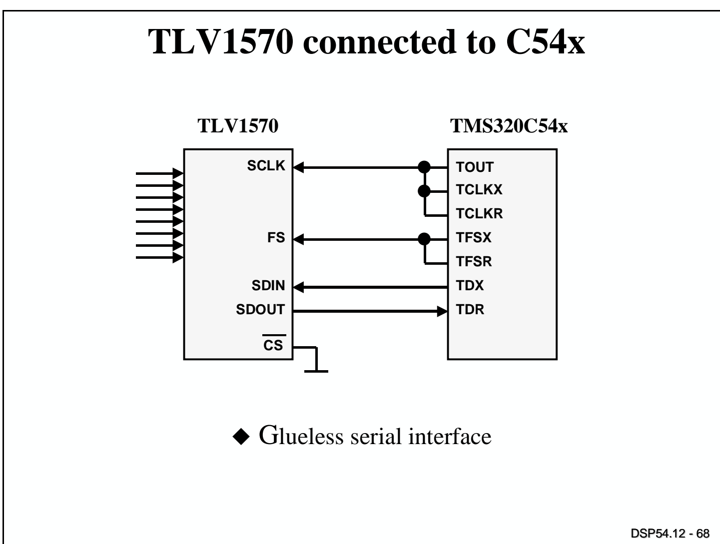
DSP54.12 - 66

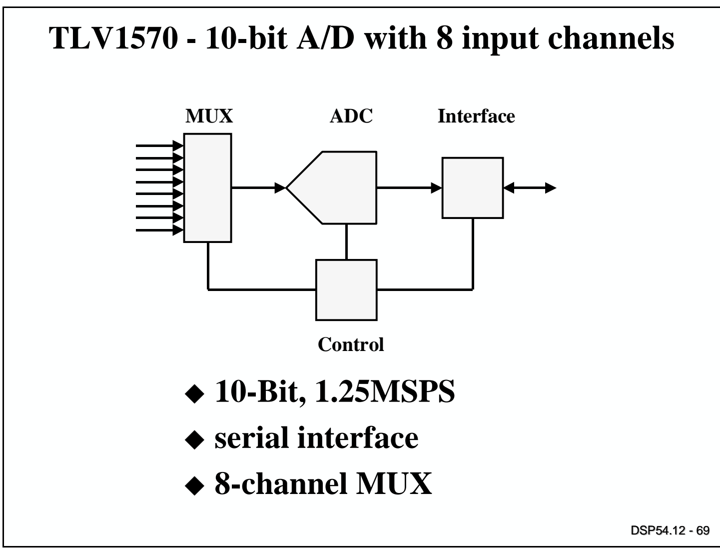
FIFO optimizes Interrupt Transfer ...

… by allowing multiple samples to be transferred per ISR iteration



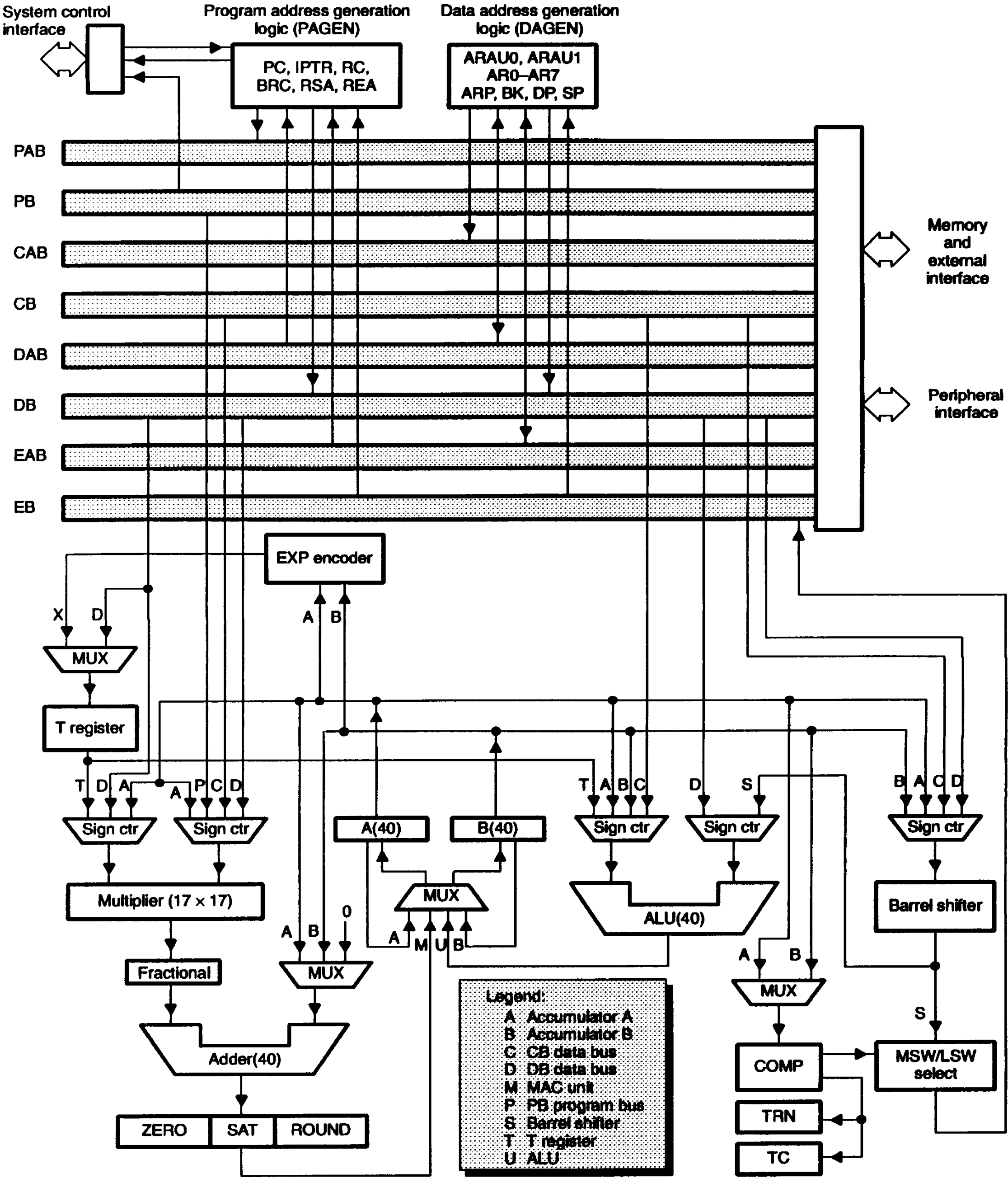
DSP54.12 - 67




TLV1570 - 10-bit A/D with 8 input channels
MUX
ADC
Interface
Control
◆ 10-Bit, 1.25MSPS
◆ serial interface
◆ 8-channel MUX
DSP54.12 - 69

Appendix

Block diagram



‘C54x Mnemonic Instruction Set Quick Reference

Arithmetic Operations

Add Instructions

Syntax	Expression	WC	
ADD Smem, src	src = src + Smem	1	1
ADD Smem, TS, src	src = src + Smem << TS	1	1
ADD Smem, 16, src [, dst]	dst = src + Smem << 16	1	1
ADD Smem [, SHIFT], src [, dst]	dst = src + Smem << SHIFT	2	2
ADD Xmem, SHFT, src	src = src + Xmem <<SHFT	1	1
ADD Xmem, Ymem, dst	dst = Xmem<<16 + Ymem<<16	1	1
ADD # lk [, SHFT], src [, dst]	dst = src + #lk << SHFT	2	2
ADD # lk, 16, src [, dst]	dst = src + #lk << 16	2	2
ADD src [, SHIFT] [, dst]	dst = dst + src << SHIFT	1	1
ADD src, ASM [, dst]	dst = dst + src << ASM	1	1
ADDC Smem, src	src = src + Smem + C	1	1
ADDM # lk, Smem	Smem = Smem + #lk	2	2
ADDS Smem, src	src = src + uns(Smem)	1	1

Subtract Instructions

Syntax	Expression	WC	
SUB Smem, src	src = src – Smem	1	1
SUB Smem, TS, src	src = src – Smem << TS	1	1
SUB Smem, 16, src [, dst]	dst = src – Smem << 16	1	1
SUB Smem [, SHIFT], src [, dst]	dst = src – Smem << SHIFT	2	2
SUB Xmem, SHFT, src	src = src – Xmem << SHFT	1	1
SUB Xmem, Ymem, dst	dst = Xmem<<16 – Ymem<<16	1	1
SUB # lk [, SHFT], src [, dst]	dst = src – #lk << SHFT	2	2
SUB # lk, 16, src [, dst]	dst = src – #lk <<16	2	2
SUB src[, SHIFT] [, dst]	dst = dst – src << SHIFT	1	1
SUB src, ASM [, dst]	dst = dst – src << ASM	1	1
SUBB Smem, src	src = src – Smem – C	1	1
SUBC Smem, src	If (src – Smem << 15) _ 0	1	1
	src = (src – Smem << 15) << 1 + 1		
	Else		
	src = src << 1		
SUBS Smem, src	src = src – uns(Smem)	1	1

Muliply Instructions

Syntax	Expression	WC	
MPY Smem, dst	dst = T * Smem	1	1
MPYR Smem, dst	dst = rnd(T * Smem)	1	1
MPY Xmem, Ymem, dst	dst = Xmem * Ymem,	1	1
	T = Xmem		
MPY Smem, # lk, dst	dst = Smem * #lk ,	2	2
	T = Smem		
MPY # lk, dst	dst = T * #lk	2	2
MPYA dst	dst = T * A(32–16)	1	1
MPYA Smem	B = Smem * A(32–16),	1	1
	T = Smem		
MPYU Smem, dst	dst = uns(T) * uns(Smem)	1	1
SQUR Smem, dst	dst = Smem * Smem,	1	1
	T = Smem		
SQUR A, dst	dst = A(32–16) * A(32–16)	1	1

Mult-Acc & Mult-Sub Instructions

Syntax	Expression	WC	
MAC Smem, src	src = src + T * Smem	1	1
MAC Xmem, Ymem, src [.dst]	dst = src + Xmem * Ymem,	1	1
	T = Xmem		
MAC # lk, src [, dst]	dst = src + T * #lk	2	2
MAC Smem, # lk, src [,dst]	dst = src + Smem * #lk,	2	2
	T = Smem		
MACR Smem, src	dst = rnd(src + T * Smem)	1	1
MACR Xmem, Ymem, src [,dst]	dst = rnd(src + Xmem * Ymem),	1	1
	T = Xmem		
MACA Smem [, B]	B = B + Smem * A(32–16),	1	1
	T = Smem		
MACA T, src [, dst]	dst = src + T * A(32–16)	1	1
MACAR Smem [, B]	B = rnd(B + Smem * A(32–16)),	1	1
	T = Smem		
MACAR T, src [, dst]	dst = rnd(src + T * A(32–16))	1	1
MACD Smem, pmad, src	src = src + Smem * pmad,	2	3
	T = Smem, (Smem + 1) = Smem		
MACP Smem, pmad, src	src = src + Smem * pmad,	2	3
	T = Smem		
MACSU Xmem, Ymem, src	src = src + uns(Xmem) * Ymem,	1	1
	T = Xmem		
MAS Smem, src	src = src – T * Smem	1	1

Mult-Acc & Mult-Sub (cont.)

Syntax	Expression	WC
MASR Xmem, Ymem, src [,dst]	dst = rnd(src – Xmem * Ymem), T = Xmem	1 1
MAS Xmem, Ymem, src [,dst]	dst = src – Xmem * Ymem, T = Xmem	1 1
MASR Smem, src	src = rnd(src – T * Smem)	1 1
MASA Smem [, B]	B = B – Smem * A(32–16), T = Smem	1 1
MASA T, src [, dst]	dst = src – T * A(32–16)	1 1
MASAR T, src [, dst]	dst = rnd(src – T * A(32–16))	1 1
SQURA Smem, src	src = src + Smem * Smem, T = Smem	1 1
SQURS Smem, src	src = src – Smem * Smem, T = Smem	1 1

Double (32-bit) Operand Instructions

Syntax	Expression	WC
DADD Lmem, src [, dst]	If C16 = 0 dst = Lmem + src If C16 = 1 dst(39–16) = Lmem(31–16) + src(31–16) dst(15–0) = Lmem(15–0) + src(15–0)	1 1
DADST Lmem, dst	If C16 = 0 dst = Lmem + (T << 16 + T) If C16 = 1 dst(39–16) = Lmem(31–16) + T dst(15–0) = Lmem(15–0) – T	1 1
DRSUB Lmem, src	If C16 = 0 src = Lmem – src If C16 = 1 src(39–16) = Lmem(31–16) – src(31–16) src(15–0) = Lmem(15–0) – src(15–0)	1 1
DSADT Lmem, dst	If C16 = 0 dst = Lmem – (T << 16 + T) If C16 = 1 dst(39–16) = Lmem(31–16) – T dst(15–0) = Lmem(15–0) + T	1 1
DSUB Lmem, src	If C16 = 0 src = src – Lmem If C16 = 1 src (39–16) = src(31–16) – Lmem(31–16) src (15–0) = src(15–0) – Lmem(15–0)	1 1
DSUBT Lmem, dst	If C16 = 0 dst = Lmem – (T << 16 + T) If C16 = 1 dst(39–16) = Lmem(31–16) – T dst(15–0) = Lmem(15–0) – T	1 1

Application-Specific Instructions

Syntax	Expression	WC
ABDST Xmem, Ymem	B = B + |A(32–16)| A = (Xmem – Ymem) << 16	1 1
ABS src [, dst]	dst = |src|	1 1
CMPL src [, dst]	dst = ~src	1 1
DELAY Smem	(Smem + 1) = Smem	1 1
EXP src	T = number of sign bits (src) – 8	1 1
FIRS Xmem, Ymem, pmad	B = B + A * pmad A = (Xmem + Ymem) << 16	2 3
LMS Xmem, Ymem	B = B + Xmem * Ymem A = (A + Xmem << 16) + 2 15	1 1
MAX dst	dst = max(A, B)	1 1
MIN dst	dst = min(A, B)	1 1
NEG src [, dst]	dst = –src	1 1
NORM src [, dst]	dst = src << TS dst = norm(src, TS)	1 1
POLY Smem	B = Smem << 16 A = rnd(A * T + B)	1 1
RND src [, dst]	dst = src + 2^15	1 1
SAT src	saturate(src)	1 1
SQDST Xmem, Ymem	B = B + A(32–16) * A(32–16) A = (Xmem + Ymem) << 16	1 1

Logical Operations

AND Instructions

Syntax	Expression	WC
AND Smem, src	src = src & Smem	1 1
AND # lk [, SHFT], src [, dst]	dst = src & #lk << SHFT	2 2
AND # lk, 16, src [, dst]	dst = src & #lk << 16	2 2
AND src [, SHIFT] [, dst]	dst = dst & src << SHIFT	1 1
ANDM # lk, Smem	Smem = Smem & #lk	2 2

OR Instructions

Syntax	Expression	WC
OR Smem, src	src = src | Smem	1 1
OR # lk [, SHFT], src [, dst]	dst = src | #lk << SHFT	2 2
OR # lk, 16, src [, dst]	dst = src | #lk << 16	2 2
OR src [, SHIFT] [, dst]	dst = dst | src << SHIFT	1 1
ORM # lk, Smem	Smem = Smem | #lk	2 2

XOR Instructions

Syntax	Expression	WC
XOR Smem, src	src = src ^ Smem	1 1
XOR # lk [, SHFT,], src [, dst]	dst = src ^ #lk << SHFT	2 2
XOR # lk, 16, src [, dst]	dst = src ^ #lk << 16	2 2
XOR src [, SHIFT] [, dst]	dst = dst ^ src << SHIFT	1 1
XORM # lk, Smem	Smem = Smem ^ #lk	2 2

Shift Instructions

Syntax	Expression	WC
ROL src	Rotate left with carry in	1 1
ROLTC src	Rotate left with TC in	1 1
ROR src	Rotate right with carry in	1 1
SFTA src, SHIFT [, dst]	dst = src << SHIFT {arith. shift}	1 1
SFTC src	if src(31) = src(30) then src = src<<1	1 1
SFTL src, SHIFT [, dst]	dst = src << SHIFT {logical}	1 1

Test Instructions

Syntax	Expression	WC
BIT Xmem, BITC	TC = Xmem(15 – BITC)	1 1
BITF Smem, # lk	TC = (Smem && #lk)	2 2
BITT Smem	TC = Smem(15 – T(3–0))	1 1
CMPM Smem, # lk	TC = (Smem == #lk)	2 2
CMPR CC, ARx	Compare ARx with AR0	1 1

Program Control Operations

Branch Instructions

Syntax	Expression	WC
B[D] pmad	PC = pmad(15–0)	2 4/[2]
BACC[D] src	PC = src(15–0)	1 6/[4]
BANZ[D] pmad, Sind 4/2/[2]	if (Sind _ 0) then PC = pmad(15–0)	2
BC[D] pmad,cond[,cond[,cond]] 5/3/[3]	if (cond(s)) then PC = pmad(15–0)	2
FB[D] extpmad	PC= pmad(15–0), XPC = pmad(22–16)	2 4/[2]
FBACC[D] src	PC = src(15–0), XPC = src(22–16)	1 6/[4]

Call Instructions

Syntax	Expression	WC
CALA[D] src	– –SP = PC, PC = src(15–0)	1 6/[4]
CALL[D] pmad	– –SP = PC, PC = pmad(15–0)	2 4/[2]
CC[D] pmad,cond[,cond[,cond]]	if (cond(s)) then – –SP = PC, PC = pmad(15–0)	2 5/5/[3]
FCALA[D] src	– –SP = PC, – –SP = XPC, PC = src(15–0), XPC = src(22–16)	1 6/[4]
FCALL[D] extpmad	– –SP = PC, – –SP = XPC, PC = pmad(15–0), XPC = pmad(22–16)	2 4/[2]

Interrupt Instructions

Syntax	Expression	WC
INTR K	– –SP = PC, PC = IPTR(15–7) + K << 2, INTM = 1	1 3
TRAP K	– –SP = PC, PC = IPTR(15–7) + K << 2	1 3

Return Instructions

Syntax	Expression	WC
FRET[D]	XPC = SP++, PC = SP++	1 6/[4]
FRETE[D]	XPC = SP++, PC = SP++, INTM = 0	1 6/[4]
RC[D] cond[,cond[,cond]]	if (cond(s)) then PC = SP++	1 5/3/[3]
RET[D]	PC = SP++	1 5/[3]
RETE[D]	PC = SP++, INTM = 0	1 5/[3]
RETF[D]	PC = RTN, PC++, INTM = 0	1 3/[1]

Repeat Instructions

Syntax	Expression	WC
RPT Smem	Repeat single, RC = Smem	1 1
RPT # K	Repeat single, RC = #K	1 1
RPT # lk	Repeat single, RC = #lk	2 2
RPTB[D] pmad	Repeat block, RSA = PC + 2[4], REA = pmad – 1	2 4/[2]
RPTZ dst, # lk	Repeat single, RC = #lk, dst = 0	2 2

Stack-Manipulating Instructions

Syntax	Expression	WC
FRAME K	SP = SP + K	1 1
POPD Smem	Smem = SP++	1 1
POPM MMR	MMR = SP++	1 1
PSHD Smem	– –SP = Smem	1 1
PSHM MMR	– –SP = MMR	1 1

Misc. Program Control Instructions

Syntax	Expression	WC
IDLE K	idle(K)	1 4
MAR Smem	If CMPT = 0, then modify ARx If CMPT = 1 and ARx _ AR0, then modify ARx, ARP = x If CMPT = 1 and ARx = AR0, then modify AR(ARP)	1 1
NOP	no operation	1 1
RESET	software reset	1 3
RSBX N, SBIT	STN (SBIT) = 0	1 1
SSBX N, SBIT	STN (SBIT) = 1	1 1
XC n , cond [, cond[, cond]]	If (cond(s)) then execute the next n instructions; n = 1 or 2	1 1

Load and Store Operations

Load Instructions

Syntax	Expression	WC
DLD Lmem, dst	dst = Lmem	1 1
LD Smem, dst	dst = Smem	1 1
LD Smem, TS, dst	dst = Smem << TS	1 1
LD Smem, 16 , dst	dst = Smem << 16	1 1
LD Smem [, SHIFT], dst	dst = Smem << SHIFT	2 2
LD Xmem, SHFT, dst	dst = Xmem << SHFT	1 1
LD # K, dst	dst = #K	1 1
LD # lk [, SHFT], dst	dst = #lk << SHFT	2 2
LD # lk, 16, dst	dst = #lk << 16	2 2
LD src, ASM [, dst]	dst = src << ASM	1 1
LD src [, SHIFT] [, dst]	dst = src << SHIFT	1 1
LD Smem, T	T = Smem	1 1
LD Smem, DP	DP = Smem(8–0)	1 3
LD # k9, DP	DP = #k9	1 1
LD # k5, ASM	ASM = #k5	1 1
LD # k3, ARP	ARP = #k3	1 1
LD Smem, ASM	ASM = Smem(4–0)	1 1
LDM MMR, dst	dst = MMR	1 1
LDR Smem, dst	dst = rnd(Smem)	1 1
LDU Smem, dst	dst = uns(Smem)	1 1
LTD Smem	T = Smem, (Smem + 1) = Smem	1 1

Store Instructions

Syntax	Expression	WC
DST src, Lmem	Lmem = src	1 2
ST T, Smem	Smem = T	1 1
ST TRN, Smem	Smem = TRN	1 1
ST # lk, Smem	Smem = #lk	2 2
STH src, Smem	Smem = src << –16	1 1
STH src, ASM, Smem	Smem = src << (ASM – 16)	1 1
STH src, SHFT, Xmem	Xmem = src << (SHFT – 16)	1 1
STH src [, SHIFT], Smem	Smem = src << (SHIFT – 16)	2 2
STL src, Smem	Smem = src	1 1
STL src, ASM, Smem	Smem = src << ASM	1 1
STL src, SHFT, Xmem	Xmem = src << SHFT	1 1
STL src [, SHIFT], Smem	Smem = src << SHIFT	2 2
STLM src, MMR	MMR = src	1 1
STM # lk, MMR	MMR = #lk	2 2

Conditional Store Instructions

Syntax	Expression	WC
CMPS src, Smem	If src(31–16) > src(15–0) then Smem = src(31–16) If src(31–16) _ src(15–0) then Smem = src(15–0)	1 1
SACCD src, Xmem, cond	If (cond) Xmem = src<<(ASM–16)	1 1
SRCCD Xmem, cond	If (cond) Xmem = BRC	1 1
STRCD Xmem, cond	If (cond) Xmem = T	1 1

Parallel Load and Mult. Instructions

<table>
<tr><th>Syntax</th><th>Expression</th><th>WC</th></tr>
<tr><td>LD Xmem, dst</td><td>dst = Xmem << 16</td><td>1 1</td></tr>
<tr><td>|| MAC Ymem, [dst_]</td><td>|| dst_ = dst_ + T * Ymem</td><td></td></tr>
<tr><td>LD Xmem, dst</td><td>dst = Xmem << 16</td><td>1 1</td></tr>
<tr><td>|| MACR Ymem, [dst_]</td><td>|| dst_ = rnd(dst_ + T * Ymem)</td><td></td></tr>
<tr><td>LD Xmem, dst</td><td>dst = Xmem << 16</td><td>1 1</td></tr>
<tr><td>|| MAS Ymem, [dst_]</td><td>|| dst_ = dst_ – T * Ymem</td><td></td></tr>
<tr><td>LD Xmem, dst</td><td>dst = Xmem << 16</td><td>1 1</td></tr>
<tr><td>|| MASR Ymem, [dst_]</td><td>|| dst_ = rnd(dst_ – T * Ymem)</td><td></td></tr>
</table>

Parallel Load and Store Instructions

<table>
<tr><th>Syntax</th><th>Expression</th><th>WC</th></tr>
<tr><td>ST src, Ymem</td><td>Ymem = src << (ASM - 16)</td><td>1 1</td></tr>
<tr><td>|| LD Xmem, dst</td><td>|| dst = Xmem << 16</td><td></td></tr>
<tr><td>ST src, Ymem</td><td>Ymem = src << (ASM - 16)</td><td>1 1</td></tr>
<tr><td>|| LD Xmem, T</td><td>|| T = Xmem</td><td></td></tr>
</table>

Parallel Store and Mult Instructions

<table>
<tr><th>Syntax</th><th>Expression</th><th>WC</th></tr>
<tr><td>ST src, Ymem</td><td>Ymem = src << (ASM – 16)</td><td>1 1</td></tr>
<tr><td>|| MAC Xmem, dst</td><td>|| dst = dst + T * Xmem</td><td></td></tr>
<tr><td>ST src, Ymem</td><td>Ymem = src << (ASM – 16)</td><td>1 1</td></tr>
<tr><td>|| MACR Xmem, dst</td><td>|| dst = rnd(dst + T * Xmem)</td><td></td></tr>
<tr><td>ST src, Ymem</td><td>Ymem = src << (ASM – 16)</td><td>1 1</td></tr>
<tr><td>|| MAS Xmem, dst</td><td>|| dst = dst – T * Xmem</td><td></td></tr>
<tr><td>ST src, Ymem</td><td>Ymem = src << (ASM – 16)</td><td>1 1</td></tr>
<tr><td>|| MASR Xmem, dst</td><td>|| dst = rnd(dst – T * Xmem)</td><td></td></tr>
<tr><td>ST src, Ymem</td><td>Ymem = src << (ASM – 16)</td><td>1 1</td></tr>
<tr><td>|| MPY Xmem, dst</td><td>|| dst = T * Xmem</td><td></td></tr>
</table>

Parallel Store & Add/Sub Instructions

<table>
<tr><th>Syntax</th><th>Expression</th><th>WC</th></tr>
<tr><td>ST src, Ymem</td><td>Ymem = src << (ASM -16)</td><td>1 1</td></tr>
<tr><td>|| ADD Xmem, dst</td><td>|| dst = dst_ + Xmem <<16</td><td></td></tr>
<tr><td>ST src, Ymem</td><td>Ymem = src << (ASM – 16)</td><td>1 1</td></tr>
<tr><td>|| SUB Xmem, dst</td><td>|| dst = (Xmem << 16) – dst_</td><td></td></tr>
</table>

Misc Load & Store Type Instructions

Syntax	Expression	WC
MVDD Xmem, Ymem	Ymem = Xmem	1 1
MVDK Smem, dmad	dmad = Smem	2 2
MVDM dmad, MMR	MMR = dmad	2 2
MVDP Smem, pmad	pmad = Smem	2 4
MVKD dmad, Smem	Smem = dmad	2 2
MVMD MMR, dmad	dmad = MMR	2 2
MVMM MMRx, MMRy	MMRy = MMRx	1 1
MVPD pmad, Smem	Smem = pmad	2 3
PORTR PA, Smem	Smem = PA	2 2
PORTW Smem, PA	PA = Smem	2 2
READA Smem	Smem = A	1 5
WRITA Smem	A = Smem	1 5

Indirect Addressing Types With a Single Data-Memory Operand

*ARx	*ARx-%
*ARx-	*ARx-0%
*ARx+	*ARx+%
*+ARx	*ARx+0%
*ARx-0	*+ARx(lk)
*ARx+0	*ARx(lk)%
*ARx+0B	*(lk)

Indirect Addressing Types With a Dual Data-Memory Operand

*ARx	*ARx-%
*ARx-	*ARx-0%

Conditions for Conditional Instructions

Operand	Condition	Description
AEQ	A = 0	Accumulator A equal to 0
BEQ	B = 0	Accumulator B equal to 0
ANEQ	A <> 0	A not equal to 0
BNEQ	B<>0	Accumulator B not equal to 0
ALT	A < 0	Accumulator A less than 0
BLT	B < 0	Accumulator B less than 0
ALEQ	A =< 0	Accumulator A less than or equal to 0
BLEQ	B =< 0	Accumulator B less than or equal to 0
AGT	A > 0	Accumulator A greater than 0
BGT	B > 0	Accumulator B greater than 0
AGEQ	A >= 0	Accumulator A greater than or equal to 0
BGEQ	B >=0	Accumulator B greater than or equal to 0
AOV †	AOV = 1	Accumulator A overflow detected
BOV †	BOV = 1	Accumulator B overflow detected
ANOV †	AOV = 0	No accumulator A overflow detected
BNOV †	BOV = 0	No accumulator B overflow detected
C †	C = 1	ALU carry set to 1
NC †	C = 0	ALU carry clear to 0
TC †	TC = 1	Test/Control flag set to 1
NTC †	TC = 0	Test/Control flag cleared to 0
BIO †	BIO low	BIO signal is low
NBIO †	BIO high	BIO signal is high
UNC †	none	Unconditional operation

† Cannot be used with conditional store instructions

Groupings of Conditions

Group1: You can select up to two conditions. Each of these conditions must be from a different category (category A or B); you cannot have two conditions from the same category. For example, you can test EQ and OV at the same time but you cannot test GT and NEQ at the same time.

Group 2: You can select up to three conditions. Each of these conditions must be from a different category (category A, B, or C); you can-not have two conditions from the same category. For example, you can test TC, C, and BIO at the same time but you cannot test NTC, C, and NC at the same time.

Group 1		Group 2		
A	B	A	B	C
EQ	OV	TC	C	BIO
NEQ	NOV	NTC	NC	NBIO
LT				
LEQ				
GT				
GEQ				

CPU Memory-Mapped Registers

Address	Name	Description
0	IMR	Interrupt mask register
1	IFR	Interrupt flag register
2–5	–	Reserved for testing
6	ST0	Status register 0
7	ST1	Status register 1
8	AL	Accumulator A low word (bits 15–0)
9	AH	Accumulator A high word (bits 31–16)
A	AG	Accumulator A guard bits (bits 39–32)
B	BL	Accumulator B low word (bits 15–0)
C	BH	Accumulator B high word (bits 31–16)
D	BG	Accumulator B guard bits (bits 39–32)
E	T	Temporary register
F	TRN	Transition register
10	AR0	Auxiliary register 0
11	AR1	Auxiliary register 1
12	AR2	Auxiliary register 2
13	AR3	Auxiliary register 3
14	AR4	Auxiliary register 4
15	AR5	Auxiliary register 5
16	AR6	Auxiliary register 6
17	AR7	Auxiliary register 7
18	SP	Stack pointer
19	BK	Circular-buffer size register
1A	BRC	Block-repeat counter
1B	RSA	Block-repeat start address
1C	REA	Block-repeat end address
1D	PMST	Processor mode status register
1E	XPC	Program counter extension register (’548/9)
1E–1F	–	Reserved

Processor Mode Status Register (PMST)

15-7	6	5	4	3	2	1	0
IPTR	MP/MC-	OVLY	AVIS	DROM	CLKOFF	SMUL*	SST*

* LP devices only; reserved on all other devices

Status Register 0 (ST0)

15-13	12	11	10	9	8-0
ARP	TC	C	OVA	OVB	DP

Status Register 1 (ST1)

15	14	13	12	11	10	9	8	7	6	5	4-0
BRAF	CPL	XF	HM	INTM	0	OVM	SXM	C16	FRCT	CMPT	ASM

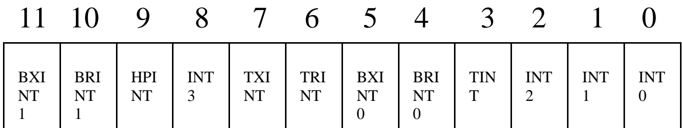
Interrupt Registers (IFR/IMR)

‘541

8	7	6	5	4	3	2	1	0
INT3	XINT1	RINT1	XINT0	RINT0	TINT	INT2	INT1	INT0

BITS 15-9 ARE RESERVED

‘548/9



11	10	9	8	7	6	5	4	3	2	1	0
BXINT1	BRINT1	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

BITS 15-12 ARE RESERVED

TMS320C54x Literature

If you prefer your databooks in electronic format, find them now at:

http://www.ti.com/sc/docs/dsps/hotline/support.htm

If you prefer paper databooks, order them at:

http://www.ti.com/sc/docs/feedbk1.htm

The next page is a partial list of literature available for the ‘C54x. Always refer to TI’s Web pages for the latest documentation and information.

User’s Manuals

TMS320 DSP DEVELOPMENT SUPPORT REFERENCE GUIDE	spru011e
TMS320C54X ASSEMBLY LANGUAGE TOOLS USER'S GUIDE	spru102b
TMS320C54X DSKPLUS DSP STARTER KIT USER'S GUIDE	spru191
TMS320C54X DSP ALGEBRAIC INSTRUCTION SET REFERENCE SET VOLUME 3	spru179a
TMS320C54X DSP APPLICATIONS GUIDE REFERENCE SET VOLUME 4	spru173
TMS320C54X DSP CPU AND PERIPHERAL REFERENCE SET VOLUME I	spru131d
TMS320C54X DSP MNEMONIC INSTRUCTION SET REFERENCE SET VOLUME 2	spru172b
TMS320C54X EVALUATION MODULE TECHNICAL REFERENCE	spru135
TMS320C54X OPTIMIZING C COMPILER USER'S GUIDE	spru103b
TMS320C54X SIMULATOR C SOURCE DEBUGGER USER'S GUIDE ADDENDUM	spru170
TMS320C5X SIMULATOR GETTING STARTED GUIDE	spru124c
TMS320C5XX C SOURCE DEBUGGER USER'S GUIDE	spru099a

Applications

A-LAW AND MU-LAW COMPANDING IMPLEMENTATIONS USING THE TMS320C54X	spra163a
ACCESSING TMS320C54X MEMORY-MAPPED REGISTERS IN C - C54XREGS.H	spra260
ACOUSTIC-ECHO CANCELLATION S/W FOR HANDS-FREE WIRELESS SYSTEMS	spra162
ADDRESSING PERIPHERALS AS DATA STRUCTURES IN C	spra226
C54X EXTENDED ADDRESSING	spra184
CALCULATION OF TMS320LC54X POWER DISSIPATION	spra164
DECT/CT2 BBSP SOFTWARE PACKAGE	bpra052
DESIGNING LOW-POWER APPLICATIONS WITH THE TMS320LC54X	spra281
DSP SOLUTIONS FOR TELEPHONY AND DATA/FACSIMILE MODEMS	spra073
DTMF TONE GENERATION AND DETECTION ON THE TMS320C54X	spra096
DUAL POWER SUPPLY MANAGEMENT FOR THE TMS320VC549 DSP	spra280
ECHO CANCELLATION S/W FOR TMS320C54X	bpra054
EMULATOR PROCESSOR ACCESS TIMEOUT	spra248
EXTENDING FIXED-POINT DYNAMIC RANGES	spra249
GUIDELINES FOR USING DECOUPLING CAPACITORS ON DSP DESIGNS	spra230
H/W CONSIDERATIONS WHEN DESIGNING AN INTERFACE USING THE TMS320C54X	spra151
IIR FILTER DESIGN ON THE TMS320C54X DSP APPLICATION REPORT	spra079
HIGH SPEED MODEM W/MULTILEVEL MULTIDIMENSIONAL MODULATION-TMS320C542	spra321
IMPROVED CONTEXT SAVE/RESTORE PERF. & INT. LATENCY FOR ISRs WRITTEN IN C	spra232
INITIALIZING THE FIXED-POINT EVM'S AIC	spra206
IS-54 DIGITAL CELLULAR PHONE: A FUNCTIONAL ANALYSIS	spra134
IS-54 SIMULATION	spra135
LINE ECHO CANCELLER	spra188
LINKING C DATA OBJECTS SEPARATE FROM THE .BSS SECTION	spra258
MU-LAW COMPRESSION ON THE TMS320C54X	spra267
MULTIPASS LINKING	spra257
PARITY GENERATION ON THE TMS320C54X	spra266
PC/TMS320C54X EVALUATION MODULE COMMUNICATION INTERFACE	bpra051
REDUCING SYSTEM POWER REQUIREMENTS	spra209
SERIAL ROM BOOT	spra233
SHARING HEADER FILES IN C AND ASSEMBLY	spra205
THE IMPLEMENTATION OF G.726 ADPCM ON TMS320C54X DSP	bpra053
TMS320C54X DSP PROGRAMMING ENVIRONMENT	spra182
USING VRAMS AND DSPS FOR SYSTEM PERFORMANCE	spra224
VITERBI DECODING TECHNIQUES IN THE TMS320C54X FAMILY APPLICATION REPORT	spra071
TMS320C548/9 BOOT LOADER AND ON-CHIP ROM DESCRIPTION	
‘5X TO ‘54X CODE TRANSLATION	