

Finite Impulse Response

By:

Hyeokho Choi

Finite Impulse Response

By:

Hyeokho Choi

Online:

< <http://cnx.org/content/col10226/1.1/> >

C O N N E X I O N S

Rice University, Houston, Texas

This selection and arrangement of content as a collection is copyrighted by Hyeokho Choi. It is licensed under the Creative Commons Attribution 1.0 license (<http://creativecommons.org/licenses/by/1.0>).

Collection structure revised: February 16, 2004

PDF generated: October 30, 2009

For copyright and attribution information for the modules contained in this collection, see p. 40.

Table of Contents

1 Chapter 1	
1.1 TMS320C6211 Architecture Overview	1
1.2 C62x Assembly Primer II	3
Solutions	18
2 Chapter 2	
2.1 Fixed Point Arithmetic	19
Solutions	25
3 Chapter 3	
3.1 Unit Sample Signal	27
3.2 Discrete-Time Filtering	27
3.3 Filter design by windowing	28
3.4 Parks-McClellan Optimal FIR Filter Design	29
3.5 FIR Filter Design using MATLAB	29
3.6 MATLAB FIR Filter Design Exercise	30
3.7 Assembly Implementation of FIR Filters on TI TMS320C62x	30
3.8 C Language Implementation of FIR Filters on TMS320C62x	32
3.9 Linear Assembly Implementation of FIR Filters on TMS320C62x	32
Solutions	35
4 Supplemental Material	
4.1 Rice DSP Lab Setup	37
4.2 Testing Filters in Rice DSP Lab	38
Solutions	??
Index	39
Attributions	40

Chapter 1

Chapter 1

1.1 TMS320C6211 Architecture Overview¹

1.1.1 Overview of C6211 Architecture

The C62x consists of internal memory, peripherals (serial port, external memory interface, *etc.*), and most importantly, the CPU that has the registers and the functional units for execution of instructions. Figure 1-1 on the next page illustrates the internal structure of the CPU and the relation with the peripherals outside the CPU. Although you don't need to care about the internal architecture of the CPU for compiling and running programs, it is necessary to understand how the CPU fetches and executes the assembly instructions to write a highly optimized assembly program.

We demonstrate the architecture and basic function of each CPU unit through the development of simple assembly language programs.

1.1.1.1 Core DSP operation

In many DSP algorithms, the Sum of Product or Multiply-Accumulate (MAC) operations are very common. A DSP CPU is designed to handle the math-intensive calculations necessary for DSP algorithms. For efficient implementation of the MAC operations, the C6211 CPU has two multipliers and each of them can perform a 16-bit multiplication in each clock cycle. For example, if we want to compute the dot product of two length-40 vectors a_n and x_n , we need to compute $\sum_{n=1}^{40} (a_n x_n)$. (For example, the FIR filtering algorithm is exactly same as this dot product operation.) When a_n and x_n are stored in memory, starting from $n = 1$, we need to compute $a_n x_n$ and add it to y (y is initially 0) and repeat this up to $n = 40$. In the C62x assembly, this MAC operation can be written as

```
MPY .M a,x,prod
ADD .L y,prod,y
```

Ignore `.M` and `.L` for now. Here, `a,x,prod,y` are numbers stored in memory and the instruction `MPY` multiplies two numbers `a` and `x` together and stores the result in `prod`. The `ADD` instruction adds two numbers `y` and `prod` together storing the result back to `y`.

¹This content is available online at <http://cnx.org/content/m10872/2.5/>.

1.1.1.2 Register Files

Where are the numbers stored in the CPU? In C62x, the numbers used in operations are stored in the registers. Because the registers are directly accessible through the data bus of the CPU, accessing the registers are much faster than accessing data in the external memory.

The C62x CPU has two register files consisting of sixteen 32-bit registers each. There are two separate register files (A and B). Each of these files contains sixteen 32-bit registers (A0-A15 for file A and B0-B15 for file B). The general-purpose registers can be used for data, data address pointers, or condition registers.

The general-purpose register files support data ranging in size from 16-bit data through 40-bit fixed-point. Values larger than 32 bits, such as 40-bit long quantities, are stored in register pairs. In a register pair, the 32 LSBs of data are placed in an even-numbered register and the remaining 8 MSBs in the next upper register (which is always an odd-numbered register). In assembly language syntax, a colon between two register names denotes the register pairs, and the odd-numbered register is specified first. For example, A1:A0 represents the register pair consisting of A0 and A1. But you don't need to be concerned with the 40-bit numbers too much. Throughout this course, you will be mostly handling either 16 or 32-bit values stored in a single register. Let's for now focus on file A only. The registers in the register file A are named A0 to A15. Each register can store a 32-bit binary number. The numbers such as `a,x,prod,y` above are stored in these registers. For example, register A0 stores `a`. For now, let's assume we interpret all 32-bit numbers stored in registers as unsigned integer. Therefore, the range of values we can represent is 0 to $2^{32} - 1$. (For representation of real numbers using binary bits, we will learn about the Q format numbers for fixed-point representation of real numbers.) Let's assume the numbers `a,x,prod,y` are in the registers A0,A1,A3,A4, respectively. Then, the above assembly instructions can be written specifically

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

The TI C62x CPU has a load/store architecture. This means that all the numbers must be stored in the registers for being used as operands for the operations for instructions such as `MPY` and `ADD`. The numbers can be read from a memory location to a register (using, for example, `LDW`, `LDB` instructions) or a register can be loaded with a constant value. The content of a register can be stored to a memory location (using, for example, `STW`, `STB` instructions).

In addition to the general-purpose register files, the CPU has a separate register file for the control registers. The control registers are used to control various CPU functions such as addressing mode, interrupts, *etc.* You will learn more about some of the control registers when we learn each individual topic.

1.1.1.3 Functional units

Then, where do the actual operations such as multiplication and addition take place? The C62x CPU has several **functional units** that perform the actual operations. Each register file has 4 functional units named `.M`, `.L`, `.S`, and `.D`. (See Figure 1-1). The 4 functional units connected to the register file A are named `.L1`, `.S1`, `.D1`, and `.M1`. Those connected to the register file B are named `.L2`, `.S2`, `.D2`, and `.M2`. See Figure 1-1. For example, the functional unit `.M1` performs multiplication on the operands that are in register file A. When the CPU executes the `MPY .M1 A0,A1,A3` above, the functional unit `.M1` takes the values stored in A0 and A1, multiply them together and stores the result to A3. The `.M1` in `MPY .M1 A0,A1,A3` indicates that this operation is performed in the `.M1` unit. The `.M1` unit has a 16 bit multiplier and all the multiplications are performed by the `.M1` unit.

Similarly, the `ADD` operation can be executed by the `.L1` unit. The `.L1` can perform all the logical operations such as bitwise AND operation (`AND` instruction) as well as basic addition (`ADD` instruction) and subtraction (`SUB` instruction).

For complete list of instructions executed by each function unit, see Table 3-2 in the handout **TMS320C62x/C64x/C67x Fixed-Point Instruction Set**. We will later learn more about assigning the functional units for assembly instructions.

Exercise 1.1

(Solution on p. 18.)

Read the description of ADD and MPY instructions in the TI manual handed out. Write an assembly program that computes

$$A0*(A1+A2)+A3$$

1.2 C62x Assembly Primer II²

1.2.1 Typical Assembly Operations

1.2.1.1 Loading constants to registers

Quite often you need to load a register with a constant. The C62x instructions you can use for this task are MVK, MVKL, and MVKH. Each of these instructions can load a 16-bit constant to a register. Read and understand the description of these instructions in the manual.

Exercise 1.2

(Solution on p. 18.)

(Loading constants): Write assembly instructions to do the following:

1. Load the 16-bit constant 0xff12 to A1.
2. Load the 32-bit constant 0xabcd45ef to B0.

1.2.1.2 Register moves, zeroing

Contents of one register can be copied to another register by using the MV instruction. There is also the ZERO instruction to set a register to zero. Learn how to use these instructions by reading the appropriate TI manual pages.

1.2.1.3 Loading from memory to registers

Because the C62x processor has the so-called load/store architecture, you must first load up the content of memory to a register to be able to manipulate it. The basic assembly instructions you use for loading are LDB, LDH, and LDW for loading up 8-, 16-, and 32-bit data from memory. (There are some variations to these instructions for different handling of the signs of the loaded values.) Read and understand how these instructions work.

However, to specify the address of the memory location to load from, you need to load up another register (used as an address index) and you can use various **addressing modes** to specify the memory locations in many different ways. The addressing modes is the method by which an instruction calculates the location of an object in memory. The table below lists all the possible different ways to handle the address pointers in C62x CPU. Note the similarity with the C pointer manipulation.

²This content is available online at <<http://cnx.org/content/m11051/2.3/>>.

Syntax	Memory address accessed	Pointer modification
*R	R	None
+++R	R	Preincrement
*-R	R	Predecrement
*R++	R	Postincrement
*R-	R	Postdecrement
++R[disp]	R+disp	None
*-R[disp]	R+disp	None
+++R[disp]	R+disp	Preincrement
*-R[disp]	R+disp	Predecrement
*R++[disp]	R+disp	Postincrement
*R-[disp]	R+disp	Postdecrement

Table 1.1

The [disp] specifies the number of elements in word, halfword, or byte, depending on the instruction type and it can be either **5-bit constant** or a **register**. The increment/decrement of the index registers are also in terms of the number of bytes in word, halfword or byte. The addressing modes with displacements are useful when a block of memory locations is accessed. Those with automatic increment/decrement are useful when a block is accessed consecutively to implement a buffer, for example, to store signal samples to implement a digital filter.

Exercise 1.3*(Solution on p. 18.)*

(Load from memory): Assume the following values are stored in memory addresses:

```

100h fe54 7834h
104h 3459 f34dh
108h 2ef5 7ee4h
10ch 2345 6789h
110h ffff eeddh
114h 3456 787eh
118h 3f4d 7ab3h

```

Suppose A10 = 0000 0108h. Find the contents of A1 and A10 after executing the each of the following instructions.

1. LDW .D1 *A10, A1
2. LDH .D1 *A10, A1
3. LDB .D1 *A10, A1
4. LDW .D1 *-A10[1], A1
5. LDW .D1 ++A10[1], A1
6. LDW .D1 ++A10[2], A1
7. LDB .D1 ++A10[2], A1
8. LDW .D1 +++A10[1], A1
9. LDW .D1 *-A10[1], A1
10. LDB .D1 +++A10[1], A1
11. LDB .D1 *-A10[1], A1

12. LDW .D1 *A10++[1], A1
13. LDW .D1 *A10-[1], A1

1.2.1.4 Storing data to memory

Storing the register contents uses the same addressing modes. The assembly instructions used for storing are STB, STH, and STW. Read and understand these instructions in the TI manual.

Exercise 1.4 *(Solution on p. 18.)*

(Storing to memory): Write assembly instructions to store 32-bit constant 53fe 23e4h to memory address 0000 0123h.

Sometimes, it becomes necessary to access part of the data stored in memory. For example, if you store the 32-bit word 0x11223344 at memory location 0x8000, the four bytes having addresses location 0x8000, location 0x8001, location 0x8002, and location 0x8003 contain the value 0x11223344. Then, if I read the byte data at memory location 0x8000, what would be the byte value to be read?

The answer depends on the **endian mode** of the memory system. In the **little endian mode**, the lower memory addresses contain the LSB part of the data. Thus, the bytes stored in the four byte addresses will be as shown in Table 1.2.

0x8000	0x44
0x8001	0x33
0x8002	0x22
0x8003	0x11

Table 1.2

In the **big endian mode**, the lower memory addresses contain the MSB part of the data. Thus, we have

0x8000	0x11
0x8001	0x22
0x8002	0x33
0x8003	0x44

Table 1.3

In this course, we use the little endian mode by default and all the lab programming must assume the little endian mode.

Exercise 1.5 *(Solution on p. 18.)*

(Little endian mode): What will be the value in A0 after executing the following assembly instructions? (functional unit specifications were omitted.)

1. MVKL 0x80000000, A10
2. MVKH 0x80000000, A10
3. MVKL 0x12345678, A9
4. MVKH 0x12345678, A9
5. STW A9, *A10
6. LDB **A10[2], A0

What will be the value in A0 if the system uses the big endian mode?

In fact, the above addressing method describes the so-called **linear** addressing mode (default upon reset), where the offset or increment/decrement of pointers occur without bound. There is a **circular** addressing modes that can handle a finite size buffer efficiently. You will implement circular buffers for the FIR filtering algorithm in the FIR filtering experiments later.

In the C62x CPU, it takes exactly one CPU clock cycle to execute each instruction. However, the instructions such as LDW need to access the slow external memory and the results of the load are not available immediately at the end of the execution. This **delay** of the execution results is called **delay slots**.

Example 1.1

For example, let's consider loading up the content of memory content at address pointed by A10 to A1 and then moving the loaded data to A2. You might be tempted to write simple 2 line assembly code as follows:

```
1    LDW    .D1    *A10, A1
2    MV     .D1    A1,A2
```

What is wrong with the above code? The result of the LDW instruction is not available immediately after LDW is executed. As a consequence, the MV instruction does not copy the desired value of A1 to A2. To prevent this undesirable execution, we need to make the CPU wait until the result of the LDW instruction is correctly loaded to A1 before executing the MV instruction. For load instructions, we need extra 4 clock cycles until the load results are valid. To make the CPU wait for 4 clock cycles, we need to insert 4 NOP (no operations) instructions between LDW and MV. Each NOP instruction makes the CPU idle for one clock cycle. The resulting code will be like this:

```
1    LDW    .D1    *A10, A1
2    NOP
3    NOP
4    NOP
5    NOP
6    MV     .D1    A1,A2
```

or simply you can write

```
1    LDW    .D1    *A10, A1
2    NOP    4
3    MV     .D1    A1,A2
```

Then, why didn't the designer of the CPU make such that LDW instruction takes 5 clock cycles to begin with, rather than let the programmer insert 4 NOPs? The answer is that you can insert other instructions other than NOPs as far as those instructions do not use the result of the LDW instruction above. By doing this, the CPU can execute additional instructions while waiting for the result of the LDW instruction to be valid, greatly reducing the total execution time of the entire program.

1.2.1.5 More on instructions with delay slots

The Table 3-5 in TI's instruction set description shows the execution of the instructions with delay slots in more detail. The instructions with delay slots are multiply (MPY, 1 delay slot), the load (LDB, LDW etc. 4 delay slots) instructions, and the branch (B, 5 delay slots) instruction.

The **functional unit latency** indicates for how many clock cycles each instructions actually use a functional unit. All C62x instructions have 1 functional unit latency, meaning that each functional unit is ready to execute the next instruction after 1 clock cycle regardless of the delay slots of the instructions. Therefore, the following instructions are valid:

```
1    LDW    .D1    *A10, A4
2    ADD    .D1    A1,A2,A3
```

Although the first LDW instruction do not load the A4 register correctly while the ADD is executed, the D1 functional unit becomes available in the clock cycle right after the one in which LDW is executed.

To clarify the execution of instructions with delay slots, let's think of the following example of LDW instruction. Let's assume A10 = 0x0100 A2=1, and your intent is loading A9 with the 32-bit word at the address 0x0104. The 3 MV instructions are not related to the LDW instruction. They do something else.

```
1    LDW    .D1    *A10++[A2], A9
2    MV     .L1    A10, A8
3    MV     .L1    A1, A10
4    MV     .L1    A1, A2
5    ...
```

We can ask several interesting questions at this point:

1. What is the value loaded to A8? That is, in which clock cycle, the address pointer is updated?
2. Can we load the address offset register A2 before the LDW instruction finishes the actual loading?
3. Is it legal to load to A10 before the first LDW finishes loading the memory content to A9? That is, can we change the address pointer before the 4 delay slots elapse?

Here are the answers:

1. Although it takes extra 4 clock cycles for the LDW instruction to load the memory content to A9, the address pointer and offset registers (A10 and A2) are read and updated in the clock cycle the LDW instruction is issued. Therefore, in line 2, A8 is loaded with the updated A10, that is A10 = A8 = 0x104.
2. Because the LDW reads the A10 and A2 registers in the first clock cycle, you are free to change these registers and do not affect the operation of the first LDW.
3. This was already answered above.

Similar theory holds for MPY and B (when using a register as a branch address) instructions. The MPY reads in the source values in the first clock cycle and loads the multiplication result after the 2nd clock cycle. For B, the address pointer is read in the first clock cycle, and the actual branching occurs after the 5th clock cycle. Thus, after the first clock cycle, you are free to modify the source or the address pointer registers. For more details, refer Table 3-5 in the instruction set description or read the description of the individual instruction.

1.2.1.6 Addition, Subtraction and Multiplication

There are several instructions for addition, subtraction and multiplication on C62x CPU. The basic instructions are ADD, SUB, and MPY. Learn about these instructions in the TI manual. ADD and SUB have 0 delay slots (meaning the results of operation are immediately available), but the MPY has 1 delay slot (the result of multiplication is valid after additional 1 clock cycle).

Exercise 1.6

(Solution on p. 18.)

(Add, subtract, and multiply): Write an assembly program to compute (0000 ef35h + 0000 33dch - 0000 1234h) * 0000 0007h

1.2.1.7 Branching and conditional operations

Often you need to control the flow of the program execution by branching to another block of code. The **B** instruction does the job in the C62x CPU. The address of the branch can be specified either by displacement or stored in a register to be used by the **B** instruction. Read and understand the **B** instruction in the manual. The **B** instruction has 5 delay slots, meaning that the actual branch occurs in the 5th clock cycle after the instruction is executed.

In many cases, depending on the result of previous operations, you execute the branch instruction conditionally. For example, to implement a loop, you decrement the loop counter by 1 each time you run a set of instructions and whenever the loop counter is not zero, you need to branch to the beginning of the code block to iterate the loop operations. In C62x CPU, this conditional branching is implemented using the **conditional operations**. Although **B** may be the instruction implemented using conditional operations most often, all instructions in C62x can be conditional.

Conditional instructions are represented in code by using square brackets, [], surrounding the condition register name. For example, the following **B** instruction is executed only if **B0** is nonzero:

```
1    [B0]    B    .L1    A0
```

To execute an instruction conditionally when the condition register is zero, we use **!** in front of the register. For example, the **B** instruction is executed when **B0** is zero.

```
1    [!B0]   B    .L1    A0
```

Not all registers can be used as the condition registers. In C62x CPU, the registers that can be tested in conditional operations are **B0**, **B1**, **B2**, **A1**, **A2**.

Exercise 1.7

(Simple loop): Write an assembly program computing the summation $\sum_{n=1}^{100} n$ by implementing a simple loop. *(Solution on p. 18.)*

1.2.1.8 Logical operations and bit manipulation

The logical operations and bit manipulations are accomplished by the **AND**, **OR**, **XOR**, **CLR**, **SET**, **SHL**, and **SHR** instructions. Read and understand the operations of these instructions.

1.2.1.9 Other assembly instructions

Other useful instructions include **IDLE** and compare instructions such as **CMPEQ** *etc.* Read and understand the operations of these instructions.

1.2.1.10 C62x instruction set summary

The set of instructions that can be performed in each functional unit is as follows (See Table 1.4: **.S** Unit, Table 1.5: **.L** Unit, Table 1.6: **.D** Unit and Table 1.7: **.M** Unit). Please refer to *TMS320C62x/C67x CPU and Instruction Set Reference Guide* for detailed description of each instruction.

.S Unit

Instruction	Description
ADD(U)	signed or unsigned integer addition without saturation
ADDK	integer addition using signed 16-bit constant
ADD2	two 16-bit integer adds on upper and lower register halves
B	branch using a register
CLR	clear a bit field
EXT	extract and sign-extend a bit field
MV	move from register to register
MVC	move between the control file and the register file
MVK	move a 16-bit constant into a register and sign extend
MVKH	move 16-bit constant into the upper bits of a register
NEG	negate (pseudo-operation)
NOT	bitwise NOT
OR	bitwise OR
SET	set a bit field
SHL	arithmetic shift left
SHR	arithmetic shift right
SSHL	shift left with saturation
SUB(U)	signed or unsigned integer subtraction without saturation
SUB2	two 16-bit integer integer subs on upper and lower register halves
XOR	exclusive OR
ZERO	zero a register (pseudo-operation)

Table 1.4

.L Unit

Instruction	Description
ABS	integer absolute value with saturation
<i>continued on next page</i>	

ADD(U)	signed or unsigned integer addition without saturation
AND	bitwise AND
CMPEQ	integer compare for equality
CMPGT(U)	signed or unsigned integer compare for greater than
CMPLT(U)	signed or unsigned integer compare for less than
LMBD	leftmost bit detection
MV	move from register to register
NEG	negate (pseudo-operation)
NORM	normalize integer
NOT	bitwise NOT
+OR	bitwise OR
SADD	integer addition with saturation to result size
SAT	saturate a 40-bit integer to a 32-bit integer
SSUB	integer subtraction with saturation to result size
SUBC	conditional integer subtraction and shift - used for division
XOR	exclusive OR
ZERO	zero a register (pseudo-operation)

Table 1.5

.D Unit

Instruction	Description
ADD(U)	signed or unsigned integer addition without saturation
ADDAB (B/H/W)	integer addition using addressing mode
LDB (B/H/W)	load from memory with a 15-bit constant offset
MV	move from register to register
STB (B/H/W)	store to memory with a register offset or 5-bit unsigned constant offset
SUB(U)	signed or unsigned integer subtraction without saturation
<i>continued on next page</i>	

SUBAB (B/H/W)	integer subtraction using addressing mode
ZERO	zero a register (pseudo-operation)

Table 1.6

.M Unit

Instruction	Description
MPY (U/US/SU)	signed or unsigned integer multiply 16lsb*16lsb
MPYH (U/US/SU)	signed or unsigned integer multiply 16msb*16msb
MPYLH	signed or unsigned integer multiply 16lsb*16msb
MPYHL	signed or unsigned integer multiply 16msb*16lsb
SMPY (HL/LH/H)	integer multiply with left shift and saturation

Table 1.7

1.2.2 Useful assembler directives

Other than the CPU instruction set, there are special commands to the assembler that direct the assembler to do various jobs when assembling the code. You should learn about some of these **assembler directives** to be able to write an assembly program. There are useful assembler directives you can use to let the assembler know various settings, such as `.set`, `.macro`, `.endm`, `.ref`, `.align`, `.word`, `.byte`, `.include`.

The `.set` directive defines a symbolic name. For example, you can have

```
1    count    .set    40
```

Then, the assembler replaces each occurrence of `count` with 40.

You have already seen how the `.ref` directive is used to declare symbolic names defined in another file. It is similar to the `extern` declaration in C.

The `.space` directive reserves a memory space with specified number of bytes. For example, you can have

```
1    buffer    .space    128
```

to define a buffer of size 128 bytes. The symbol `buffer` has the address of the first byte reserved by `.space`. The `.bss` directive is similar to `.space`, but the label has the address of the last byte reserved.

To put a constant value in the memory, you can use `.byte`, `.word`, *etc.* If you have

```
1    const1    .word    0x1234
```

the assembler places the word constant 0x1234 at a memory location and `const1` has the address of the memory location. `.byte` *etc.* works similarly.

Sometimes you need to place your data or code at a specific memory address boundaries such as word, halfword, *etc.* You can use the `.align` directive to do this. For example, if you have

```

1          .align    4
2  buffer  .space    128
3          ...

```

Then, the first address of the reserved 128 bytes is at the word boundary in memory, that is the 2 LSBs of the address (in binary) are 0. Similarly, for half-word alignment, you should have `.align` directive to do this. For example, if you have

```

1          .align    2
2  buffer  .space    128
3          ...

```

The `.include` directive is used to read the source lines from another file. If you have

```

1          .include  'other.asm'

```

will input the lines in `other.asm` at this location. This is useful when working with multiple files. Instead of making a project having multiple files, you can simply include these different files in one file.

Other assembler directives include `.end`, *etc.* You will learn about the macro directives `.macro`, `.endm` later.

How do you write comments in your assembly program? Anything that follows `;` is considered as a comment and ignored by the assembler. For example,

```

1          ; this is a comment
2          ADD      .L1    A1,A2,A3      ;add a1 and a2

```

1.2.3 Assigning functional units

Each instruction has particular functional units that can execute it. For a complete list of the instructions that can be executed in each functional unit, see Table 3-2 in the instruction set manual. Note that some instructions can be executed by several different functional units.

shows how data and addresses can be transferred between the registers, functional units and the external memory. If you observe carefully, the destination path (marked as **dst**) going out of the `.L1`, `.S1`, `.M1` and `D1` units are connected to the register file A.

NOTE: This means that any instruction with one of the A registers as destination (the result of operation is stored in one of A registers) should be executed in one of these 4 functional units.

For the same reason, if the instructions have B registers as destination, the `.L2`, `.S2`, `.M2` and `D2` units should be used.

Therefore if you know the instruction and the destination register, you should be able to assign the functional unit to it.

Exercise 1.8

(Solution on p. 18.)

(Functional units): List all the functional units you can assign to each of these instructions:

1. `ADD .?? A0,A1,A2`

2. B .?? A1
3. MVKL .?? 000023feh, B0
4. LDW .?? *A10, A3

If you look at again, each functional unit must receive one of the source data from the corresponding register file. For example, look at the following assembly instruction:

```
1    ADD    .L1    A0,B0,A1
```

The .L1 unit gets data from A0 (this is natural) and B0 (this is not) and stores the result in A1 (this is a must). The data path through which the content of B0 is conveyed to the .L1 unit is called **1X cross path**. When this happens, we add x to the functional unit to designate the cross path:

```
1    ADD    .L1x   A0,B0,A1
```

Similarly the data path from register file B to the .M2, .S2 and .L2 units are called 2X cross path.

Exercise 1.9

(Solution on p. 18.)

(Cross path): List all the functional units that can be assigned to each of the instruction:

1. ADD .??? B0,A1,B2
2. MPY .??? A1,B2,A4

In fact, when you write an assembly program, you can omit the functional unit assignment altogether. The assembler figures out the available functional units and properly assigns them. However, manually assigned functional units help you to figure out where the actual execution takes place and how the data move around between register files and functional units. This is particularly useful when you put multiple instructions in parallel. We will learn about the parallel instructions later on.

1.2.4 Writing the inner product program

Now you should know enough about C62x assembly to implement the inner product algorithm to compute

$$y = \sum_{n=1}^{10} (a_n \times x_n)$$

Exercise 1.10

(Solution on p. 18.)

(Inner product): Write the complete inner product assembly program to compute

$$y = \sum_{n=1}^{10} (a_n \times x_n)$$

where a_n and x_n take the following values:

```
a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, a }
x[] = { f, e, d, c, b, a, 9, 8, 7, 6 }
```

The a_n and x_n values must be stored in memory and the inner product is computed by reading the memory contents.

1.2.5 Pipeline, Delay slots and Parallel instructions

When an instruction is executed, it takes several steps, which are fetching, decoding, and execution. If these steps are done one at a time for each instruction, the CPU resources are not fully utilized. To increase the throughput, CPUs are designed to be pipelined, meaning that the foregoing steps are carried out at the same time.

On the C6x processor, the instruction fetch consists of 4 phases; generate fetch address (F1), send address to memory (F2), wait for data (F3), and read opcode from memory (F4). Decoding consists of 2 phases; dispatching to functional units (D1) and decoding (D2). The execution step may consist of up to 6 phases (E1 to E6) depending on the instructions. For example, the multiply (MPY) instructions has 1 delay resulting in 2 execution phases. Similarly, load (LDx) and branch (B) instructions have 4 and 5 delays respectively.

When the outcome of an instruction is used by the next instruction, an appropriate number of NOPs (no operation or delay) must be added after multiply (one NOP), load (four NOPs, or NOP 4), and branch (five NOPs, or NOP 5) instructions in order to allow the pipeline to operate properly. Otherwise, before the outcome of the current instruction is available (which is to be used by the next instruction), the next instructions are executed by the pipeline, generating undesired results. The following code is an example of pipelined code with NOPs inserted:

```

1           MVK    40, A2
2   loop:   LDH    *A5++, A0
3           LDH    *A6++, A1
4           NOP    4
5           MPY    A0, A1, A3
6           NOP
7           ADD    A3, A4, A4
8           SUB    A2, 1, A2
9   [A2]    B      loop
10          NOP    5
11          STH    A4, *A7

```

In line 4, we need 4 NOPs because the A1 is loaded by the LDH instruction in line 3 with 4 delays. After 4 delays, the value of A1 is available to be used in the MPY A0, A1, A3 in line 5. Similarly, we need 5 delays after the [A2] B loop instruction in line 9 to prevent the execution of STH A4, *A7 before branching occurs.

The C6x Very Large Instruction Word (VLIW) architecture, several instructions are captured and processed simultaneously. This is referred to as a Fetch Packet (FP). This Fetch Packet allows C6x to fetch eight instructions simultaneously from on-chip memory. Among the 8 instructions fetched at the same time, multiple of them can be executed at the same time if they do not use same CPU resources at the same time. Because the CPU has 8 separate functional units, maximum 8 instructions can be executed in parallel, although the type of parallel instructions are limited because they must not conflict each other in using CPU resources. In assembly listing, parallel instructions are indicated by double pipe symbols (||). When writing assembly code, by designing code to maximize parallel execution of instructions (through proper functional unit assignments, etc.) the execution cycle of the code can be reduced.

1.2.6 Parallel instructions and constraints

We have seen that C62x CPU has 8 functional units. Each assembly instruction is executed in one of these 8 functional units, and it takes exactly one clock cycle for the execution. Then, while one instruction is being executed in one of the functional units, what are other 7 functional units doing? Can other functional units execute other instructions at the same time?

The answer is YES. Thus, the CPU can execute maximum 8 instructions in each clock cycle. The instructions executed in the same clock cycle are called **parallel instructions**. Then, what instructions can

be executed in parallel? A short answer is: as far as the parallel instructions do not use the same resource of the CPU, they can be put in parallel. For example, the following two instructions do not use the same CPU resource and they can be executed in parallel.

```
1      ADD    .L1    A0,A1,A2
2  ||    ADD    .L2    B0,B1,B2
```

1.2.6.1 Resource constraints

Then, what are the constraints on the parallel instructions? Let's look at the resource constraints in more detail.

1.2.6.1.1 Functional unit constraints

This is simple. Each functional unit can execute only one instruction per each clock cycle. In other words, instructions using the same functional unit cannot be put in parallel.

1.2.6.1.2 Cross paths constraints

If you look at the data path diagram of the C62x CPU, there exists only one cross path from B register file to the L1, M1 and S1 functional units. This means the cross path can be used only once per each clock cycle. Thus, the following parallel instructions are invalid because the 1x cross path is used for both instructions.

```
1      ADD    .L1x   A0,B1,A2
2  ||    MPY    .M1x   A5,B0,A3
```

The same rule holds for the 2x cross path from the A register file to the L2, M2 and S2 functional units.

1.2.6.1.3 Loads and Stores constraints

The D units are used for load and store instructions. If you examine the C62x data path diagram, the addresses for load/store can be obtained from either A or B side using the multiplexers connecting crisscross to generate the addresses DA1 and DA2. Thus, the instructions such as

```
1      LDW    .D2    *B0, A1
```

is valid. **The functional unit must be on the same side as the address source register** (address index in B0 and therefore D2 above), because D1 and D2 units must receive the addresses from A and B sides, respectively.

Another constraint is that while loading a register in one register file from memory, you cannot simultaneously store a register in the same register file to memory. For example, the following parallel instructions are invalid:

```
1      LDW    .D1    *A0, A1
2  ||    STW    .D2    A2, *B0
```

1.2.6.1.4 Constraints on register reads

You cannot have more than **four** reads from the same register in each clock cycle. Thus, the following is invalid:

```

1      ADD    .L1    A1, A1, A2
2  ||    MPY    .M1    A1, A1, A3
3  ||    SUB    .D1    A1, A4, A5

```

1.2.6.1.5 Constraints on register writes

A register cannot be written to more than once in a single clock cycle. However, note that the actual writing to registers may not occur in the same clock cycle during which the instruction is executed. For example, the MPY instruction writes to the destination register in the next clock cycle. Thus, the following is valid:

```

1      ADD    .L1    A1, A1, A2
2  ||    MPY    .M1    A1, A1, A2

```

The following two instructions (not parallel) are invalid (why?):

```

1      MPY    .M1    A1, A1, A2
2      ADD    .L1    A3, A4, A2

```

Some of these write conflicts are very hard to detect and not detected by the assembler. Extra caution should be exercised with the instructions having nonzero delay slots.

1.2.7 Ad-Hoc software pipelining

At this point, you might have wondered why the C62x CPU allows parallel instructions and generate so much headache with the resource constraints, especially with the instructions with delay slots. And, why not just make the MPY instruction take 2 clock cycles to execute so that we can always use the multiplied result after issuing it?

The reason is that by executing instructions in parallel, we can reduce the total execution time of the program. A well-written assembly program executes as many instructions as possible in each clock cycle to implement the desired algorithm.

The reason for allowing delay slots is that although it takes 2 clock cycles for an MPY instruction generate the result, we can execute another instruction while waiting for the result. This way, you can reduce the clock cycles wasted while waiting for the result from slow instructions, thus increasing the overall execution speed.

However, how can we put instructions in parallel? Although there's a systematic way of doing it (we will learn a bit later), at this point you can try to restructure your assembly code to execute as many instructions as possible in parallel. And, you should try to execute other instructions in the delay slots of those instructions such as MPY, LDW, *etc.*, instead of inserting NOPs to wait the instructions produce the results.

Exercise 1.11

(Solution on p. 18.)

(parallel instructions): Modify your assembly program for the inner product computation in the previous exercise to use parallel instructions as much as possible. Also, try to fill the delay slots

as much as possible. Using the code composer's profiling, compare the clock cycles necessary for executing the modified program. How many clock cycles could you save?

Solutions to Exercises in Chapter 1

Solution to Exercise 1.1 (p. 3)

solution here

Solution to Exercise 1.2 (p. 3)

Intentionally left blank.

Solution to Exercise 1.3 (p. 4)

Intentionally left blank.

Solution to Exercise 1.4 (p. 5)

Intentionally left blank.

Solution to Exercise 1.5 (p. 5)

Intentionally left blank.

Solution to Exercise 1.6 (p. 7)

Intentionally left blank.

Solution to Exercise 1.7 (p. 8)

Intentionally left blank.

Solution to Exercise 1.8 (p. 12)

Intentionally left blank.

Solution to Exercise 1.9 (p. 13)

Intentionally left blank.

Solution to Exercise 1.10 (p. 13)

Intentionally left blank.

Solution to Exercise 1.11 (p. 16)

Intentionally left blank.

Chapter 2

Chapter 2

2.1 Fixed Point Arithmetic¹

2.1.1 Fixed-point arithmetic

This handout explains how numbers are represented in the fixed point TI C6211 DSP processor. Because hardware can only store and process **bits**, all the numbers must be represented as a collection of bits. Each bit represents either "0" or "1", hence the number system naturally used in microprocessors is the binary system. This handout explains how numbers are represented and processed in DSP processors for implementing DSP algorithms.

2.1.1.1 How numbers are represented

A collection of N binary digits (bits) has 2^N possible states. This can be seen from elementary counting theory, which tells us that there are two possibilities for the first bit, two possibilities for the next bit, and so on until the last bit, resulting in

$$2 \times 2 \times 2 \cdots = 2^N$$

possibilities or states. In the most general sense, we can allow these states to represent anything conceivable. The point is that there is no meaning inherent in a binary word, although most people are tempted to think of them as positive integers. However, the meaning of an N -bit binary word depends entirely on its **interpretation**.

2.1.1.1.1 Unsigned integer representation

The **natural binary** representation interprets each binary word as a positive integer. For example, we interpret an 8-bit binary word

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

as an integer

$$x = b_72^7 + b_62^6 + \cdots + b_12 + b_0 = \sum_{i=0}^7 (2^i b_i)$$

This way, an N -bit binary word corresponds to an integer between 0 and $2^N - 1$. Conversely, all the integers in this range can be represented by an N -bit binary word. We call this interpretation of binary words **unsigned integer** representation, because each word corresponds to a positive (or unsigned) integer.

We can add and multiply two binary words in a straightforward fashion. Because all the numbers are positive, the results of addition or multiplication are also positive.

¹This content is available online at <<http://cnx.org/content/m11054/2.2/>>.

However, the result of adding two N -bit words in general results in an $N + 1$ bits. When the result cannot be represented as an N -bit word, we say that an **overflow** has occurred. In general, the result of multiplying two N -bit words is a $2N$ bit word. Note that as we multiply numbers together, the number of necessary bits increases indefinitely. This is undesirable in DSP algorithms implemented on hardware. So, later (Section 2.1.1.1.3: Fractional representation) we will introduce the fractional interpretation of binary words, to overcome this problem.

Another problem of the unsigned integer representation is that it can only represent positive integers. To represent negative values, naturally we need a different interpretation of binary words, and we introduce the **two's complement** representation and corresponding operations to implement arithmetic on the numbers represented in the two's complement format.

2.1.1.1.2 Two's complement integer representation

Using the natural binary representation, an N -bit word can represent integers from 0 to $2^N - 1$. However, to represent negative numbers as well as positive integers, we can use the **two's complement** representation. In 2's complement representation, an N -bit word represents integers from $(-2)^{N-1}$ to $2^{N-1} - 1$.

For example, we interpret an 8-bit binary word

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

as an integer

$$x = -(b_72^7) + b_62^6 + \cdots + b_12 + b_0 = -(b_72^7) + \sum_{i=0}^6 (2^i b_i)$$

in the 2's complement representation, and x ranges from -128 ($-(2^7)$) to 127 ($2^7 - 1$). Several examples:

binary	decimal
00000000	0
00000001	1
01000000	64
01111111	127
10000000	-128
10000001	-127
11000000	-64
11111111	-1

Table 2.1

When x is a positive (negative) number in 2's complement format, $-x$ can be found by inverting each bit and adding 1. For example, 01000000_2 is 64 in decimal and -64 is found by first inverting the bits to obtain 10111111_2 and adding 1, thus -64 is 11000000_2 as shown in the above table. Because the MSB indicates the sign of the number represented by the binary word, we call this bit the **sign bit**. If the sign bit is 0, the word represents positive number, while negative numbers have 1 as the sign bit.

In 2's complement representation, subtraction of two integers can be accomplished by usual binary summation by computing $x - y$ as $x + (-y)$. We investigate the operations on the 2's complement numbers later (Section 2.1.1.2: Two's complement arithmetic). However, when you add two 2's complement numbers, you must keep in mind that the 1 in MSB is actually -1.

Exercise 2.1*(Solution on p. 25.)*

(2's complement): What are the decimal numbers corresponding to the 2's complement 8-bit binary numbers; 01001101_2 , 11100100_2 , 01111001_2 , and 10001011_2 ?

Sometimes, you need to convert an 8-bit 2's complement number to a 16-bit number. What is the 16-bit 2's complement number representing the same value as the 8-bit numbers 01001011_2 and 10010111_2 ? The answer is 0000000001001000_2 and 1111111110010111_2 . For nonnegative numbers (sign bit = 0), you simply add enough 0's to extend the number of bits. For negative numbers, you add enough 1's. This operation is called **sign extension**. The same rule holds for extending a 16-bit 2's complement number to a 32-bit number.

For the arithmetic assembly instructions, C62x CPU has different versions depending on how it handles the signs. For example, the load instructions LDH and LDB load halfword and byte value to a 32-bit register with sign extension. That is, the loaded values are converted to 32-bit 2's complement number and loaded into a register. The instructions LDHU and LDBU do not perform sign extension. They simply fill zeros for the upper 16- and 24-bits, respectively.

For the shift right instructions SHR and SHRU, the same rule applies. The ADDU instruction simply treats the operands as unsigned values.

2.1.1.1.3 Fractional representation

Although using 2's complement integers we can implement both addition and subtraction by usual binary addition (with special care for the sign bit), the integers are not convenient to handle to implement DSP algorithms. For example, if we multiply two 8-bit words together, we need 16 bits to store the result. The number of required word length increases without bound as we multiply numbers together more. Although not impossible, it is complicated to handle this increase in word-length using integer arithmetic. The problem can be easily handled by using numbers between -1 and 1 , instead of integers, because the product of two numbers in $[-1, 1]$ are always in the same range.

In the 2's complement fractional representation, an N bit binary word can represent 2^N equally spaced numbers from $\frac{(-2)^{N-1}}{2^{N-1}} = -1$ to $\frac{2^{-(N-1)}}{2^{N-1}} = 1 - 2^{N-1}$.

For example, we interpret an 8-bit binary word

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

as a fractional number

$$x = \frac{-(b_72^7) + b_62^6 + \dots + b_12 + b_0}{2^7} = -(b_7) + \sum_{i=0}^6 (2^{i-7}b_i) \in [-1, 1 - 2^{-7}]$$

This representation is also referred as **Q-format**. We can think of having an implied binary digit right after the MSB. If we have an N -bit binary word with MSB as the sign bit, we have $N - 1$ bits to represent the fraction. We say the number has Q-($N - 1$) format. For example, in the example, x is a Q-7 number. In C6211, it is easiest to handle Q-15 numbers represented by each 16 bit binary word, because the multiplication of two Q-15 numbers results in a Q-30 number that can still be stored in a 32-bit wide register of C6211. The programmer needs to keep track of the implied binary point when manipulating Q-format numbers.

Exercise 2.2*(Solution on p. 25.)*

(Q format): What are the decimal fractional numbers corresponding to the Q-7 format binary numbers; 01001101_2 , 11100100_2 , 01111001_2 , and 10001011_2 ?

2.1.1.2 Two's complement arithmetic

The convenience of 2's complement format comes from the ability to represent negative numbers and compute subtraction using the same algorithm as a binary addition. The C62x processor has instructions to add,

subtract and multiply numbers in the 2's complement format. Because, in most digital signal processing algorithms, Q-15 format is most easy to implement on C62x processors, we only focus on the arithmetic operations on Q-15 numbers in the following.

2.1.1.2.1 Addition and subtraction

The addition of two binary numbers is computed in the same way as we compute the sum of two decimal numbers. Using the relation $0 + 0 = 0$, $0 + 1 = 1 + 0 = 1$ and $1 + 1 = 10$, we can easily compute the sum of two binary numbers. The C62x instruction ADD performs this binary addition on different operands.

However, care must be taken when adding binary numbers. Because each Q-15 number can represent numbers in the range $[-1, 1 - 2^{-15}]$, if the result of summing two Q-15 numbers is not in this range, we cannot represent the result in the Q-15 format. When this happens, we say an **overflow** has occurred. Unless carefully handled, the overflow makes the result incorrect. Therefore, it is really important to prevent overflows from occurring when implementing DSP algorithms. One way of avoiding overflow is to scale all the numbers down by a constant factor, effectively making all the numbers very small, so that any summation would give results in the $[-1, 1)$ range. This **scaling** is necessary and it is important to figure out how much scaling is necessary to avoid overflow. Because scaling results in loss of effective number of digits, increasing quantization errors, we usually need to find the minimum amount of scaling to prevent overflow.

Another way of handling the overflow (and underflow) is **saturation**. If the result is out of the range that can be properly represented in the given data size, the value is saturated, meaning that the value closest to the true result is taken in the range representable. Such instructions as SADD, SSUB perform the operations followed by saturation.

Exercise 2.3

(Solution on p. 25.)

(Q format addition, subtraction): Perform the additions $01001101_2 + 11100100_2$, and $01111001_2 + 10001011_2$ when the binary numbers are Q-7 format. Also compute $01001101_2 - 11100100_2$ and $10001011_2 - 00110111_2$. In which cases, do you have overflow?

2.1.1.2.2 Multiplication

Multiplication of two 2's complement numbers is a bit complicated because of the sign bit. Similar to the multiplication of two decimal fractional numbers, the result of multiplying two Q-N numbers is Q-2N, meaning that we have 2N binary digits following the implied binary digit. However, depending on the numbers multiplied, the result can have either 1 or 2 binary digits before the binary point. We call the digit right before the binary point the **sign bit** and the one preceding the sign bit (if any) the **extended sign bit**.

The following is the two examples of binary fractional multiplications:

```

0.110 0.75 Q-3
X 1.110 -0.25 Q-3
-----
 0000
0110
   0110
  1010
-----
1110100 -0.1875 Q-6

```

Above, all partial products are computed and represented in Q-6 format for summation. For example, $0.110 * 0.010 = 0.01100$ in Q-6 for the second partial product. For the 4th partial product, care must be

taken because in 0.110×1.000 , 1.000 represents -1 , so the product is $-0.110 = 1.01000$ (in Q-6 format) that is 2's complement of 0.11000 . As noticed in this example, it is important to represent each partial product in Q-6 (or in general Q- $2N$) format before adding them together. Another example makes this point clearer:

```

1.110 -0.25 Q-3
X 0.110 0.75 Q-3
-----
0000
  111110
  11110
  0000
  -----
11110100 -0.1875 Q-6

```

For the second partial product, we need 1.110×0.010 in Q-6 format. This is obtained as 1111100 in Q-6 (check!). A simple way to obtain it is to first perform the multiplication in normal fashion as $1110 \times 0010 = 11100$ ignoring the binary points, then perform **sign extension** by putting enough 1s (if the result is negative) or 0s (if the result is nonnegative), then put the binary point to obtain a Q-6 number. Also notice that we need to remove the extra sign bit to obtain the final result.

In C62x, if we multiply two Q-15 numbers using one of multiply instruction (for example MPY), we obtain 32 bit result in Q-30 format with 2 sign bits. To obtain the result back in Q-15 format, (i) first we remove 15 trailing bits and (ii) remove the extended sign bit.

Exercise 2.4

(Solution on p. 25.)

(Q format multiplication): Perform the multiplications 01001101×11100100 , and 01111001×10001011 when the binary numbers are Q-7 format.

2.1.1.3 Assembly language implementation

When A0 and A1 contain two 16-bit numbers in the Q-15 format, we can perform the multiplications using MPY followed by a right shift.

```

1    MPY    .M1    A0,A1,A2
2    NOP
3    SHR    .S1    A2,15,A2    ;lower 16 bit contains result
4                                ;in Q-15 format

```

Rather than throwing away the 15 LSBs of the multiplication result by shifting, you can round up the result by adding $0x4000$ before shifting.

```

1    MPY    .M1    A0,A1,A2
2    NOP
3    ADDK   .S1    4000h,A6
4    SHR    .S1    A2,15,A2    ;lower 16 bit contains result
5                                ;in Q-15 format

```

2.1.1.4 C language implementation

Let's suppose we have two 16-bit numbers in Q-15 format, stored in variable **x** and **y** as follows:

```
short x = 0x0011;    /* 0.000518799 in decimal */
short y = 0xfe12;    /* -0.015075684 in decimal */
short z;             /* variable to store x*y */
```

The product of **x** and **y** can be computed and stored in Q-15 format as follows:

```
z = (x * y) >> 15;
```

The result of **x*y** is a 32-bit word with 2 sign bits. Right shifting it by 15 bits ignores the last 15 bits, and storing the shifted result in **z** that is a **short** variable (16 bit) removes the extended sign bit by taking only lower 16 bits.

Solutions to Exercises in Chapter 2

Solution to Exercise 2.1 (p. 20)

Intentionally left blank.

Solution to Exercise 2.2 (p. 21)

Intentionally left blank.

Solution to Exercise 2.3 (p. 22)

Intentionally left blank.

Solution to Exercise 2.4 (p. 23)

Intentionally left blank.

Chapter 3

Chapter 3

3.1 Unit Sample Signal¹

3.1.1 Unit Sample Signal

The unit sample signal (unit impulse) in discrete-time systems is one of the most important signals, which is defined by

$$\text{Unit Sample Signal} \tag{3.1}$$

Unit Sample Signal

This is an unsupported media type. To view, please see <http://cnx.org/content/m10897/latest/>

Figure 3.1

A discrete-time signal has a decomposition as a summation of weighted and shifted unit sample. (3.2)

3.2 Discrete-Time Filtering²

3.2.1 Discrete-Time Filtering

Most of you should be already familiar with continuous-time filters, which takes continuous-time input signal and output a continuous-time signal. Discrete-time filters, which the sampling device and digital-to-analog

¹This content is available online at <http://cnx.org/content/m10897/2.1/>.

²This content is available online at <http://cnx.org/content/m10908/2.3/>.

converter, can perform the same function as the continuous-time filters with properly designed system blocks. Continuous-time filters are fully specified by its impulse response $h(t)$ and the output signal $y(t)$ for input signal $x(t)$ is given by the convolution integral

$$(3.3)$$

Similarly, discrete-time filters are specified by their unit sample response $h(n)$. The output signal $y(n)$ for the input signal $x(n)$ (samples of input continuous-time signal $x(t)$) is given by the discrete-time convolution

$$(3.4)$$

3.2.2 Discrete-Time Filter Design

The discrete-time filter design problem is to design the impulse response $h(n)$ so that the discrete-time filter, together with the sampling device and the discrete-to-continuous time converter, performs the same signal processing functions as an analog filter. There are many algorithms to design $h(n)$ to implement desired filtering.

3.3 Filter design by windowing³

3.3.1 FIR Filter Design by Windowing

3.3.1.1 Desired Ideal Filter Response

Unlike the design of IIR filters, the design of FIR filters starts from the spectrum of the desired filter in the DTFT domain. Let $H_d\left(e^{j\frac{\omega}{2}}\right)$ be the ideal desired frequency response. Then, the impulse response $h_d[n]$ corresponding to $H_d\left(e^{j\frac{\omega}{2}}\right)$ is obtained by the inverse DTFT as

$$(3.5)$$

For general desired frequency response, $h_d[n]$ is usually noncausal and infinitely long.

3.3.1.2 Windowing of Impulse Response

To obtain an FIR filter approximating the frequency response of the desired ideal filter, we need to find a filter impulse response a causal and finite $h[n]$ that approximates $h_d[n]$. The simplest way to obtain such $h[n]$ is to define a new system with impulse response $h[n]$ given by

$$(3.6)$$

where $w[n]$ is a finite duration window. For example, simple truncation corresponds to the **boxcar window** given as

$$(3.7)$$

Other commonly used windows include Bartlett, Hamming, Hanning, and Blackman windows.

3.3.1.3 Frequency Response of Windowed Impulse Response

Because the multiplication by the window function in time domain corresponds to a convolution in the DTFT domain, we can easily visualize the spectrum of the designed FIR filter. Different window functions have different main lobe width and peak sidelobe heights. The width of the main lobe governs the property of the frequency transition at band edges. The height of the peak sidelobes is related to the oscillations near the transition frequencies.

³This content is available online at <<http://cnx.org/content/m10912/2.2/>>.

3.4 Parks-McClellan Optimal FIR Filter Design⁴

3.5 FIR Filter Design using MATLAB⁵

3.5.1 FIR Filter Design Using MATLAB

3.5.1.1 Design by windowing

The MATLAB function `fir1()` designs conventional lowpass, highpass, bandpass, and bandstop linear-phase FIR filters based on the windowing method. The command

```
b = fir1(N,Wn)
```

returns in vector `b` the impulse response of a lowpass filter of order `N`. The cut-off frequency `Wn` must be between 0 and 1 with 1 corresponding to the half sampling rate.

The command

```
b = fir1(N,Wn,'high')
```

returns the impulse response of a highpass filter of order `N` with normalized cutoff frequency `Wn`.

Similarly, `b = fir1(N,Wn,'stop')` with `Wn` a two-element vector designating the stopband designs a bandstop filter.

Without explicit specification, the **Hamming window** is employed in the design. Other windowing functions can be used by specifying the windowing function as an extra argument of the function. For example, Blackman window can be used instead by the command `b = fir1(N, Wn, blackman(N))`.

3.5.1.2 Parks-McClellan FIR filter design

The MATLAB command

```
b = remez(N,F,A)
```

returns the impulse response of the length `N+1` linear phase FIR filter of order `N` designed by Parks-McClellan algorithm. `F` is a vector of frequency band edges in ascending order between 0 and 1 with 1 corresponding to the half sampling rate. `A` is a real vector of the same size as `F` which specifies the desired amplitude of the frequency response of the points $(F(k), A(k))$ and $(F(k+1), A(k+1))$ for odd `k`. For odd `k`, the bands between `F(k+1)` and `F(k+2)` is considered as transition bands.

⁴This content is available online at <http://cnx.org/content/m10914/2.2/>.

⁵This content is available online at <http://cnx.org/content/m10917/2.2/>.

3.6 MATLAB FIR Filter Design Exercise⁶

3.6.1 FIR Filter Design MATLAB Exercise

3.6.1.1 Design by windowing

Exercise 3.1

(Solution on p. 35.)

Assuming sampling rate at 48kHz, design an order-40 low-pass filter having cut-off frequency 10kHz by windowing method. In your design, use Hamming window as the windowing function.

3.6.1.2 Parks-McClellan Optimal Design

Exercise 3.2

(Solution on p. 35.)

Assuming sampling rate at 48kHz, design an order-40 lowpass filter having transition band 10kHz-11kHz using the Parks-McClellan optimal FIR filter design algorithm.

3.7 Assembly Implementation of FIR Filters on TI TMS320C62x⁷

3.7.1 Implementation of FIR filters in assembly

3.7.1.1 Storing filter coefficients

Rather than defining the filter coefficients in your main assembly program file, it is usually more convenient to store them in a separate file. By defining the coefficients in a separate assembly (for example, `coeff.asm`) file, you can load the coefficients at a desired memory location at the run time, although it is not essential for the current simple FIR filtering lab.

The assembly file containing the filter coefficients can be written as follows:

```
.def    _coef
.sect   "coeffs"

_coef:
.short  0ff9bh
.short  0ff06h
.short  0feffh
.short  0ff93h
.short  070h
.short  0117h
.short  0120h
.short  07bh
```

Each coefficient must be converted to the Q-15 format and defined by each `.short` assembly directive. For your convenience, I wrote a short MATLAB script `save_coef.m` that converts the filter coefficients stored as a MATLAB vector to Q-15 format and then writes to a file exactly in the above format. (You can download

⁶This content is available online at <http://cnx.org/content/m10918/2.2/>.

⁷This content is available online at <http://cnx.org/content/m10920/2.4/>.

`save_coef.m` from the course web page.) The section `coeffs` should be defined in the link command file so that the coefficients are to be loaded at the correct memory location.

You can simply include the `coeff.asm` using the `.include` directive at the beginning of your main assembly program.

Exercise 3.3

(Solution on p. 35.)

Make coefficient files for each of the filters you designed in the previous exercise.

3.7.1.2 Assembly implementation

Based on the codec input and output program you have written in the previous labs, you can now implement a real-time FIR filtering algorithm.

Exercise 3.4

(Solution on p. 35.)

Write an assembly routine that implements the FIR filter by modifying the inner product program you have written in Lab 3. Combine the FIR filtering routine with the interrupt-based codec input-output code you wrote in the previous lab. Your code should perform FIR filtering on the input samples and output the filtered result to the codec. Both the left and right channels should be filtered. To write the designed MATLAB vector of filter coefficients as a `.asm` file, use the provided `save_coef.m` matlab function. First implement the length-40 lowpass filter with 10kHz cutoff designed using the `remez.m`.

3.7.1.3 Implementation using circular addressing modes

As you might already have noticed, a lot of cycles are wasted in FIR filtering while maintaining the buffer to see if you reached the end of buffer and update the address pointers properly. To avoid this unnecessary buffer maintenance, the TI DSP processors have a special addressing mode, called **circular addressing**. Using circular addressing, you can define a block of memory as a circular buffer. As you increase (or decrease) the pointer register pointing to the buffer index beyond the buffer limit, it automatically points to the other end of the buffer, implementing a circle of data array. Instead of moving the data samples themselves, you can move the pointer which specifies the beginning of the buffer, as each new sample is processed. You don't need to check if you reached the end of buffer because the address pointer returns to the beginning of the buffer immediately after reaching the end.

Of the 32 registers on the C6x, 8 of them can perform circular addressing. These registers are A4 through A7 and B4 through B7. Since circular addressing is not default, each of these registers must be specified as circular using the AMR (Address Mode Register) register. The lower 16 bits of the AMR are used to select the mode for each of the 8 registers. The upper 10 bits (6 are reserved) are used to set the length of the circular buffer. Buffer size is determined by 2^{N+1} bytes, where N is the value appearing in the block size fields of the AMR register. The top address of the buffer needs to be aligned with proper physical memory block address using the `.align` assembler directive.

Exercise 3.5

(Solution on p. 35.)

First read TMS320C62x/C67x CPU and Instruction Set Reference Guide to learn how to define circular buffers. Modify your FIR filtering assembly code to use circular addressing modes. After optimizing your code as much as you can, count the number of required clock cycles for each FIR filter output computation. Compare the number with the code written without circular addressing.

3.8 C Language Implementation of FIR Filters on TMS320C62x⁸

3.8.1 C Language Implementation of FIR Filters

Exercise 3.6

(Solution on p. 35.)

Write a C function implementing an FIR filter. Combine this routine with the provided codec input/output loop program to implement a real-time FIR filter. Test your filter program.

3.9 Linear Assembly Implementation of FIR Filters on TMS320C62x⁹

3.9.1 FIR Filter Implementation in TI Linear Assembly

3.9.1.1 What is Linear Assembly?

TI's linear assembly language enables you to write an assembly-like programs without worrying about register usage, pipelining, delay slots, etc. The assembler optimizer program reads the linear assembly code to figure out the algorithm, and then it produces an optimized list of assembly code to perform the operations. The linear assembly programming lets you:

- use symbolic names,
- forget pipeline issues,
- ignore putting NOPs, parallel bars, functional units, register names,
- more efficiently use CPU resources than C.

The linear assembly files have `.sa` extensions. When you have a linear assembly file in your Code Composer Studio project, the assembly optimizer is invoked automatically to generate optimized actual assembly routine. You can consider the linear assembly language as a tool to describe algorithms. To effectively convey the intent of the programmer to the assembly optimizer for proper optimization, there are quite a few extra directives in linear assembly.

3.9.1.2 C callable Linear Assembly procedure

The following is an example of C callable linear assembly routine that computes the dot product of two vectors. It implements a C function

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

If `a[]` and `x[]` are two length-40 vectors, the C function call has the form

```
short a[];
short x[];
short z;
...
...
z = dotp(a,x,40);
```

⁸This content is available online at <http://cnx.org/content/m10923/2.1/>.

⁹This content is available online at <http://cnx.org/content/m10922/2.2/>.

...

(see below how the arguments are passed and the pointers are used.) In the following, you learn various assembler directives used below and how the optimized assembly code is generated by the assembler optimizer.

```

_dotp: .cproc  ap, xp, cnt
      .reg   a, x, prod, y

      MVK   40, cnt
loop:  .trip 40
      LDH  *ap++, a
      LDH  *ax++, x
      MPY  a, x, prod
      ADD  y, prod, y
      SUB  cnt, 1, cnt
[ cnt ] B      loop

      .return      y
      .endproc

```

The `.cproc` directive starts a C callable procedure. It must be used with `.endproc` to end a C procedure. `_dotp:` is the label used to name the procedure. By using `.cproc` to start the procedure, the assembly optimizer performs some operations automatically in a `.cproc` region in order to make the function conform to the C calling conventions and to C register usage convention. The following optional variables (`ap`, `xp`, `cnt` above) represent function parameters. The variable entries are very similar to parameters declared in a C function.

The arguments to the `.cproc` directive can be either machine-register names or symbolic names. When register names are specified, its position in the argument list must correspond to the argument passing conventions for C. For example, the first argument in C function must be register A4. When symbolic names are specified, the assembly optimizer ensures proper allocation and initialization (if necessary) of registers at the beginning of the procedure. To represent a 40-bit argument, a register pair can be specified as an argument. In this lab, however, we only use 32bit values as arguments.

The `.reg` directive allows you to use descriptive names for values that will be stored in registers. It is valid only within procedures only.

The `.return` directive functionality is equivalent to the return statement in C code. It places the optional argument in the appropriate register for a return value as per the C calling conventions. If no argument is specified, no value is returned, similar to a `void` function in C code. To perform a conditional `.return`, you can simply put conditional branch around a `.return` as:

```

[!cc] B  around
      .return
around:

```

The `.trip` directive specifies the value of the trip count. The **trip count** indicates how many times a loop will iterate. By giving this extra information to the assembler optimizer, a better optimization is achieved for loops. The label preceding `.trip` directive represents the beginning of the loop. This is a required parameter.

For more information on writing C callable linear assembly procedure, refer to **TMS320C6x Optimizing C Compiler User's Guide**. For C6x assembly instructions, refer to **TMS320C62x/C67x CPU and Instruction Set Reference Guide**.

Exercise 3.7

(Solution on p. 35.)

Write a C callable FIR filtering routine in linear assembly. When using different optimization levels, what is the number of clock cycles of each FIR filtering?

Solutions to Exercises in Chapter 3

Solution to Exercise 3.1 (p. 30)

```
b = fir1(40,10.0/48.0)
```

Solution to Exercise 3.2 (p. 30)

```
b = remez(40,[1 1 0 0],[0 10/48 11/48 1])
```

Solution to Exercise 3.3 (p. 31)

Solution to Exercise 3.4 (p. 31)

Solution to Exercise 3.5 (p. 31)

Solution to Exercise 3.6 (p. 32)

Solution to Exercise 3.7 (p. 34)

Chapter 4

Supplemental Material

4.1 Rice DSP Lab Setup¹

4.1.1 Laboratory Equipment

Each lab PC is equipped with a Texas Instrument TMS320C6211 DSK board. The DSK board is hooked to the parallel port of the PC running Windows 2000; the PC can control the DSK via the development environment called “Code Composer Studio” (CCS) developed by Texas Instrument. Currently we use CCS version 2.1. If you have your owl net account, create the Samba password to log in to the machines. If you don’t have an owl net account, the TA will give you a username and password for local log in.

For analog signal input and output, the DSK board has a PCM3003 codec daughtercard mounted on the board. This board provides 16 bit stereo analog input and output channels at maximum sampling rate of 48kHz. Currently, the jumpers on the board are configured to provide a fixed 48kHz sampling rate both for input and output. **Do not change the jumper settings unless you know what you are doing.** For variable sampling rates, the timers of the C6211 processor can be used to generate appropriate clock signal. You will learn how to use the CPU timers in Lab 4.

Each lab station is equipped with a function generator to provide signal to be used to test your signal-processing algorithms and an oscilloscope to display the processed waveforms.

The source analog signal also can be generated using the soundcard of the PC and input to the DSK board for processing. There are two shareware softwares installed on each PC that enables you to use the sound card as a tone generator and a simple spectrum analyzer. In the lab experiments, you will be using the tone generator to generate white noise input to the DSK board and you can watch the spectrum of the DSK output using the soundcard spectrum analyzer.

4.1.2 Hardware Setup

The TI DSK board is connected to the parallel port of the PC. The board requires external power supply connected to it. A speaker and microphone can be connected to the board via the two 3.5mm jacks mounted on the daughtercard that are connected to the PCM3003 codec. The hardware reset button (white) is on the DSK board to enable hardware reset of the board. When reset, the board goes through several steps of self testing and initialization. Please refer to the online help of the Code Composer Studio for detailed board operation. After each hardware reset, the DSK board tends to lose the connection with the host PC. Therefore, you should close the CCS before performing hardware reset. Be sure to save all the files before closing the CCS. After the DSK board finishes the reset, start the CCS again.

Be careful when you handle cables, tools, etc, not to short any circuits on the board.

¹This content is available online at <<http://cnx.org/content/m10871/2.3/>>.

4.2 Testing Filters in Rice DSP Lab²

4.2.1 Testing filters in Rice DSP Laboratory

You can test the implemented filters using the tone generator and spectrum analyzer on the PC.

1. Connect the soundcard tone generator to the codec microphone jack to input 5kHz sine wave. Connect the oscilloscope to observe both the codec input and output. As you change the frequency of the sine wave, observe how the amplitude of the filter output changes.
2. Input white noise from PC sound card to the codec input and observe the spectrum of the filter output using the spectrum analyzer program. Can you see the filter frequency response?

²This content is available online at <<http://cnx.org/content/m10921/2.2/>>.

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

- 5** 5-bit constant, 4
- A** addressing modes, 3
architecture, § 1.1(1)
assembler directives, 11
assembly, § 1.2(3), § 3.7(30)
- B** big endian mode, 5
- C** C, § 3.8(32)
circular, 6
circular addressing, 31
conditional operations, 8
cross path, 13
- D** delay slots, 6
DSP, § 4.1(37)
- E** endian mode, 5
exercise, § 3.6(30)
extended sign bit, 22
- F** filter, § 4.2(38)
FIR, § 3.5(29), § 3.6(30), § 3.7(30), § 3.8(32)
fixed point arithmetic, § 2.1(19)
functional unit, § 1.1(1)
functional unit latency, 7
- L** laboratory, § 4.1(37)
- linear, 6
linear assembly, § 3.9(32)
little endian mode, 5
- M** MATLAB, § 3.5(29), § 3.6(30)
- N** natural binary, 19
- O** overflow, 20, 22
- P** parallel instructions, 14
processing, § 4.1(37)
- Q** Q-format, 21
- R** register, § 1.1(1), 4
Rice DSP, § 4.2(38)
- S** saturation, 22
sign bit, 20, 22
sign extension, 21
signal, § 4.1(37)
- T** test, § 4.2(38)
TMS320C6211, § 1.1(1)
TMS320C62x, § 3.7(30), § 3.8(32), § 3.9(32)
two's complement, 20, 20
- U** unsigned integer, 19

Attributions

Collection: *Finite Impulse Response*

Edited by: Hyeokho Choi

URL: <http://cnx.org/content/col10226/1.1/>

License: <http://creativecommons.org/licenses/by/1.0>

Module: "TMS320C6211 Architecture Overview"

By: Hyeokho Choi

URL: <http://cnx.org/content/m10872/2.5/>

Pages: 1-3

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Module: "C62x Assembly Primer II"

By: Hyeokho Choi

URL: <http://cnx.org/content/m11051/2.3/>

Pages: 3-17

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Module: "Fixed Point Arithmetic"

By: Hyeokho Choi

URL: <http://cnx.org/content/m11054/2.2/>

Pages: 19-24

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Module: "Unit Sample Signal"

By: Hyeokho Choi

URL: <http://cnx.org/content/m10897/2.1/>

Page: 27

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Module: "Discrete-Time Filtering"

By: Hyeokho Choi

URL: <http://cnx.org/content/m10908/2.3/>

Pages: 27-28

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Module: "Filter design by windowing"

By: Hyeokho Choi

URL: <http://cnx.org/content/m10912/2.2/>

Page: 28

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Module: "Parks-McClellan Optimal FIR Filter Design"

By: Hyeokho Choi

URL: <http://cnx.org/content/m10914/2.2/>

Page: 29

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Module: "FIR Filter Design using MATLAB"

By: Hyeokho Choi

URL: <http://cnx.org/content/m10917/2.2/>

Page: 29

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Module: "MATLAB FIR Filter Design Exercise"

By: Hyeokho Choi

URL: <http://cnx.org/content/m10918/2.2/>

Page: 30

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Module: "Assembly Implementation of FIR Filters on TI TMS320C62x"

By: Hyeokho Choi

URL: <http://cnx.org/content/m10920/2.4/>

Pages: 30-31

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Module: "C Language Implementation of FIR Filters on TMS320C62x"

By: Hyeokho Choi

URL: <http://cnx.org/content/m10923/2.1/>

Page: 32

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Module: "Linear Assembly Implementation of FIR Filters on TMS320C62x"

By: Hyeokho Choi

URL: <http://cnx.org/content/m10922/2.2/>

Pages: 32-34

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Module: "Rice DSP Lab Setup"

By: Hyeokho Choi

URL: <http://cnx.org/content/m10871/2.3/>

Page: 37

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Module: "Testing Filters in Rice DSP Lab"

By: Hyeokho Choi

URL: <http://cnx.org/content/m10921/2.2/>

Page: 38

Copyright: Hyeokho Choi

License: <http://creativecommons.org/licenses/by/1.0>

Finite Impulse Response

A course on FIR filter design.

About Connexions

Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.