

DSP Hardware Laboratory

Class 3

9/16/09

Introduction to FIR Filters

C runtime environment

Reading Assigned : Chassaing Ch. 4

DDS Q/A

- Explain DDS as just a sinewave generator
- Desired output :
 - $Y(t) = A \sin(\omega t)$
 - $Y(t) = A \sin(2\pi f t)$
- “Discretizing”
 - $Y[n \cdot t_s] = A \sin(2\pi f n \cdot t_s)$
 - Or $y[n] = A \sin(2\pi n f / f_s)$

“Ideal” DDS progression

- $Y[n] = A \cdot \sin(2 \cdot \pi \cdot n \cdot f / f_s)$

```
n=0
While(1)
{
  y = A*sin(2*pi*n*f/fs);
  send the sample
  n++;
}
```

```
phase=0
While(1)
{
  y = A*sin(phase);
  send the sample
  phase += 2*pi*f/fs;
}
```

```
phase=0
While(1)
{
  y = A*sin(phase);
  send the sample
  phase += precomputed_step;
}
```

```
phase=0
While(1)
{
  y = A*sin(phase);
  send the sample
  phase += precomputed_step;
  if phase >= 2pi, phase -= 2pi
}
```

Non-ideal approximations

1st approximation : take phase, which is from $0-2\pi$ and map to a finite length table to approximate the value of the sine function

```
phase=0
While(1)
{
  y = A*sin(phase);
  send the sample
  phase += precomputed_step;
  if phase >= 2pi, phase -= 2pi
}
```

2nd approximation : phase ($0-2\pi$), needs to be represented by a finite precision number (integer). The more accurately that you can preserve the real phase when making this quantization, the more accurate freq you can get.

Where We Are, Where We Are Going

- Introduction
- Sine Generation (DDS)
- **FIR Filtering (C)**
- FIR Filtering (Optimized)
- IIR Filtering
- Discrete Fourier Transform, Fast Fourier Transform
- ...
- McBSP
- ALU
- MAC
- Parallel Execution
- Memory Spaces
 - Caching, Arrangement
- Direct Memory Access (DMA)
- Circular Buffering
- Assembly
- Floating point/Fixed Point
 - Q15, Saturation

LED/SW Location : 0x90080000

Table 2: CPLD USER_REG Register

Bit	Name	R/W	Description
7	USER_SW3	R	User DIP Switch 3(1 = Off, 0 = On)
6	USER_SW2	R	User DIP Switch 2(1 = Off, 0 = On)
5	USER_SW1	R	User DIP Switch 1(1 = Off, 0 = On)
4	USER_SW0	R	User DIP Switch 0(1 = Off, 0 = On)
3	USER_LED3	R/W	User-defined LED 3 Control (0 = Off, 1 = On)
2	USER_LED2	R/W	User-defined LED 2 Control (0 = Off, 1 = On)
1	USER_LED1	R/W	User-defined LED 1 Control (0 = Off, 1 = On)
0	USER_LED0	R/W	User-defined LED 0 Control (0 = Off, 1 = On)

This memory address is in external memory, and is decoded by the on-board CPLD in order to provide you with switches and leds

Reading/Writing I/O Ports

```
unsigned int userDipSettings;  
//Read  
User_reg = *(unsigned volatile char *)(0x90080000);  
//Write  
*(unsigned volatile char *)(0x90080000) = what I want to write to LEDs;
```

- **volatile declarator**
 - The **volatile** keyword is a type qualifier used to declare that an object can be modified in the program by something other than statements, such as the operating system, the hardware, or a concurrently executing thread.
- The following example declares a volatile integer nVint whose value can be modified by external processes:
 - int volatile nVint;
- Objects declared as **volatile** are not used in optimizations because their value can change at any time.
 - The system always reads the current value of a volatile object at the point it is requested, even if the previous instruction asked for a value from the same object.
 - Also, the value of the object is written immediately on assignment.

CODEC Output

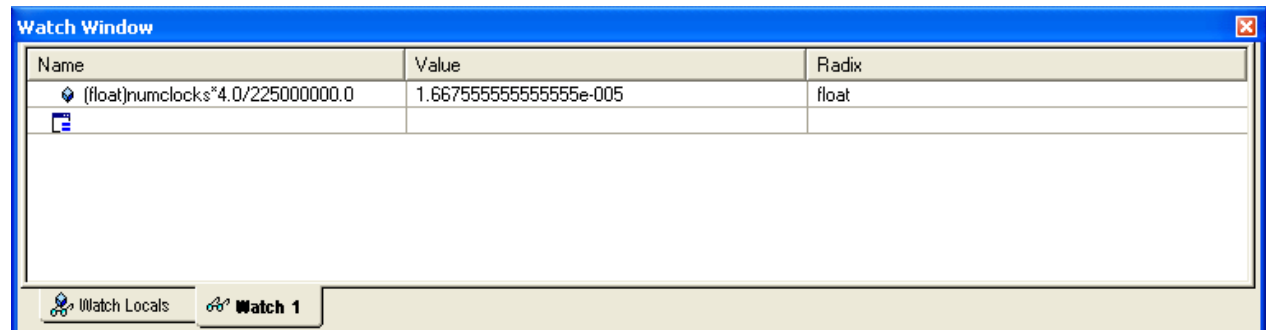
- Be careful of the data formatting when calling “output_sample()”
 - Output_sample() takes an “int” as a parameter
 - 32 bits
 - 16 correspond to left, and 16 to right
 - Each 16 bit value is then treated by the codec as a signed integer.
 - Really, this is an array of two short integers
 - When writing directly from DAC table (integers) this is probably not a problem. One might have to think a few minutes about the appropriate type conversion when starting with a floating point number.

Brief Aside : Timing Estimation / Measurement

```
#define TIMER_CTL          (*((volatile unsigned int*) 0x01940000))
#define TIMER_PRD        (*((volatile unsigned int*) 0x01940004))
#define TIMER_CNT        (*((volatile unsigned int*) 0x01940008))
void start_timer()
// just starts the c6713 timer counting at its cpucck/4 rate
{
    TIMER_PRD = 0xffffffff; //should rollover at 4billion
    TIMER_CTL = 0x000003c0;
}

// now, around something you wish to time
start_time = TIMER_CNT;
dout = DSPF_dp_dotprod(&(datareg[newdataindex]), taps, NUMTAPS);
stop_time = TIMER_CNT;
numclocks = stop_time-start_time;
```

Cpucck = 225MHz



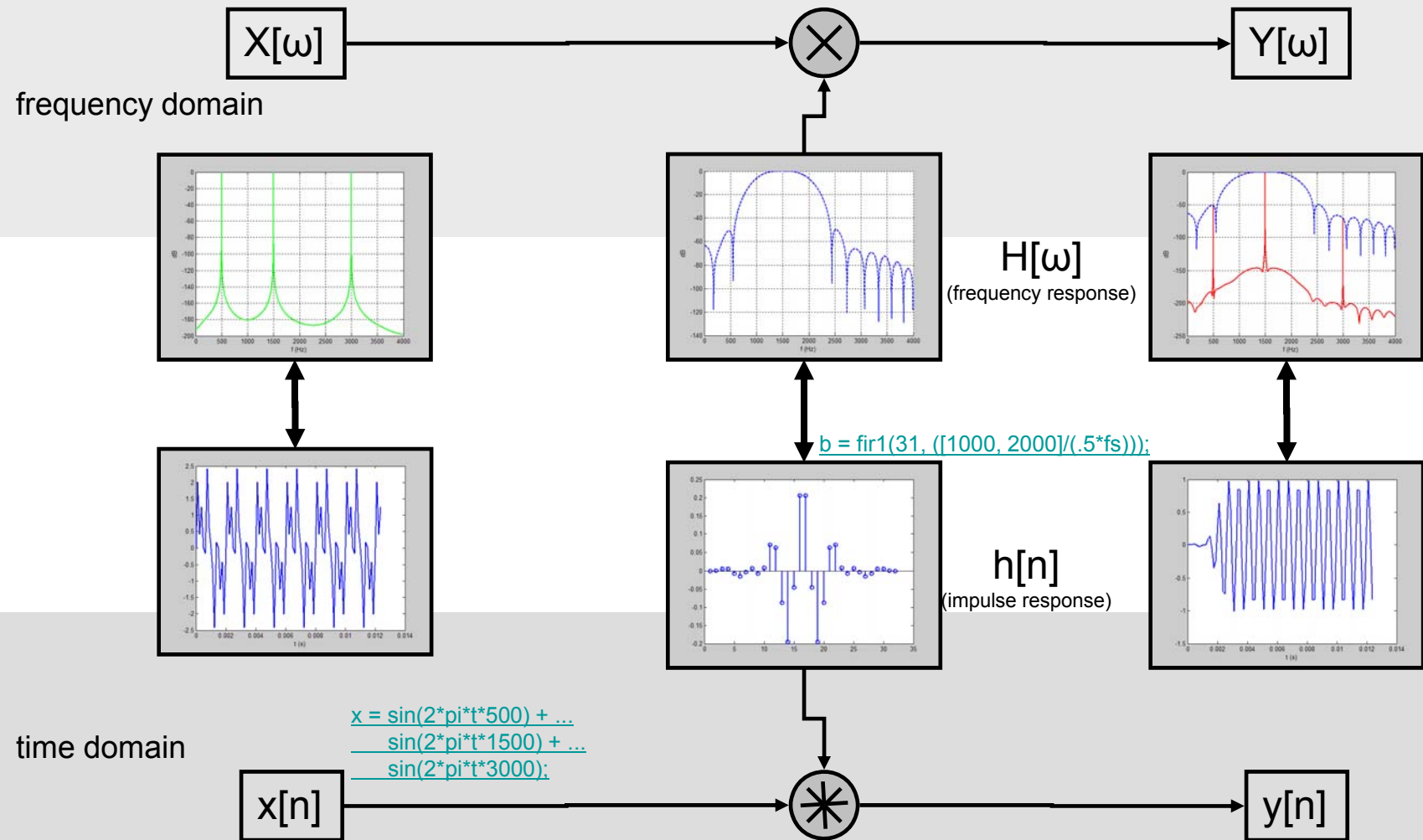
FIR1 Laboratory

- This laboratory assignment will introduce the implementation of a FIR filter in a DSP.
 - In addition, the DDS developed for Lab 2 will be used as an “agile frequency source” to create a chirp signal to test the frequency response of the FIR filter implementation.
 - This assignment will focus on designing a filter in MATLAB, implementing the filter entirely in C, and measuring its frequency response.
 - This is the first of a two-part assignment – next week’s lab, **FIR, Lab 2** will focus on DSP-specific features that can accelerate the FIR implementation as well as DSP libraries optimized to perform fast filtering.

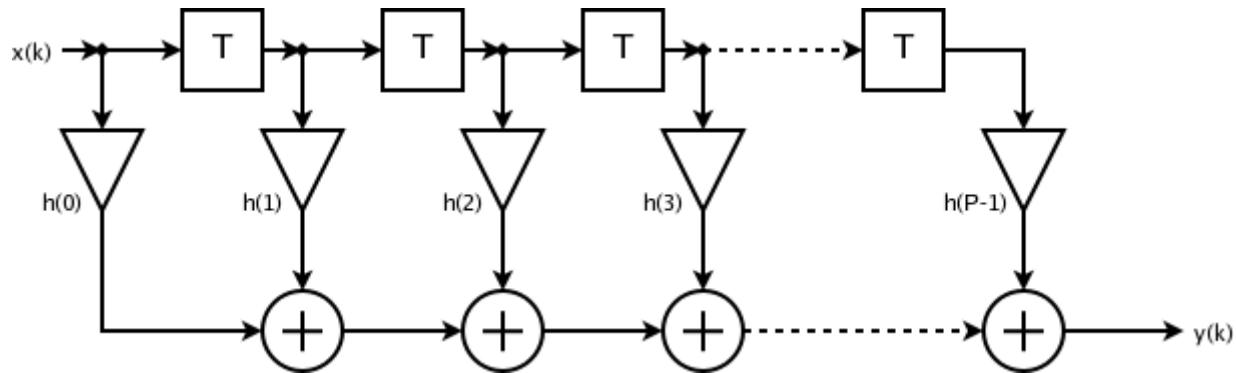
Convolution Theorem

- Key fact:
 - Multiplication in the frequency domain is the equivalent to convolution in the time domain (and vice-versa) – this is known as the *Convolution Theorem* (see <http://mathworld.wolfram.com/ConvolutionTheorem.html>)
 - Stated another way: Multiplication of Fourier transforms implies the convolution of the corresponding sequences
- Therefore, our filtering operation can be described as either:
 - Frequency Domain: $Y(z) = X(z)H(z)$
 - Time Domain:
$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k]$$

Finite Impulse Response (FIR) Filtering



Implementation



- “T” is a delay (often written as z^{-1}) – in our case it will be a memory location
- $h(0:P-1)$ is the impulse response of the FIR filter – i.e. the signal we will convolve the input with
 - Each $h[n]$ is often called a “tap”

CHAISSING Implementation

```
interrupt void c_int11()          //ISR
{
    short i;

    dly[0] = input_sample();      //new input @ beginning
    yn = 0;                       //initialize output

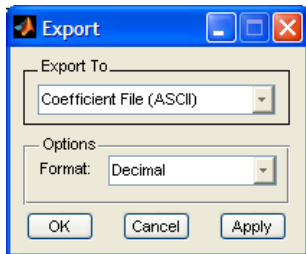
    //convolution
    for (i = 0; i < N; i++)
        yn += (h[i] * dly[i]); //y(n) += h(i)* x(n-i)

    //shift data
    for (i = N-1; i > 0; i--) //starting @ end
        dly[i] = dly[i-1]; //update delays

    output_sample(yn >> 15);     //scale output filter
    return;
}
```

MATLAB FIR Filter Design Using FDATool

Export coefficients using
 "File|Export":

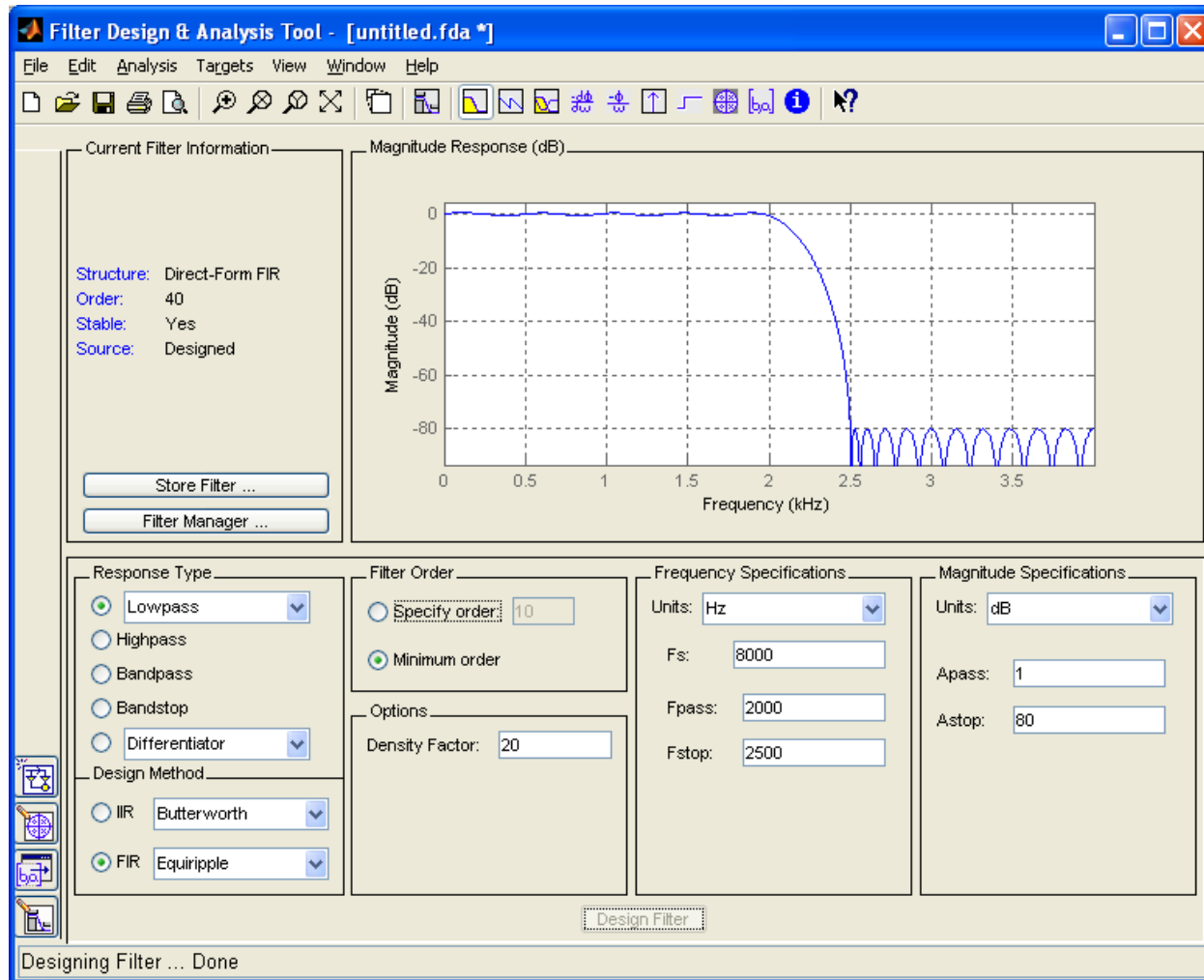


```

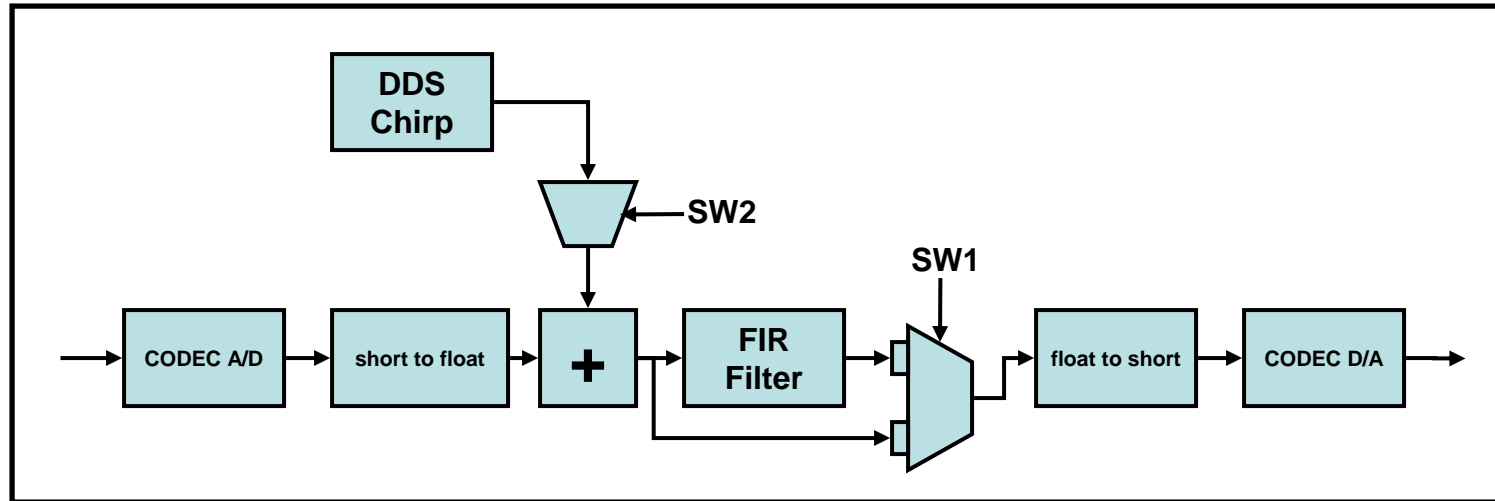
% Generated by MATLAB(R) 7.0.1 and the ...
% Generated on: 26-Sep-2005 12:23:19
%
% Coefficient Format: Decimal
%
% Discrete-Time FIR Filter (real)
%
%-----
% Filter Structure : Direct-Form FIR
% Filter Order    : 50
% Stable          : Yes
% Linear Phase    : Yes (Type 1)
%
% Numerator:
-0.00091909820846825603
-0.0027176960265955
-0.0024869527598323101
 0.0036614383835070902
 0.0136509252306624
 0.017351165901093299
 0.0076653061904216804
-0.0065547188696423999
-0.0076967840370653602
 0.00610545942113943601
    
```

A C-header file can also be created from the "Targets" menu

fdatool



FIR1_Lab Block Diagram



FIR1 Lab Requirements

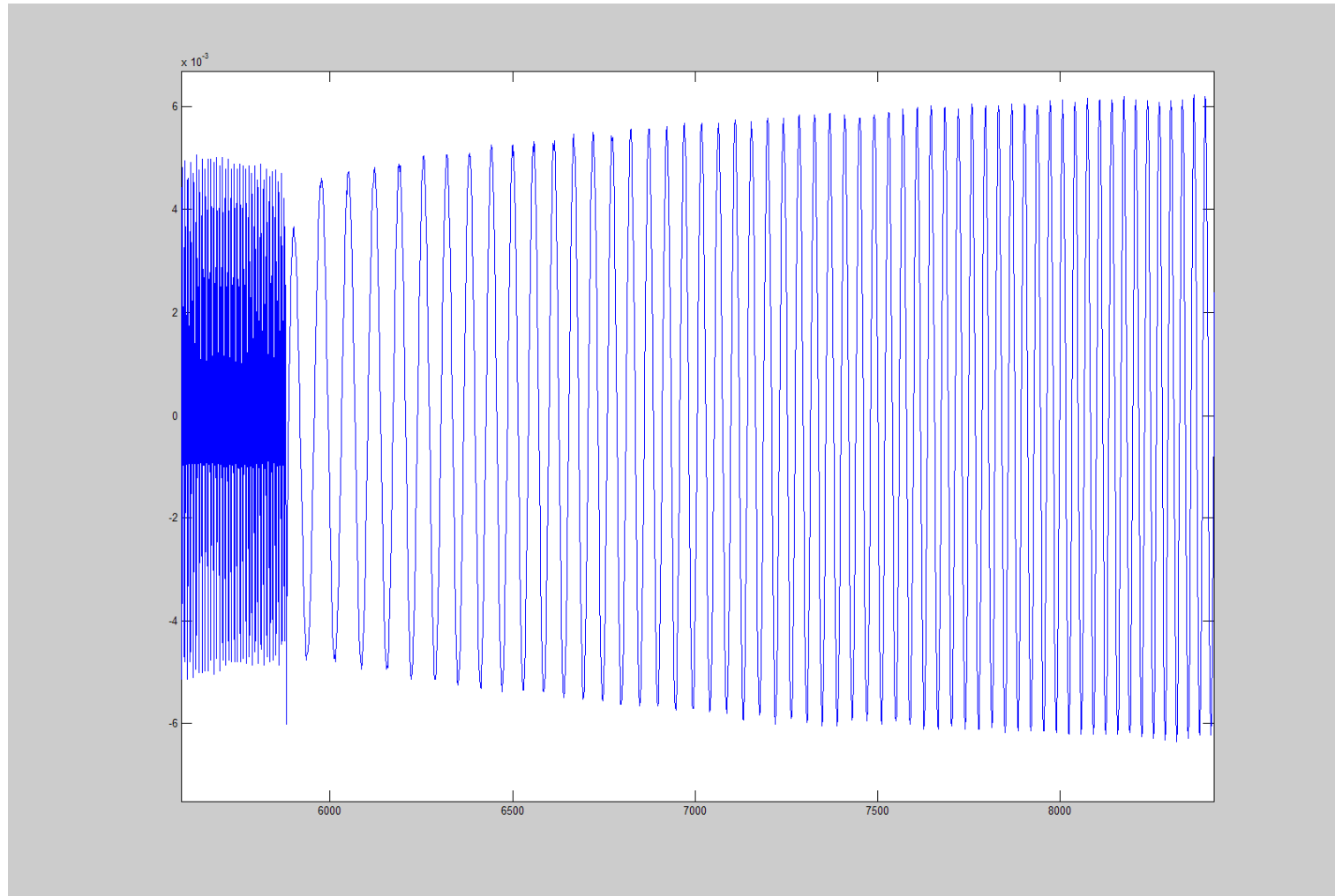
- Implement the filter designed above using either an interrupt or polling to process audio data received from the codec.
- Perform all filter processing using single precision (i.e., use the datatype “float”).
- The FIR filter implementation has the following additional requirements:
 - If SW1 is depressed, light LED1 and include the FIR filter loopback path. If the switch is not depressed, do not include the FIR filter. This will allow the processing system to run in a “loopback mode” and a “filter mode.”
 - If SW2 is depressed, light LED2 and add a linearly chirped DDS signal into the codec input, prior to your filter. The DDS signal should chirp from 100Hz to 24kHz over a 12 second period.
 - Light LED3 for the entire time period you are processing a new input sample.
 - Rather than shifting samples, your program should implement a circular buffer manually (i.e., don’t use any assembly, including the built-in support for circular addressing – perform the address calculations in C).
 - Unless instructed otherwise, compile the implementation with the symbolic debug support and no optimization.

DDS: Agile Frequency Generator

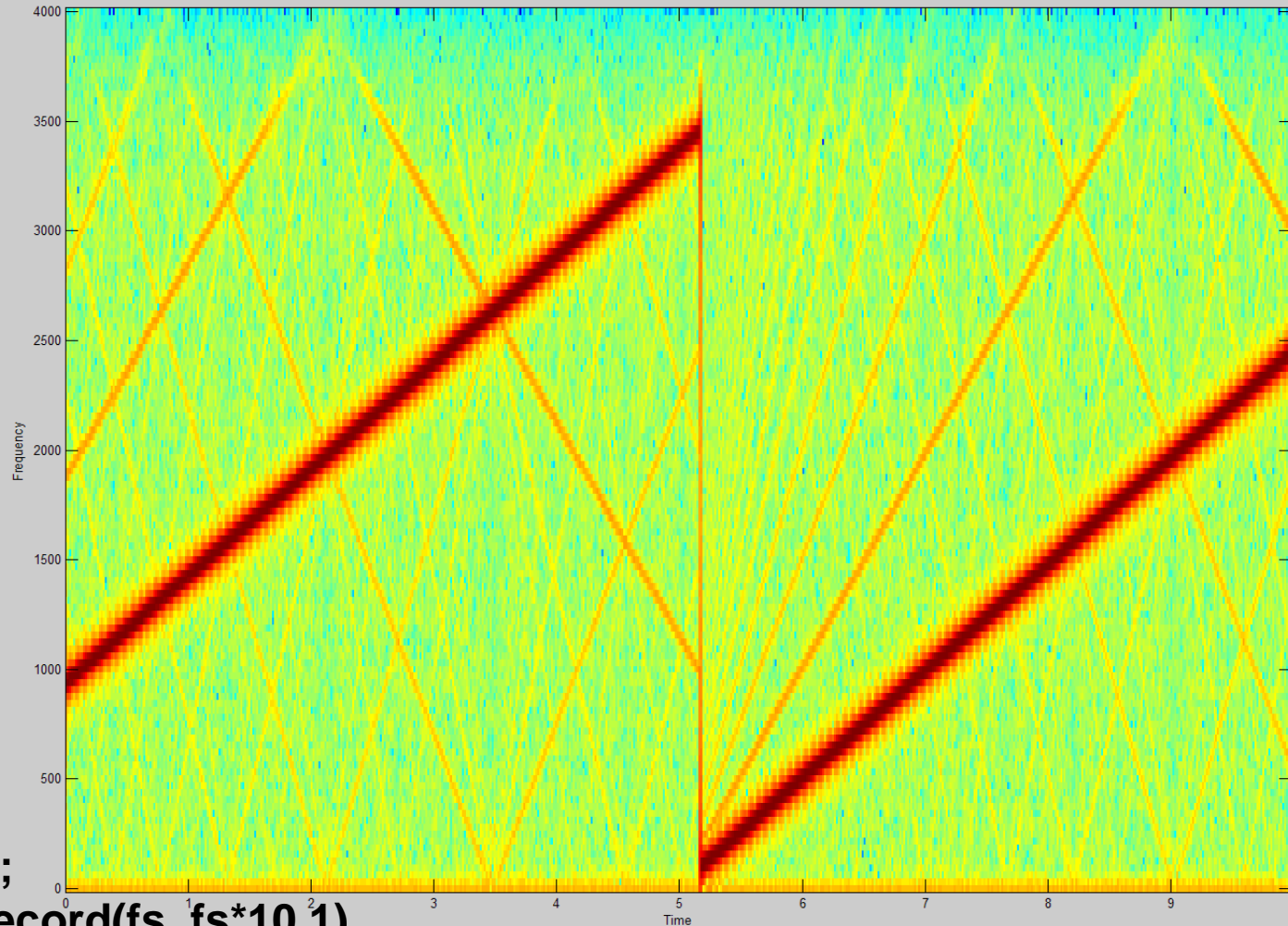
- By linearly increasing the phase step of our phase step at the appropriate rate, we can create a linearly chirped output signal with coherent phase
 - Note that by altering the rate at which we update the step, we can create different chirps (such as a quadratically chirping signal, etc.)
- For example, in our MATLAB simulation we could code this as:

```
for i = 1:n
    y(i) = sinTable(ind+1);
    if ~mod(i,16)
        f = f + 1;
    end
    if f > 3500, f = 100, end
    step = (2*pi)*f/fs;
    phase = phase + step;
    if phase > 2*pi, phase = phase - 2*pi;, end
    ind = floor(phase * mult);
End
```

Chirped Signal as Recorded by MATLAB

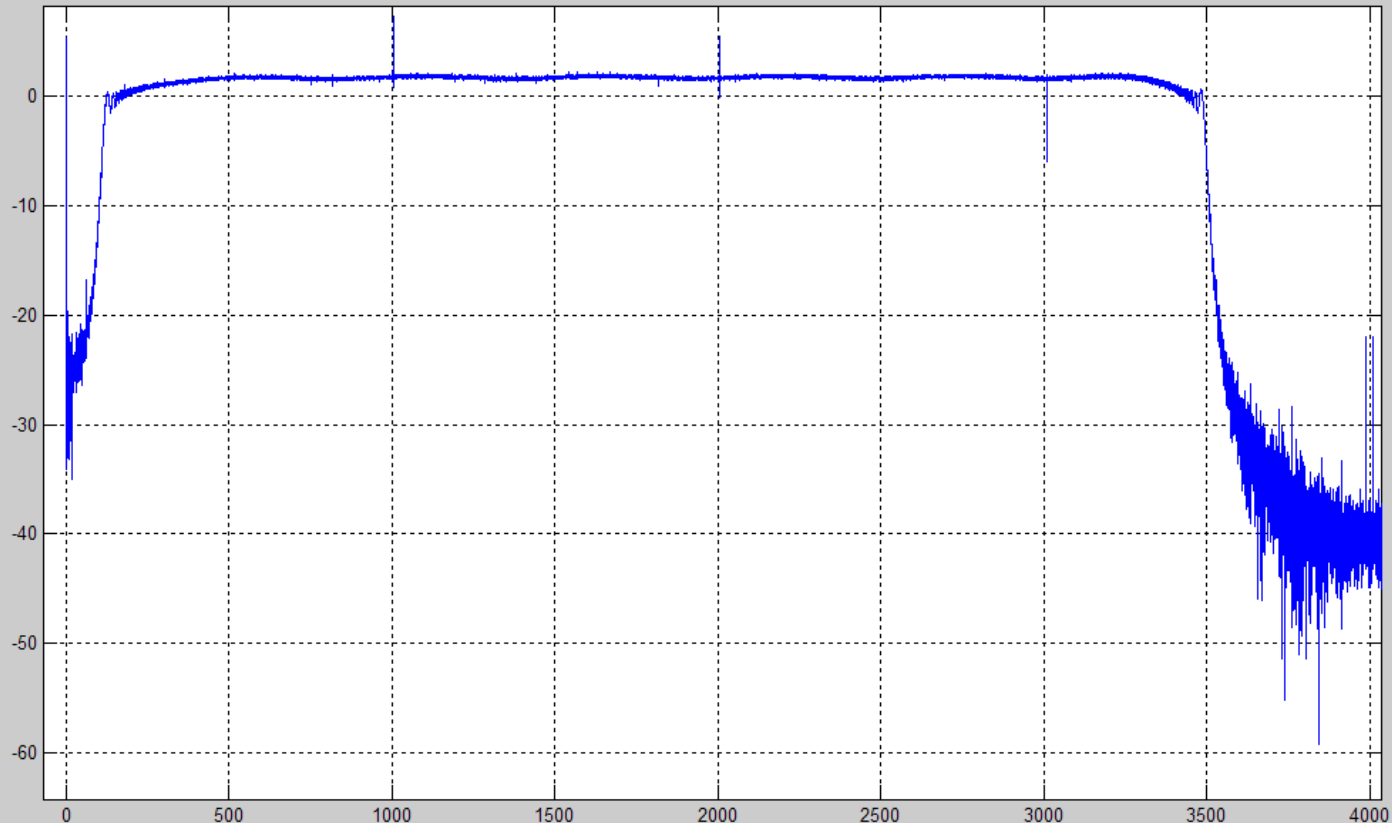


Linearly Chirped DDS



```
fs = 8000;  
Y = wavrecord(fs, fs*10,1)  
specgram(y, 256, fs)
```

Frequency Response of D/A Measured by DDS



Note – this is not your answer, as it is from the old kit, with samplerate = 8k

Buffering Inefficiency/Circular Buffers

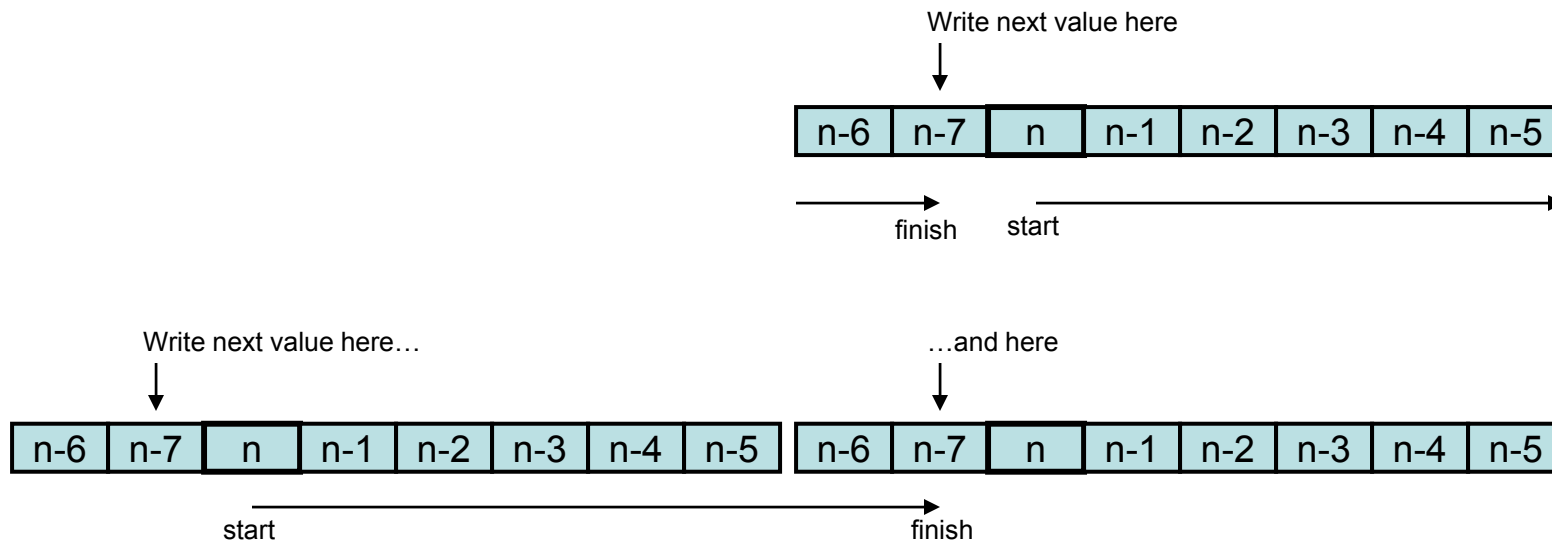
- Note that in CHAISSING's simple code, at every new sample all of the data is shifted over one:

```
for (i = N-1; i > 0; i--) //starting @ end of buffer
    dly[i] = dly[i-1];    //update delays with data move
```

- While this can be implemented efficiently in hardware (as a shift register), it is a slow sequential operation on a processor
- To avoid this it is common to build a “circular buffer” to store the input samples
 - In fact, it is common enough that the C6711 DSP has built in HW support for circular buffering
 - For this lab we will build the circular buffer ourselves, in C

Circular Buffers in C

- To implement the circular buffers in C, we will need to perform the address arithmetic manually, including:
 - Keeping track of the next index to write to
 - Manually “looping” from the end of the circular buffer to the beginning
- This technique can be sped up a number of ways:
 - Only allowing the number of taps to be a power of 2 means that address calculations can avoid the complexity of a modulo operation or a branch
 - Double-buffering the input with two circular buffers means that through “unrolling” the processing can access the memory in a linear fashion
 - Millions of other techniques exist...



Analysis

- Perform the following analysis on the FIR filter implementation:
 - Measure the amplitude response of the analog output. You can do this by including the DDS chirp in the processing chain and excluding the filter.
 - Measure the amplitude response of the filter. You can do this by including the DDS chirp and the filter in the processing chain. Compare with the theoretical response.
 - Measure the amount of time it takes to execute your filter processing (you may include the DDS chirp generation). You can do this by measuring the duty cycle of the LED3 signal using an oscilloscope available in the computer lab.
 - Extend the length of your filter by powers of two to find the maximum number of taps that you can fit into the allotted time. By plotting the amount of time each power of two takes, you can estimate the maximum number of taps that can fit. Note that you can extend the length of your filter by just adding zeros – the DSP takes as long to multiply by zero as by any other number.
 - Redesign the filter using the FDATool for the extended filter length, include the taps in your C implementation, and measure the amplitude response of the filter.
 - Recompile with the highest level of optimization, and re-measure the amount of time it takes to execute your filter processing (with the extended filter).

Programming in C

- How is this different than the C programming that we are perhaps more accustomed to (like Visual C on a PC)?
 - Embedded C environments typically differ from pure ANSI C
 - Data Types
 - Memory Allocation
 - Typically many extensions which allow more control over hardware
 - Aspects of C which do not fit target architecture well are left out.
- Rules by which the C compiler lives, and expects others to live are referred to as the Run-Time-Model
 - i.e. which registers are to be preserved when calling a subroutine
 - How are parameters passed around, in what registers? on stack?
 - Where in memory do certain types of variables go?

C Data Types

Table 7-1. 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	-32 768	32 767
unsigned short	16 bits	Binary	0	65 535
int, signed int	32 bits	2s complement	-2 147 483 648	2 147 483 647
unsigned int	32 bits	Binary	0	4 294 967 295
long, signed long	40 bits	2s complement	-549 755 813 888	549 755 813 887
unsigned long	40 bits	Binary	0	1 099 511 627 775
long long, signed long long	64 bits	2s complement	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits	Binary	0	18 446 744 073 709 551 615
enum	32 bits	2s complement	-2 147 483 648	2 147 483 647
float	32 bits	IEEE 32-bit	1.175 494e-38 ⁽¹⁾	3.40 282 346e+38
double	64 bits	IEEE 64-bit	2.22 507 385e-308 ⁽¹⁾	1.79 769 313e+308
long double	64 bits	IEEE 64-bit	2.22 507 385e-308 ⁽¹⁾	1.79 769 313e+308
pointers, references, pointer to data members	32 bits	Binary	0	0xFFFFFFFF

Programming in C

- In a typical embedded system, we must know ahead of time what is going where.
 - Chances are, you'll run out of space somewhere
 - Ex : linker command file
- Programmer is far more intimately knowledgeable about what memory exists, where it is, and its performance

Memory Allocation

```
/*C6713dsk.cmd Linker command file*/
MEMORY
{
IVECS: org=0h, len=0x220
IRAM:  org=0x00000220, len=0x0002FDE0 /*internal memory*/
SDRAM: org=0x80000000, len=0x00100000 /*external memory*/
FLASH: org=0x90000000, len=0x00020000 /*flash memory*/
}
SECTIONS
{
.EXT_RAM :> SDRAM
.vectors :> IVECS /*in vector file*/
.text :> IRAM
.bss :> IRAM
.cinit :> IRAM
.stack :> IRAM
.systemem :> IRAM
.const :> IRAM
.switch :> IRAM
.far :> IRAM
.cio :> IRAM
.csldata :> IRAM
}
```

FIGURE 1.16. Generic linker command file (C6713dsk.cmd).

Data Storage Definitions

- **Stack**
 - Area of memory typically used for
 - temporary storage within subroutines,
 - local (nonstatic) variables,
 - subroutine parameters, return values (sometimes)
 - Return address storage (call stack)
- **Heap**
 - Pool of memory used for dynamic memory allocation to requestors
 - malloc, calloc allocate memory from this pool, and it is managed by those memory management functions provided by the compiler. Free(xx) frees the memory for use by someone else
- **BSS**
 - Space allocated for storage of global and static variables

Stack and Heap

- **Setting the stack size and heap size**
 - Done in a couple of ways, see student posting to group on this topic
- **Default size is 1024 bytes for each**
 - Large enough for general purpose stuff, but not for allocating many local variables.
 - Example on chalkboard about stack, B15, and how stack “grows towards lesser addresses”
 - Compiler can’t check for stack overflow, since logic of the program itself determines how stack moves
 - Stack size should certainly be large enough for local vars, and
 - You can check for stack overflow
 - Leave a cushion of initialized memory below the stack
 - If stack ever grows into that space, the values of that cushion will be wrong when you are debugging a fault.
- **Heap overflowing is noticeable at runtime, as malloc will return error condition. Note that heap is also very small, as this is not used nearly as often as you might think**
- **Get variables into bss by declaring them “static” or global if necessary**
- **Variables that are only used briefly in a very local scope can be allocated on the stack**

Some C Compiler Extensions

- **Keywords : near, far, cregister, interrupt**
- **Cregister** : (Specifics Later in class) allows you to talk to DSP control registers directly from C
 - Needed for registers that aren't memory mapped
- **Interrupt** : used to specify that a function is an interrupt (and needs such treatment in its code)
 - Preserving registers that are understood by the run-time model to be preserved across interrupts
 - We will have a lab which specifically requires us to use interrupts
- **Near/far** (some other time as well)

Other keywords

- **register** int myvar;
 - The TMS320C6000 C/C++ compiler treats register variables (variables defined with the register keyword) differently, depending on whether you use the -o option. With the -o option, it ignores your register suggestions, and does it the best it can. Without -o
 - If you use the register keyword, you can suggest variables as candidates for allocation into registers.
 - The compiler uses the same set of registers for allocating temporary expression results as it uses for allocating register variables.
 - The compiler attempts to honor all register definitions. If the compiler runs out of appropriate registers, it frees a register by moving its contents to memory. If you define too many objects as register variables, you limit the number of registers the compiler has for temporary expression results. This limit causes excessive movement of register contents to memory.
- **volatile** int *leds_and_switches = 0x90080000
 - Then, when writing led values:
 - *leds_and_switches = 1;

Volatile Example

7.4.6 The volatile Keyword

The compiler analyzes data flow to avoid memory accesses whenever possible. If you have code that depends on memory accesses exactly as written in the C/C++ code, you must use the volatile keyword to identify these accesses. A variable qualified with a volatile keyword is allocated to an uninitialized section (as opposed to a register). The compiler does not optimize out any references to volatile variables.

In the following example, the loop waits for a location to be read as 0xFF:

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

In this example, *ctrl is a loop-invariant expression, so the loop is optimized down to a single-memory read. To correct this, define *ctrl as:

```
volatile unsigned int *ctrl;
```

Here the *ctrl pointer is intended to reference a hardware location, such as an interrupt flag.

- DIP Switches for Lab3 should be defined as this.
 - Anytime that reading or writing has a side-effect, volatile should be used, since compiler might optimize out a read or write if it is not used, and can certainly re-order it.