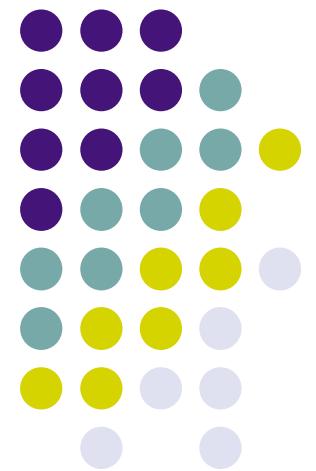


# Digital Signal Processing and Applications with the TMS320C6713 DSK

Day 2

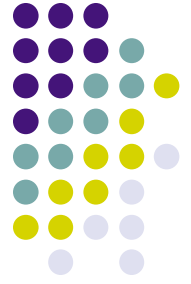
D. Richard Brown III  
Associate Professor  
Worcester Polytechnic Institute  
Electrical and Computer Engineering Department  
drb@ece.wpi.edu

October 15-16, 2007

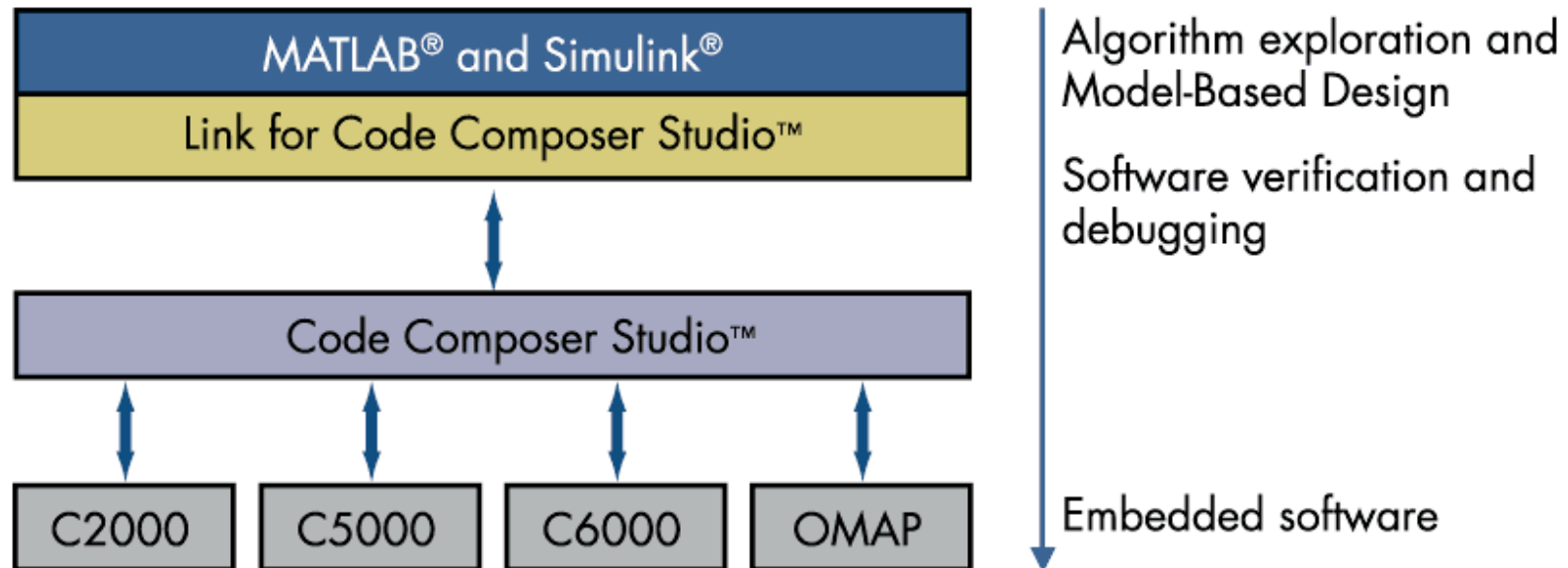


Technology for Innovators™

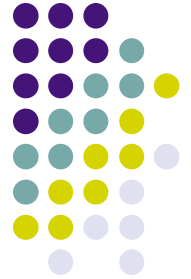
# Matlab's Link for Code Composer Studio



- Yesterday we used Matlab to design an FIR filter.
- The “Link for Code Composer Studio” toolbox allows even more direct Matlab-CCS integration.



# Getting Started with Matlab's Link for CCS



```
% make sure DSK is connected to the computer via USB  
% and is on before proceeding
```

```
% display a list of available DSP boards and processors  
ccsboardinfo
```

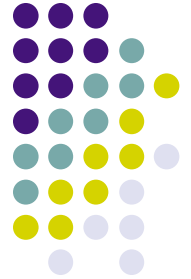
```
% create Matlab/CCS link for board=0, processor=0  
cc = ccstdsp('boardnum',0,'procnum',0);
```

```
% get information about the status and capabilities of the link  
display(cc)  
info(cc)  
isrunning(cc)  
isrtdxcapable(cc)  
getproc(cc)
```

```
% make CCS visible  
visible(cc,1)
```



# Opening, building, loading, and running a project



```
% open existing project
cd(cc, 'c:\myproject\helloworld');
open(cc, 'helloworld.pjt');

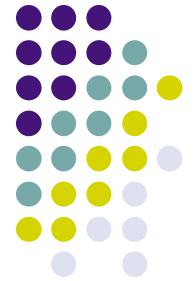
% build the project (returns a value of 1 if successful)
build(cc, 'all')

% load the binary file to the DSK
load(cc, 'Debug\helloworld.out')

% run the code on the DSK and check to see if it is running
restart(cc)
run(cc)
isrunning(cc)

% halt execution on the DSK and check to see if it stopped
halt(cc)
isrunning(cc)
```





# Reading/Writing DSK Variables

```
Uint32 temp;  
short foo[100];
```

*(variables declared in CCS)*

```
% Important: Do not attempt to read/write data from/to the DSK while it is running.  
% Insert one or more breakpoints in the code, run to the breakpoint,  
% perform the read, then resume execution
```

```
% confirm the DSK is not running  
isrunning(cc)
```

```
% create an object for the DSK variables temp and foo (can be global or local)  
tempobj = createobj(cc,'temp');  
foobj = createobj(cc,'foo');
```

```
% read/write examples
```

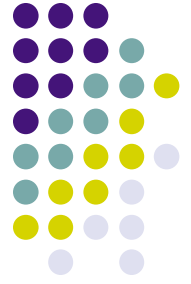
```
x = read(tempobj)  
write(tempobj,1234)  
y = read(cc,foobj)  
z = read(cc,foobj,10)  
write(foobj,4,999)
```

```
% DSK temp -> Matlab x  
% 1234 -> DSK temp  
% DSK foo -> Matlab y (whole array)  
% DSK foo -> Matlab y (10th element)  
% 999 -> DSK foo (4th element)
```



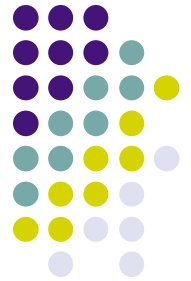
Technology for Innovators™

# Cool things that you can do with Matlab's Link for CCS



1. Rapid filter design: Halt execution on the DSK (**halt**), write new filter coefficients (**write**), resume execution (**restart/run**), and test your filter without rebuilding the project.
2. Use specific test signals: Generate a specific test signal in Matlab, overwrite the codec samples (**write**) with your test signal samples, run the processing code on the DSK, observe the output.
3. Rapid data analysis/graphing: Read the contents of the a filter output (**read**) to Matlab, analyze the spectrum or other properties, generate plots.

# Tip: Making Interesting Test Signals in Matlab



## Example: In-Phase and Quadrature Sinusoids

```
>> fs=44100;           % set up sampling frequency
>> t=0:1/fs:5;        % time vector (5 seconds long)
>> x=[sin(2*pi*1000*t) cos(2*pi*1000*t)]; % left = sin, right = cos
>> soundsc(x,fs);     % play sound through sound card
```

## Another example: white noise (in stereo)

```
>> L=length(t);
>> x=[randn(L,1) randn(L,1)];
>> soundsc(x,fs);     % play sound through sound card
```

These signals are all generated as double precision floats but can be cast to fixed point or integer formats if necessary.

You can save your signals to .wav files with Matlab's **wavwrite** function. These .wav files can be burned to CD and played with conventional stereo equipment.



TEXAS INSTRUMENTS

Technology for Innovators™



# Some Things to Try

Try out some of the basic functions of Matlab's Link for CCS

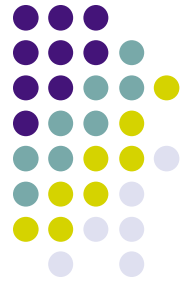
- Establish a link from Matlab to CCS
- Open your FIR filter project from yesterday.
- Build, load, run, and halt it.
- Set breakpoints.
- Read and write global/local variables.

Try rapid filter design:

- While your code is running, generate new filter coefficients in Matlab using FDA tool, e.g. try a bandstop filter
  - Halt code execution
  - Change the filter coefficients in the DSK memory (**write** function in Matlab)
  - Resume code execution
  - Test the frequency response to see if it changed

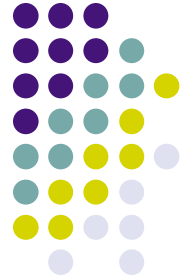


# Profiling Your Code and Making it More Efficient

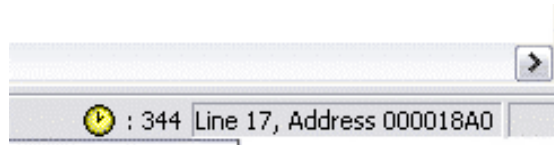


- How to estimate the **execution time** of your code.
- How to use the **optimizing compiler** to produce more efficient code.
- How **data types** and **memory usage** affect the efficiency of your code.

# How to estimate code execution time when connected to the DSK



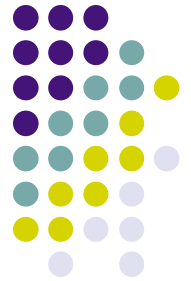
1. Start CCS with the C6713 DSK connected
2. Debug -> Connect (or alt+C)
3. Open project, build it, and load .out file to the DSK
4. Open the source file you wish to profile
5. Set two breakpoints for the start/end of the code range you wish to profile
6. Profile -> Clock -> Enable
7. Profile -> Clock -> View
8. Run to the first breakpoint
9. Reset the clock
10. Run to the second breakpoint
11. Clock will show raw number of execution cycles between breakpoints.



Tip: You can save your breakpoints, probe points, graphs, and watch windows with [File -> Workspace -> Save Workspace As](#)

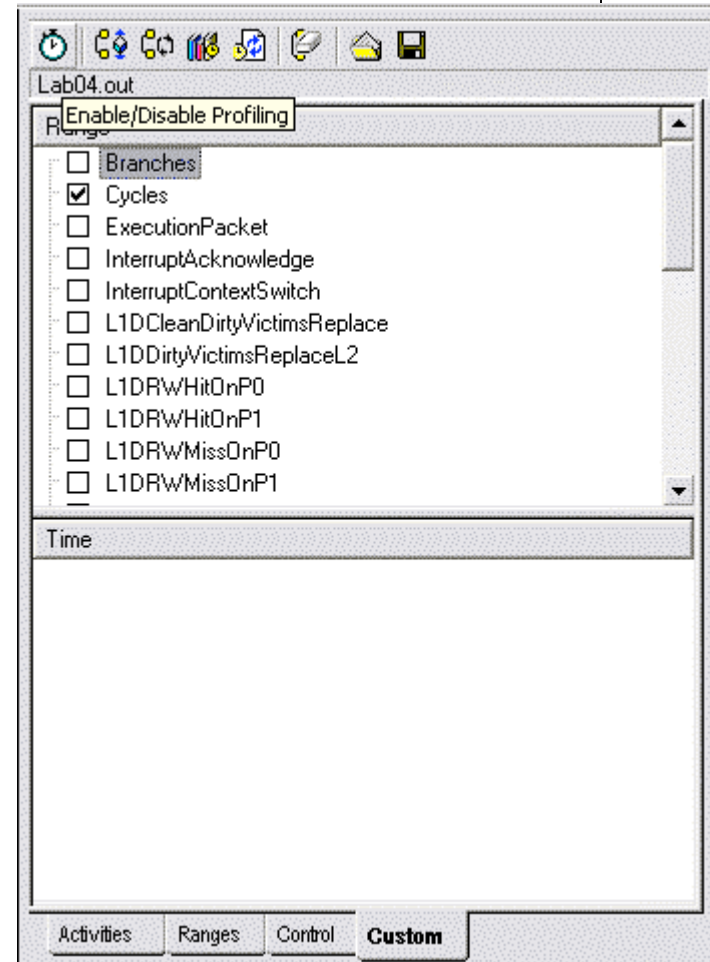


# Another method for estimating code execution time (part 1 of 3)

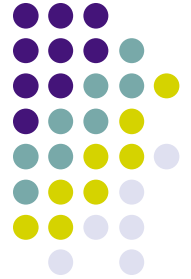


Repeat steps 1-4 previous method.

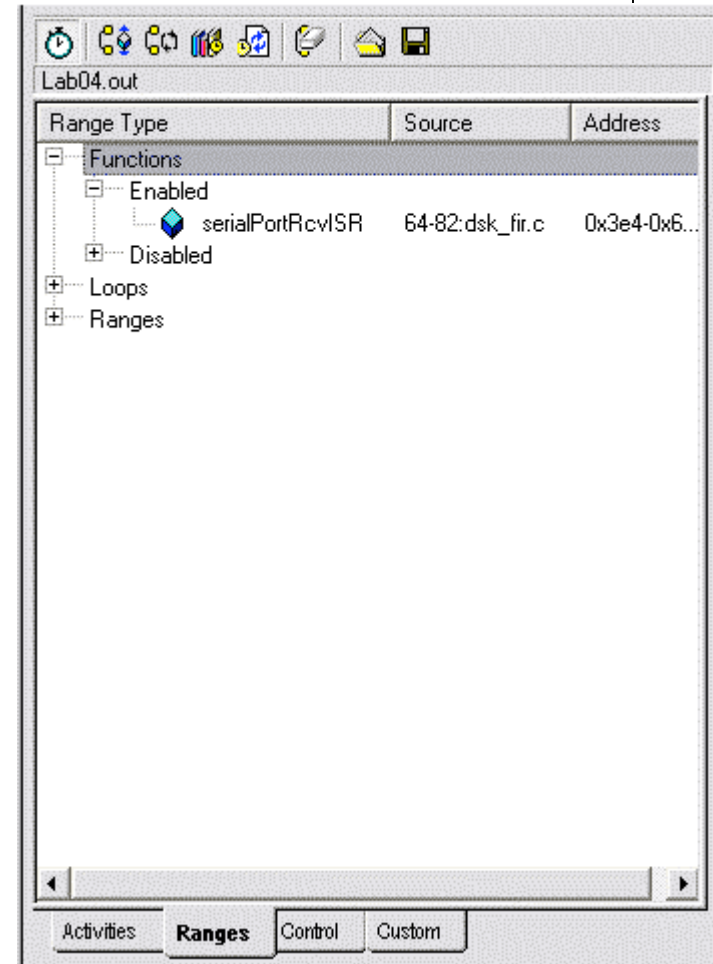
5. Clear any breakpoints in your code
6. Profile -> Setup
7. Click on Custom tab
8. Select "Cycles"
9. Click on clock (enable profiling)



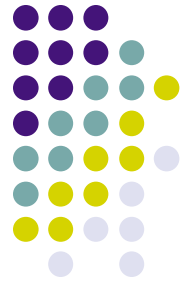
# Another method for estimating code execution time (part 2 of 3)



10. Select Ranges tab
11. Highlight code you want to profile and drag into ranges window (hint: you can drag whole functions into this window)
12. Repeat for other ranges if desired



# Another method for estimating code execution time (part 3 of 3)

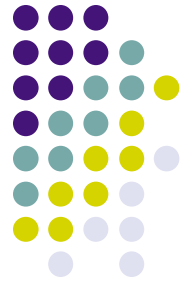


13. Profile -> Viewer
14. Run (let it run for a minute or more)
15. Halt
16. Observe profiling results in Profile Viewer window

The screenshot shows the Profile Viewer window with the following table:

Address Range	Symbol Name	SLR	Symbol Type	Access Count	Cycles: Incl. Avg.	Cycles: Excl. Avg.
0:0x3e4-0x670	serialPortRcvISR	64-82:dsk_fir.c	function	49	464	392

A red box with the text "Hint: edit the columns to see averages" is overlaid on the table. An arrow points from the bottom-left corner of the window to the Profiler icon in the toolbar.

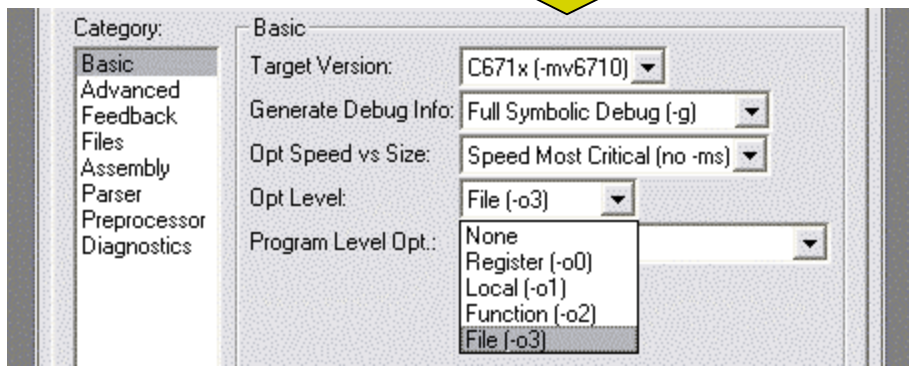
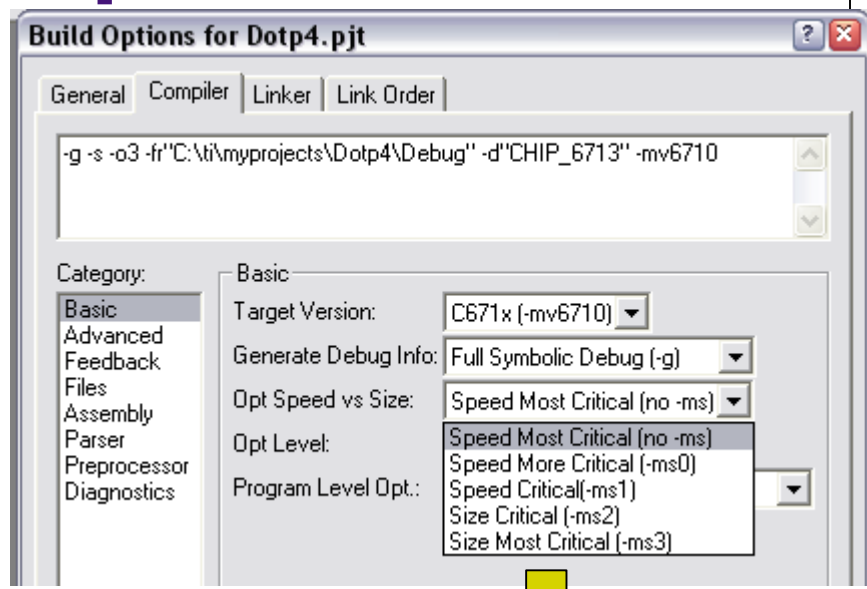
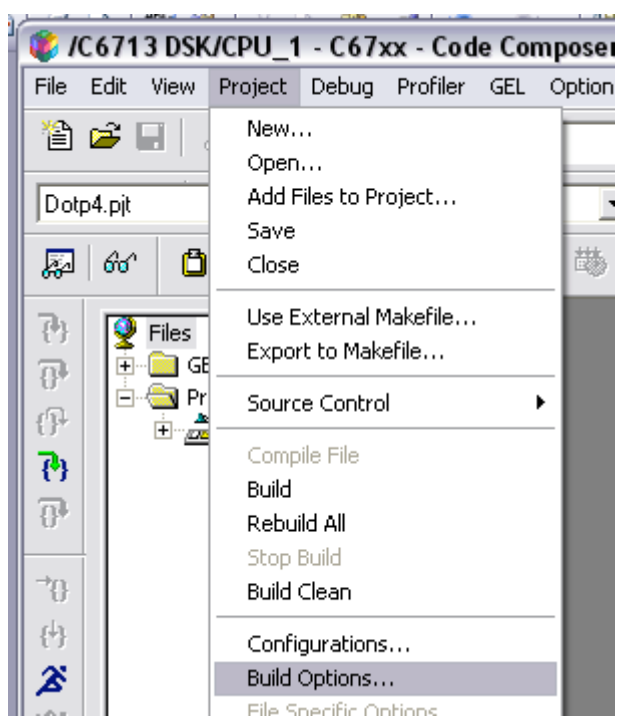


# What does it mean?

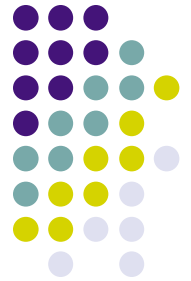
- **Access count** is the number of times that CCS profiled the function
  - Note that the function was probably called more than 49 times. CCS only timed it 49 times.
- **Inclusive average** is the average number of cycles needed to run the function *including* any calls to subroutines
- **Exclusive average** is the average number of cycles needed to run the function *excluding* any calls to subroutines



# Optimizing Compiler



# Profiling results after compiler optimization



- In this example, we get a 3x-4x improvement with “Speed Most Critical” and “File (-o3)” optimization
- Optimization gains can be much larger, e.g. 20x

Profile Viewer << 0 >> Current - C6713 DSK/CPU\_1

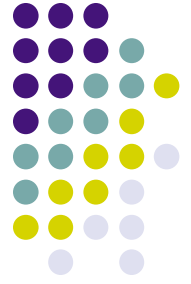
Address Range	Symbol Name	SLR	Symbol Type	Access Count	Cycles: Incl. Avg.	Cycles: Excl. Avg.
0:0x9a0-0xad0	serialPortRcvISR	64-82:dsk_fir.c	function	117	127	127

Profiler



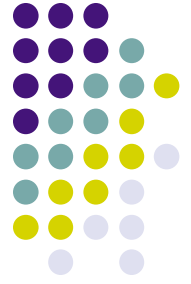
# Limitations of hardware profiling

- Breakpoint/clock profiling method may not work with compiler-optimized code
- Profile -> View method is known to be somewhat inaccurate when connected to real hardware (see “profiling limitations” in CCS help)
  - Accuracy is better when only one or two ranges are profiled
  - Best accuracy is achieved by running a simulator



# Other factors affecting code efficiency

- Memory
  - C6713 has 64kB internal ram (L2 cache)
  - DSK provides additional 16MB external RAM (SDRAM)
  - Code location (.text in command file)
    - internal memory (fast)
    - external memory (slow, typically 2-4x worse)
  - Data location (.data in command file)
    - internal memory (fast)
    - external memory (slow, depends on datatypes)
- Data types
  - Slowest execution is double-precision floating point
  - Fastest execution is fixed point, e.g. short



# Command file example

## MEMORY

```
{  
  vecs:      o = 00000000h      l = 00000200h  
  IRAM:      o = 00000200h      l = 0000FE00h  
  CE0:       o = 80000000h      l = 01000000h  
}
```

## SECTIONS

```
{  
  "vectors" > vecs  
  .cinit > IRAM  
  .text > IRAM  
  .stack > IRAM  
  .bss > IRAM  
  .const > IRAM  
  .data > IRAM  
  .far > IRAM  
  .switch > IRAM  
  .sysmem > IRAM  
  .tables > IRAM  
  .cio > IRAM  
}
```

Code goes here

Data goes here

Addresses 00000000-0000FFFF are mapped to internal memory (IRAM). This is 64kB.

External memory (CE0) is mapped to address range 80000000 – 80FFFFFFF. This is 16MB.



## TMS320C6000 C/C++ Data Types

Type	Size	Representation	Range	
			Minimum	Maximum
char, signed char	8 bits	ASCII	-128	127
unsigned char	8 bits	ASCII	0	255
short	16 bits	2s complement	-32768	32767
unsigned short	16 bits	Binary	0	65535
int, signed int	32 bits	2s complement	-2147483648	214783647
unsigned int	32 bits	Binary	0	4294967295
long, signed long	40 bits	2s complement	-549755813888	549755813887
unsigned long	40 bits	Binary	0	1099511627775
enum	32 bits	2s complement	-2147483648	214783647
float	32 bits	IEEE 32-bit	1.175494e-38†	3.40282346e+38
double	64 bits	IEEE 64-bit	2.22507385e-308†	1.79769313e+308
long double	64 bits	IEEE 32-bit	2.22507385e-308†	1.79769313e+308

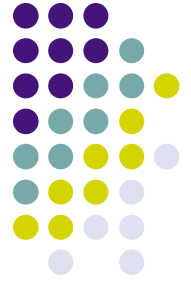




# Some Things to Try

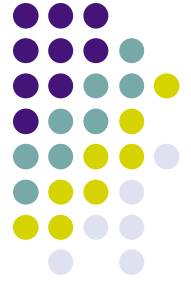
- Try profiling parts of your FIR filter code from Day 1 without optimization. Try both profiling methods.
- Rebuild your project under various optimization levels and try various settings from “size most critical” to “speed most critical”.
- Compare profile results for no optimization and various levels of optimization.
- Change the data types in your FIR filter code and rebuild (with and without optimization) to see the effect on performance.
- Try moving the data and/or program to internal/external memory and profiling (you will need to modify the linker command file to do this)
- Contest: Who can make the most efficient 8th order bandpass filter (that works)?

# Assembly Language Programming on the TMS320C6713

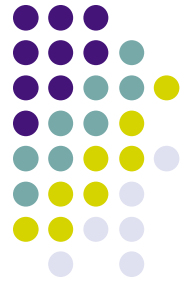


- Sometimes you have to take matters into your own hands...
- Three options:
  1. Linear assembly (.sa)
    - Compromise between effort and efficiency
    - Typically more efficient than C
    - Assembler takes care of details like assigning “functional units”, registers, and parallelizing instructions
  2. ASM statement in C code (.c)
    - asm(“assembly code”)
  3. C-callable assembly function (.asm)
    - Full control of assigning functional units, registers, parallelization, and pipeline optimization

# C-Callable Assembly Language Functions



- Basic concepts:
  - Arguments are passed in via registers A4, B4, A6, B6, ... in that order. All registers are 32-bit.
  - Result returned in A4 also.
  - Return address of calling code (program counter) is in B3. Don't overwrite B3!
  - Naming conventions:
    - In C code: label
    - In ASM code: `_label` (note the leading underbar)
  - Accessing global variables in ASM:
    - `.ref _variablename`



# Skeleton C-Callable ASM Function

; header comments

; passed in parameters in registers A4, B4, A6, ... in that order

```
.def _myfunc                ; allow calls from external
ACONSTANT .equ 100          ; declare constants
                .ref _aglobalvariable ; refer to a global variable

_myfunc:  NOP                ; instructions go here
                B            B3    ; return (branch to addr B3)
                ; function output will be in A4
                NOP            5    ; pipeline flush

                .end
```



# Example C-Callable Assembly Language Program



```

;FIRCASMfunc.asm ASM function called from C to implement FIR
;A4 = Samples address, B4 = coeff address, A6 = filter order
;Delays organized as:x(n-(N-1))...x(n);coeff as h[0]...h[N-1]

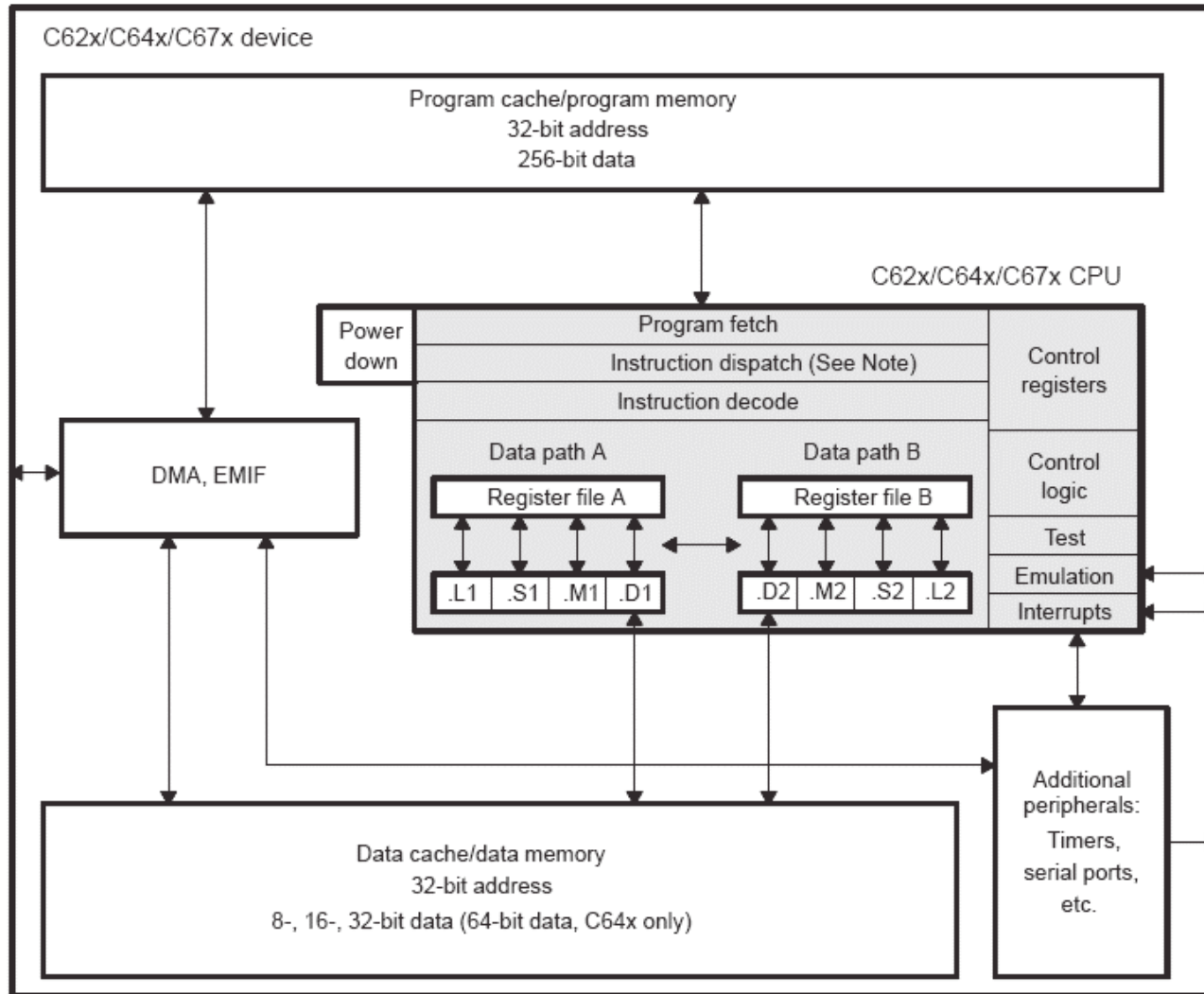
        .def      _fircasmfunc
_fircasmfunc:
        MV        A6,A1          ;ASM function called from C
        MPY       A6,2,A6        ;setup loop count
        ZERO      A8            ;since dly buffer data as byte
        ADD       A6,B4,B4       ;init A8 for accumulation
        SUB       B4,1,B4        ;since coeff buffer data as byte
        loop:
        LDH       *A4++,A2       ;B4=bottom coeff array h[N-1]
        LDH       *B4--,B2       ;start of FIR loop
        NOP       4              ;A2=x[n-(N-1)+i] i=0,1,...,N-1
        MPY       A2,B2,A6       ;B2=h[N-1-i] i=0,1,...,N-1
        NOP
        ADD       A6,A8,A8       ;A6=x[n-(N-1)+i]*h[N-1-i]
        LDH       *A4,A7         ;accumulate in A8
        NOP       4              ;A7=x[(n-(N-1)+i+1]update delays
        STH       A7,*-A4[1]     ;using data move "up"
        SUB       A1,1,A1        ;-->x[(n-(N-1)+i] update sample
        [A1] B    loop           ;decrement loop count
        NOP       5              ;branch to loop if count # 0

        MV        A8,A4          ;result returned in A4
        B         B3             ;return addr to calling routine
        NOP       4
    
```





# TMS320C67x Block Diagram



One instruction is 32 bits. Program bus is 256 bits wide.

⇒ Can execute up to 8 instructions per clock cycle (225MHz→4.4ns clock cycle).

8 independent functional units:  
- 2 multipliers  
- 6 ALUs

Code is efficient if all 8 functional units are always busy.

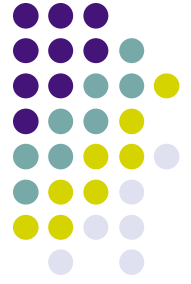
Register files each have 16 general purpose registers, each 32-bits wide (A0-A15, B0-B15).

Data paths are each 64 bits wide.



# C6713 Functional Units

- Two data paths (A & B)
- Data path A
  - Multiply operations (**.M1**)
  - Logical and arithmetic operations (**.L1**)
  - Branch, bit manipulation, and arithmetic operations (**.S1**)
  - Loading/storing and arithmetic operations (**.D1**)
- Data path B
  - Multiply operations (**.M2**)
  - Logical and arithmetic operations (**.L2**)
  - Branch, bit manipulation, and arithmetic operations (**.S2**)
  - Loading/storing and arithmetic operations (**.D2**)
- All data (not program) transfers go through **.D1** and **.D2**



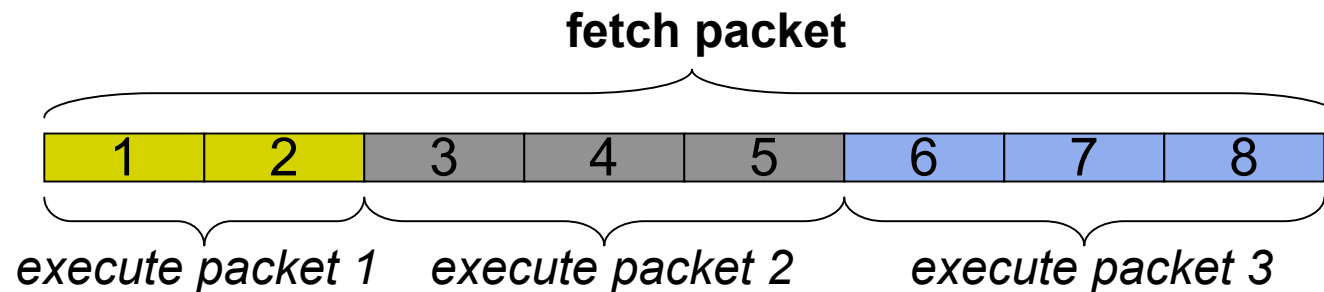
# Fetch & Execute Packets

- C6713 fetches 8 instructions at a time (256 bits)
- Definition: “Fetch packet” is a group of 8 instructions fetched at once.
- Coincidentally, C6713 has 8 functional units.
  - Ideally, all 8 instructions would be executed in parallel.
- Often this isn’t possible, e.g.:
  - 3 multiplies (only two .M functional units)
  - Results of instruction 3 needed by instruction 4 (must wait for 3 to complete)



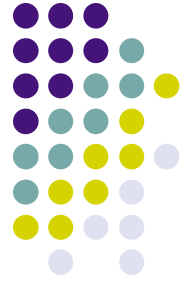
# Execute Packets

- Definition: “Execute Packet” is a group of (8 or less) consecutive instructions in one fetch packet that can be executed in parallel.



- C compiler provides a flag to indicate which instructions should be run in parallel.
- You have to do this manually in Assembly using “||”. See Chapter 3 of the Chassaing textbook.

# C6713 Instruction Pipeline Overview



All instructions flow through the following steps:

## 1. Fetch

- a) PG: Program address Generate
- b) PS: Program address Send
- c) PW: Program address ready Wait
- d) PR: Program fetch packet Receive

## 2. Decode

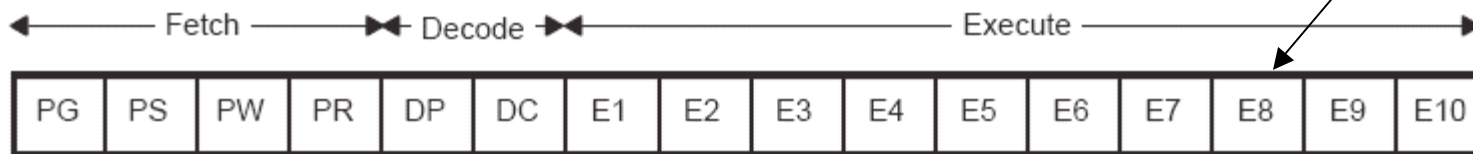
- a) DP: Instruction DisPatch
- b) DC: Instruction DeCode

## 3. Execute

- a) 10 phases labeled E1-E10
- b) Fixed point processors have only 5 phases (E1-E5)

**each step  
= 1 clock cycle**

Figure 7-5. Floating-Point Pipeline Phases





# Pipelining: Ideal Operation

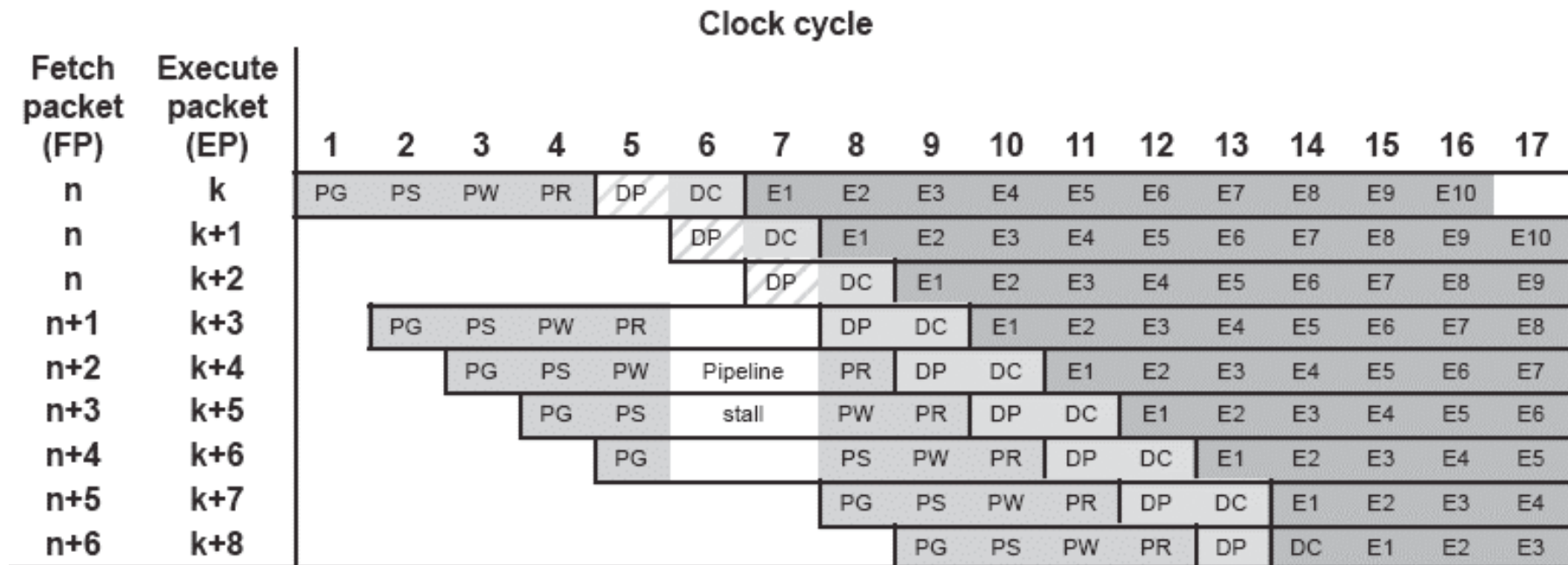
Fetch packet	Clock cycle																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
n	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	
n+1		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
n+2			PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9
n+3				PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8
n+4					PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7
n+5						PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
n+6							PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
n+7								PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
n+8									PG	PS	PW	PR	DP	DC	E1	E2	E3
n+9										PG	PS	PW	PR	DP	DC	E1	E2
n+10											PG	PS	PW	PR	DP	DC	E1

## Remarks:

- At clock cycle 11, the pipeline is “full”
- There are no holes (“bubbles”) in the pipeline in this example

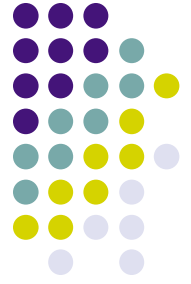


# Pipelining: “Actual” Operation



## Remarks:

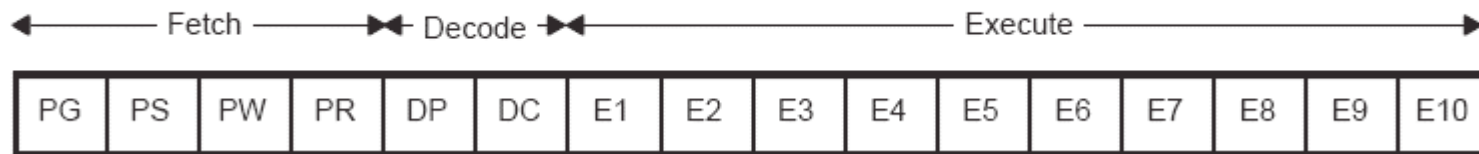
- Fetch packet n has 3 execution packets
- All subsequent fetch packets have 1 execution packet
- Notice the holes/bubbles in the pipeline caused by lack of parallelization



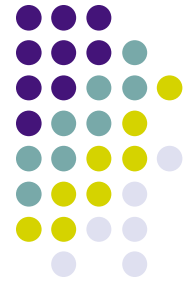
# Execute Stage of C6713 Pipeline

- C67x has 10 execute phases (floating point)

Figure 7–5. Floating-Point Pipeline Phases



- C62x/C64x have 5 execute phases (fixed point)
- Different types of instructions require different numbers of these phases to complete their execution
  - Anywhere between 1 and all 10 phases
  - Most instruction tie up their functional unit for only one phase (E1)



# Execution Stage Examples (1)

## ABSSP

*Single-Precision Floating-Point Absolute Value*

### Syntax

**ABSSP** (.unit) *src2*, *dst*

.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xsp	.S1, .S2
<i>dst</i>	sp	

### Pipeline

Pipeline Stage	
	E1
Read	<i>src2</i>
Written	<i>dst</i>
Unit in use	.S

**results available after E1 (zero delay slots)**

**Functional unit free after E1 (1 functional unit latency)**

### Instruction Type

Single-cycle





# Execution Stage Examples (2)

## ADDSP

*Single-Precision Floating-Point Addition*

Syntax

**ADDSP** (.unit) *src1*, *src2*, *dst*

.unit = .L1 or .L2

Pipeline

Pipeline Stage	E1	E2	E3	E4
Read	<i>src1</i> <i>src2</i>			
Written				<i>dst</i>
Unit in use	.L			

Instruction Type

4-cycle

**results available after E4 (3 delay slots)**

Delay Slots

3

Functional Unit Latency

1

**Functional unit free after E1 (1 functional unit latency)**





# Execution Stage Examples (3)

## MPYDP

*Double-Precision Floating-Point Multiply*

Syntax

MPYDP (.unit) src1, src2, dst

Pipeline

.unit = .M1 or .M2

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
Read	src1_l src2_l	src1_l src2_h	src1_h src2_l	src1_h src2_h						
Written									dst_l	dst_h
Unit in use	.M	.M	.M	.M						

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTD**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

Instruction Type

MPYDP

**Results available after E10 (9 delay slots)**

Delay Slots

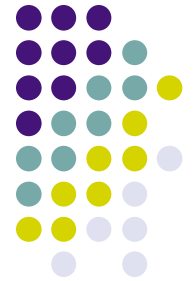
9

**Functional unit free after E4 (4 functional unit latency)**

Functional Unit Latency

4





# Functional Latency & Delay Slots

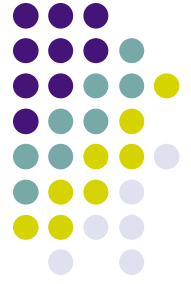
- **Functional Latency:** How long must we wait for the functional unit to be free?
- **Delay Slots:** How long must we wait for the result?
- General remarks:
  - Functional unit latency  $\leq$  Delay slots
  - Strange results will occur in ASM code if you don't pay attention to delay slots and functional unit latency
  - All problems can be resolved by “waiting” with NOPs
  - Efficient ASM code tries to keep functional units busy all of the time.
  - Efficient code is hard to write (and follow).

# Lunch Break



Workshop resumes at 1:30pm...

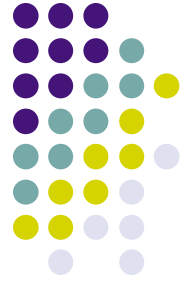




# Some Things to Try

- Try rewriting your FIR filter code as a C-callable ASM function
  - Create a new ASM file
  - Call the ASM function from your main code
  - See Chassaing examples [fircasm.pjt](#) and [fircasmfast.pjt](#) for ideas
- Profile your new FIR code and compare to the optimized compiler.

# Infinite Impulse Response (IIR) Filters



- Advantages:
  - Can achieve a desired frequency response with less memory and computation than FIR filters
- Disadvantages:
  - Can be unstable
  - Affected more by finite-precision math due to feedback
- Input/output relationship:

$$y[n] = \sum_{k=0}^{M-1} b[k]x[n-k] - \sum_{k=1}^{N-1} a[k]y[n-k]$$



# IIR Filtering - Stability

- Transfer function:

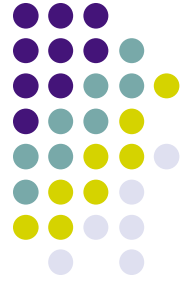
$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M-1} b[k]z^{-k}}{1 + \sum_{k=1}^{N-1} a[k]z^{-k}} = \frac{b_0 + b_1z^{-1} + \dots + b_{M-1}z^{-(M-1)}}{1 + a_1z^{-1} + \dots + a_{N-1}z^{-(N-1)}}$$

- Note that the filter is stable only if all of its poles (roots of the denominator) have magnitude less than 1.
- Easy to check in Matlab: **max(abs(roots(a)))**
- Quantization of coefficients (a's and b's) will move the poles. A stable filter in infinite precision **may not be stable after coefficient quantization.**
- Numerator of H(z) does not affect stability.



# Creating IIR Filters

1. Design filter **Matlab**
  - Type: low pass, high pass, band pass, band stop, ...
  - Filter order N
  - Desired frequency response
2. Decide on a realization structure
3. Decide how coefficients will be quantized.
4. Compute coefficients
5. Decide how everything else will be quantized **CCS**  
(input samples, output samples, result of multiplies, result of additions)
6. Write code to realize filter
7. Test filter and compare to theoretical expectations

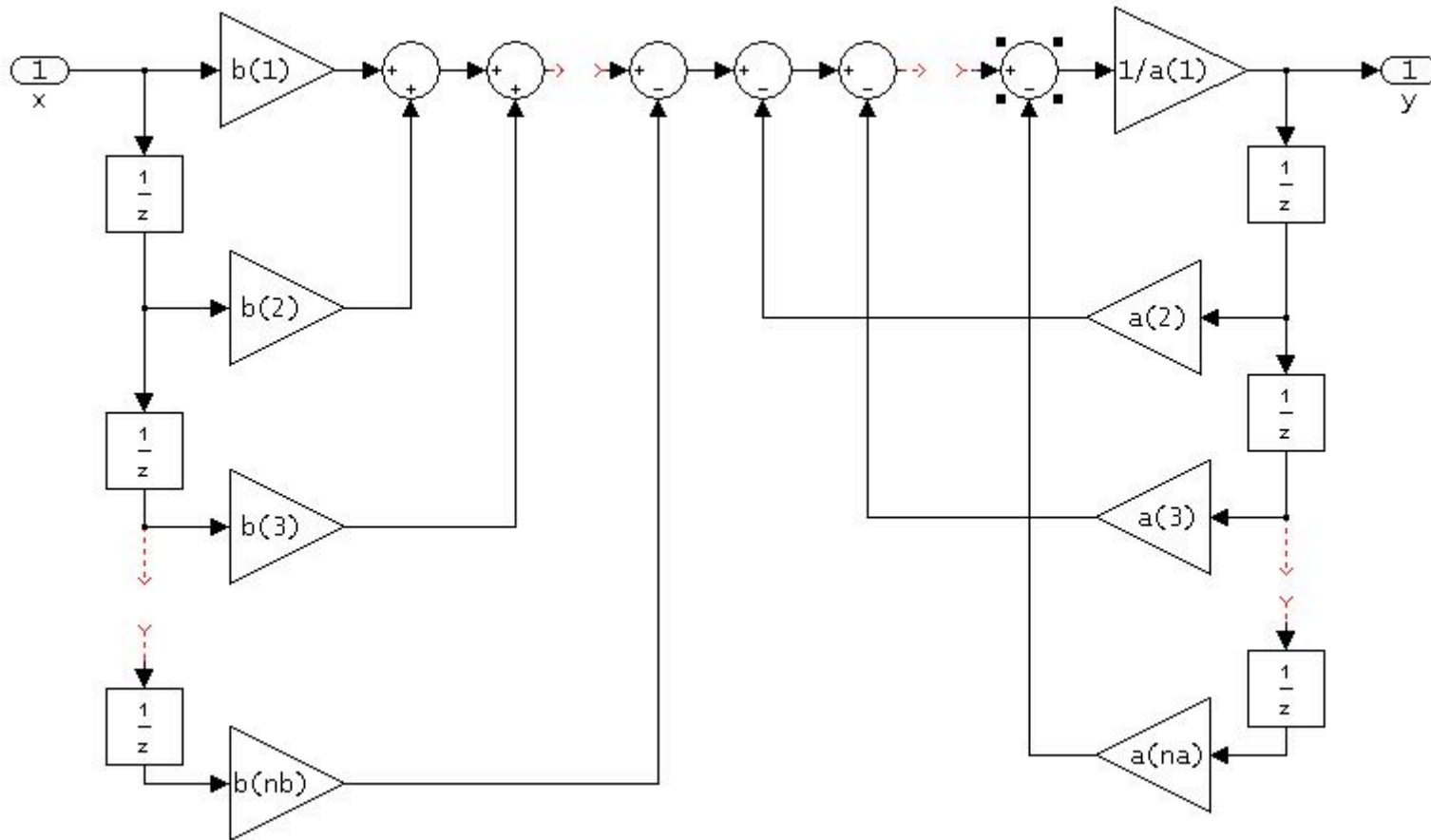


# IIR Realization Structures

- Many different IIR realization structures available (see options in fdatool)
  - Structures can have different memory and computational requirements
  - All structures give the same behavior when the math is infinite precision
  - Structures can have very different behavior when the math is finite precision
    - Stability
    - Accuracy with respect to the desired response
    - Potential for overflow/underflow



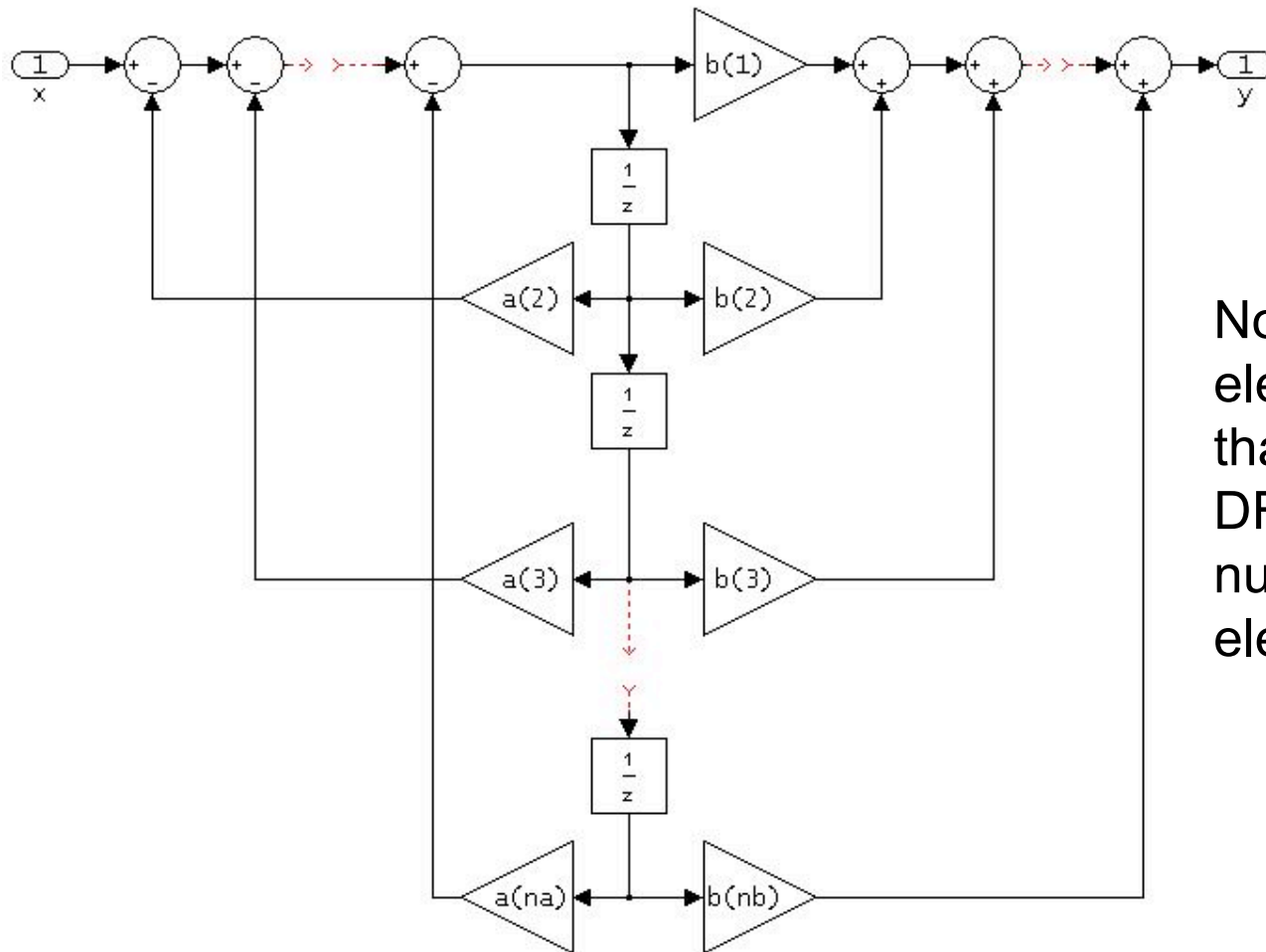
# Direct Form I



Notation:  $1/z$  = one sample delay



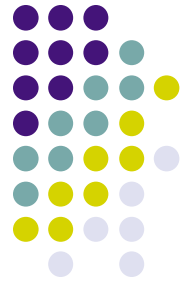
# Direct Form II



Note fewer delay elements (less memory) than DFI. Can prove that DFII has **minimum** number of delay elements.

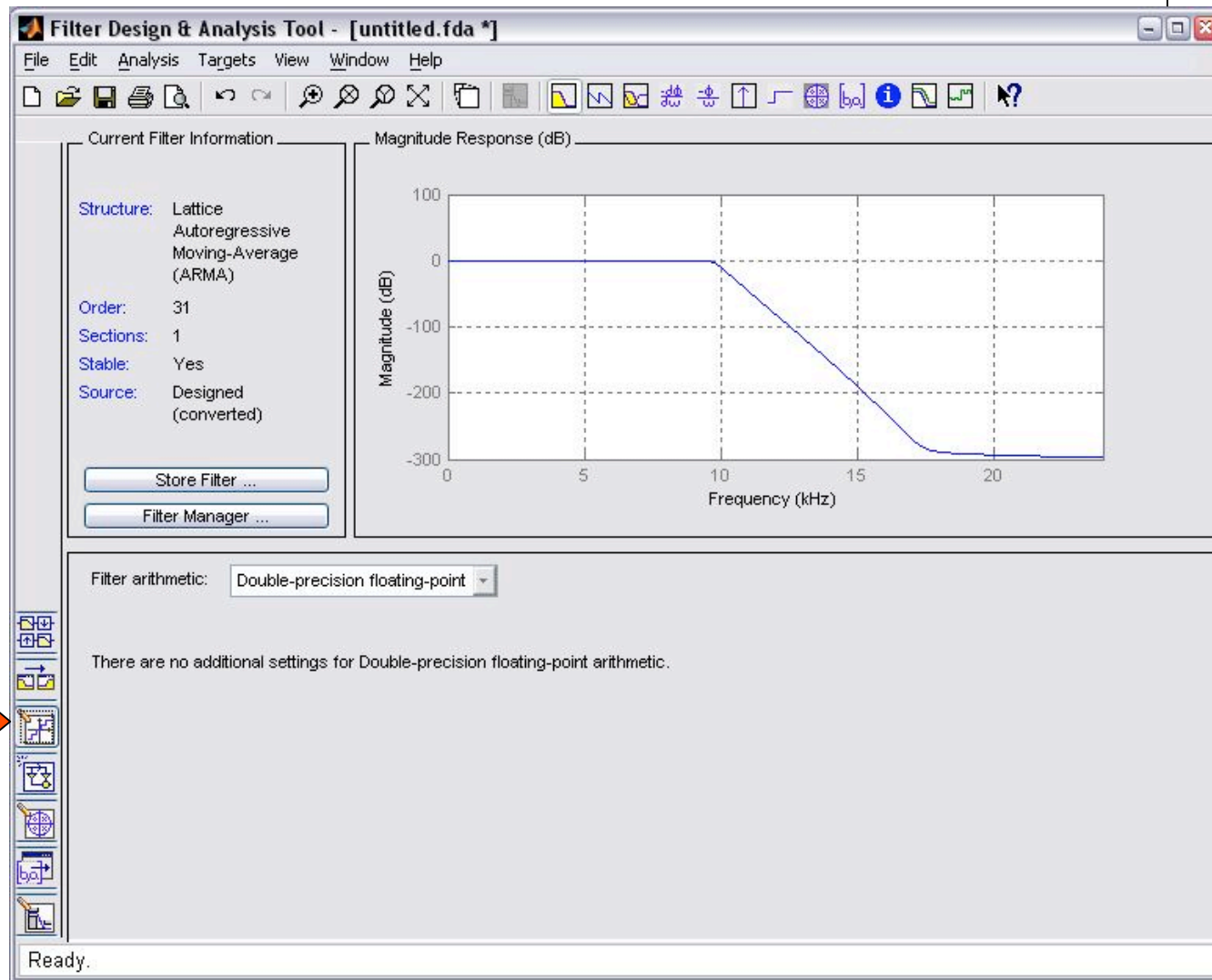


# Direct Form II: Second Order Sections



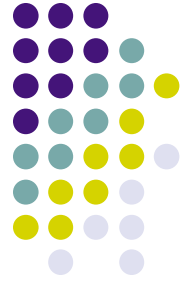
- Very popular realization structure
  - Low memory requirements (same as DFII)
  - Easy to check the stability of each SOS
  - Can write one DFII-SOS filter function and reuse it for any length filter
  - Tends to be less sensitive to finite precision math than DFI or DFII. Why?
    - Dynamic range of coefficients in each SOS is smaller
    - Coefficient quantization only affects local poles/zeros

# Determining How Coefficient Quantization Will Affect Your Filter



set quantization parameters





# IIR Filtering Final Remarks

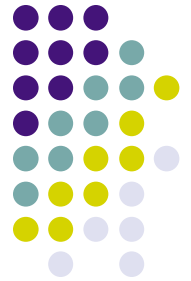
- IIR filters are more sensitive to choice of **realization structure** and **data types** than FIR filters due to feedback
  - Memory requirements
  - Time required to compute filter output
  - Accuracy with respect to the desired response
  - Stability
  - Potential for overflow/underflow
- **fdatool** can be useful for examining the tradeoffs before writing code



# Some Things to Try

- In fdatool, design an IIR filter with the following specs:
  - Bandstop
  - First passband 0-2500Hz, 0dB nominal gain, 0.5dB max deviation
  - First transition band 2500-3500Hz
  - Stop band 3500-10500Hz, -20dB minimum suppression
  - Second transition band 10500-12500Hz
  - Second passband 12500-22050Hz 0dB nominal gain, 0.5dB max deviation
  - Minimum filter order
- Explore DFII with and without Second Order Sections
- Try various coefficient quantizations including fixed point
- Implement your “best” filter in CCS
- Compare actual performance to the theoretical predictions

# Other Interesting Applications of Real-Time DSP



- Fast Fourier Transform (FFT): Chapter 6
  - Example projects:
    - DFT, FFT256C, FFTSinetable, FFTr2, FFTr4, FFTr4\_sim, fastconvo, fastconvo\_sim, graphicEQ
    - Note that TI provides optimized FFT functions (search for cfftr2\_dif, cfftr2\_dif, cfftr4\_dif)
- Adaptive Filtering: Chapter 7
  - Example projects:
    - Adaptc, adaptnoise, adaptnoise\_2IN, adaptIDFIR, adaptIDFIRw, adaptIDIIR, adaptpredict, adaptpredict\_2IN,



# Workshop Day 2 Summary

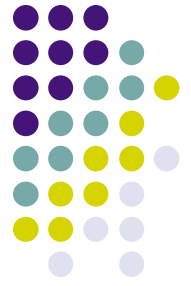
## What you learned today:

- How to use Matlab's Link for Code Composer Studio
- How to profile code size and execution times.
- How data types and memory usage affect code execution times.
- How to reduce code size and execution time with CCS's optimizing compiler.
- How assembly language can be integrated into your projects.
- Basics of the TMS320C6713 architecture.
  - Fetch packets, execute packets, pipelining
  - Functional unit latency and delay slots
- How to design and implement IIR filters on the C6713
  - Realization structures
  - Quantization considerations
- Other applications for the C6713 DSK
  - FFT
  - Adaptive filtering



# Workshop Day 2

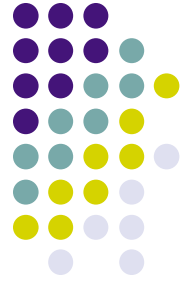
## Reference Material



- Matlab's Link for Code Composer Studio help (>> doc ccslink)
- Chassaing textbook Chapters 3, 5-8
- CCS Help system
- **SPRU509F.PDF** CCS v3.1 IDE Getting Started Guide
- **C6713DSK.HLP** C6713 DSK specific help material
- **SPRU198G.PDF** TMS320C6000 Programmer's Guide
- **SPRU189F.PDF** TMS320C6000 CPU and Instruction Set Reference Guide
- Matlab fdatool help (>> doc fdatool)
- Other Matlab help (>> doc soundsc, >> doc wavwrite)

***Latest documentation available at  
[http://www.ti.com/sc/docs/psheets/man\\_dsp.htm](http://www.ti.com/sc/docs/psheets/man_dsp.htm)***





# Additional Exploration

- Explore some of Chassaing's FFT and adaptive filtering projects in the "myprojects" directory
- Explore some of the reference literature (especially the Chassaing text and the CCS help system)
- Try a lab assignment in the ECE4703 real-time DSP course: <http://spinlab.wpi.edu/courses/ece4703>