

ECE 420 Introduction to TI DSP Assembly

Prepared by Thomas Shen

Heavily based off of 'ECE320 Introduction to TI DSP Assembly' by Michael Kramer and Daniel Sachs

1 Introduction

The purpose of this document is to provide an overview of the basics of assembly language and the software tools used in the labs. This includes a brief description of how to write and assemble the code, the memory layout of the DSPs, and an introduction the fundamental TI assembly language instructions.

The evaluation board at each station includes a Texas Instruments 320C5510 DSP. For more information about the DSP chip itself, look at SPRU371 55x DSP CPU Reference Guide. Each board has been filled out to include analog-to-digital and digital-to-analog converters, memory, as well as stereo 3.5mm input and output jacks. A daughtercard has been installed to provide four input and four output channels. The BNC connectors allow the user to easily connect input from a function generator and view the output on an oscilloscope.

2 Writing and Assembling Code

Assembly source code can be written using any text editor and typically has a `.asm` extension. The following describes the format of the TI assembly code and what is required to assemble it for your lab exercises.

Assembly Fields: In the example below, notice that the code is separated into three fields. The first field is reserved for labels. Labels are simply symbols for numbers used during the assembly process and will not explicitly exist in the final code. For example, line 1 defines the label `N_taps` using the `.set` directive, and can then be used in the code to refer to the value 16. Once the code is assembled, the value 16 will be used in place of each occurrence of `N_taps`.

```
1   N_taps   .set    16
2
3           .sect  ".text"
4
5   main     add    AC0, AC1
```

Labels can also refer to data or program memory locations, as in line 5. In this case `main` is a label to the location in program memory where the add instruction is stored.

The second field in the code is for instructions or directives. Instructions are the commands for the DSP that are converted to opcodes and loaded onto the processor while directives are like labels

in that they are only used during the assembly process. The third field is for operands for the instructions or directives. In line 1 for example the second field has the `.set` directive and the operand, 16, is located in the third field.

Each field must be separated by at least one space, although tabbing each field is often recommended for readability.

Assembler Directives: As mentioned, assembler directives are commands for the assembler to use but do not get compiled for execution on the DSP. Below is a list of commonly used directives and a brief description of their use. More information about each can be found in SPRU280 55x Assembly Language Tools User's Guide.

Directive	Function
<code>.copy</code>	copy contents of file at current memory location
<code>.set</code>	initialize a label
<code>.sect</code>	begin assembling into a different section of memory
<code>.align</code>	push memory counter to bit-boundary
<code>.space</code>	reserve one bit in memory and initialize to zero
<code>.word</code>	initialize one word of memory

Assembling Code: The project file contains settings that define the files needed to compile, assemble, and link your code. These three steps can be completed by selecting **Debug->Rebuild All**.

3 TI Assembly Language Primer

To program effectively in assembly language it is important to have a good understanding of the basic processor instructions and memory usage. As the majority of signal processing applications involve simple arithmetic operations, we first review some of the addition and multiplication instructions available. Memory access instructions are then discussed to familiarize you with the basics of reading and writing to memory on the DSP, followed by a summary of conditional and logical (bit-wise) instructions.

For more information on these topics, or any particular instruction, please refer to the CPU overview and/or Mnemonic Instruction Set manuals.

Register Usage In contrast to higher level languages such as C or Basic, programming at the assembly language level does not allow for arbitrary variable definition. For example, in a high level language we can define two variables, say $X=2$ and $Y = 4$, then add them together, $Z= X + Y$. To perform the same task however at the assembly language level the data is first loaded from memory, then added with the result stored in an accumulator register. Finally, the contents of the accumulator is store back into memory.

This requires that the assembly language programmer become considerably more familiar with the available registers on the CPU than would be required for higher level programming. The primary data/ mathematical related registers on the TI processor include accumulators for arithmetic operations, address registers to act as, and modify "pointers" for memory access, and program control registers for loops, function calls and branching. Each of these groups of registers will be discussed in further detail in the following sections.

On the TI C55X chip, each of these primary registers are memory mapped in internal memory from locations 0x00h to 0x5Fh. See Chapter 2 of SPRU371 55x DSP CPU Reference Guide for a list of all the CPU registers.

3.1 Arithmetic Operations

Any arithmetic operation (multiplies, adds, etc.) are performed by the Arithmetic Logic Unit (ALU) and the result of any of these operations is stored into one of three accumulators, AC0, AC1 or AC2. Given that each data memory location is 16 bits wide, a multiplication of the contents of two data memory locations would require 32 bits of precision. Each accumulator is 40 bits wide, 32 to account for the range required for multiplication, and 8 extra guard bits to accommodate some overflow during addition.

The ALU on the TI 55x chip operates in 2's complement notation, and the following examples presuppose a full understanding of this notation. The reader less familiar with conversion between signed fractional, signed integer, and 2's complement notation should refer to the section at the end of this handout for a review.

Fractional Multiplier The TI 55x DSP microprocessor is a 16-bit integer processor, with some extra support for fractional arithmetic. Fractional arithmetic turns out to be very useful for DSP programming, since it frees us from worries about overflow on multiplies. (Two 16-bit numbers, multiplied together, can require 32 bits for the result. Two 16-bit fixed point fractional numbers also require 32 bits for the results, but the 32-bit result can be rounded into 16 bit while only introducing an error of approximately 2^{-16} .) For this reason, we will be using fixed-point fractional representation to describe filter taps and inputs throughout this course.

Unfortunately, the assembler and debugger we are using do not recognize this fractional fixed-point representation. For this reason, when you're using the assembler or debugger, you'll see decimal values (ranging from -32768 to 32767) on screen instead of the fraction being represented. The fractional number being represented is simply the decimal value shown divided by 32,768. This allows us to represent numbers between -1.0 and $(1 - 2^{-15})$. (Not that 1.0 cannot be represented exactly.)

It turns out that when we multiply using this representation, an extra shift left is required. Consider the two examples below where the multiplication is done in standard integer form:

When we do the multiplication, we are primarily interested in the top 16 bits of the result, since it is the data that is actually used when we store the result back into memory and send it

fractional	0.5	x	0.5	=	0.25				
decimal	16384	x	16384	=	268435456	=	4096	*	2^{16} (= .125)
hex	4000h	x	4000h	=	1000 0000h	=	1000h	*	2^{16}
fractional	0.125	x	0.75	=	0.093750				
decimal	4096	x	24576	=	101400576	=	1536	*	2^{16} (= .046875)
hex	1000h	x	6000h	=	0600 0000h	=	0600h	*	2^{16}

out to the digital-to-analog converter. (The entire result is actually stored in the accumulator, so rounding errors do not accumulate when we do a sequence of multiply-accumulate operations.) As the example above shows, the top 16 bits of the result of multiplying the fixed point fractional numbers together is half the expected fractional result. The extra left shift multiplies the result by two, giving us the correct final product.

The left-shift requirement can alternatively be explained by way of decimal place alignment. Remember that when we multiply decimal numbers, we can just multiply them ignoring the decimal points, then put the decimal point back. The decimal point is placed so that the total number of digits right of the decimal point in the numbers we're multiplying is the same as the number of digits right of the decimal point in the result. The same applies here; the "decimal point" is to the right of the leftmost (sign) bit, and there are 15 bits (digits) to the right of this point. So there's a total of 30 bits to the right of the decimal in the source. But if we don't shift the result, there's 31 bits to the right of the decimal in the 32-bit result. So we shift the number to the left by one bit, which effectively reduces the number of bits right of the decimal to 30.

Before the numbers are multiplied by the ALU, each term is sign-extended generating a 17-bit number from the 16-bit input. Because the examples presented above are all positive, the effect of this sign extension is simply adding an extra "0" bit at the top of the register (i.e. positive numbers are not affected by the sign extension). As the following example illustrates, not including this sign-bit for negative numbers results in erroneous results.

fractional	-0.5	x	0.5	=	-0.25				
decimal	49152	x	16384	=	805306368	=	12288	*	2^{16} (= 0.375)
hex	C000h	x	4000h	=	3000 0000h	=	3000h	*	2^{16}

Note that even after the result is left shift by one bit following the multiply, the top bit of the result is still "0", implying that the result is incorrectly interpreted as a positive number.

To correct this problem, if either or both of the numbers being multiplied are negative, the ALU sign-extends the negative multiplier inputs by placing a "1" instead of a "0" in the added bit. This is called sign extension because the sign bit is "extended" to the left another place, adding an extra bit to the left of the number without changing the value.

Although the top bit of this result is still "0", after the final 1-bit left-shift the result is E000

fractional	-0.5	x	0.5	=	-0.25	
hex	1C000h	x	4000h	=	7000 0000h	= 7000h * 2 ¹⁶

0000h which is an appropriate negative number (the top bit is "1"). To check the final answer, we can negate the result using the 2's complement method described above. After flipping all the bits we have 1FFF FFFFh, and adding one yields 2000 0000h, which equals 0.25 when interpreted as a 32-bit fractional number.

Storing results: After the desired result is obtained in the accumulator, it often must be stored back into memory. This is accomplished with one of the store accumulator content to memory instructions, MOV. In the case of the fractional multiply example, the contents of the accumulator is interpreted as a fractional value with the significant bits residing in the high part of the accumulator, and so we would use the instruction

```
mov HI(ACO), #2000
```

Because the MOV instruction stores only the 16 bits of the high part of the accumulator there are two conditions to be aware of, saturation and rounding. Saturation becomes an issue if the contents of the accumulator are greater than 1.0 or less than -1.0 when interpreted as a fractional value. If only the desired 16-bits of the high part of the accumulator are moved then wraparound occurs (What happens if the sign-bit is lost in 2's complement notation?) Accumulator saturation prior to storage is also an option, and would simply store the 2's complement notation of 0.999999 if the actual contents of the accumulator were greater than or equal to 1.0.

The rounding problem arises when the store-high instruction chops off the lower least 16 bits (in the low part of the accumulator). If the result is small then this truncation could result in large errors, always forcing the saved result to be smaller than the contents of the accumulator. To reduce the amount of error this truncation introduces, use the RND instruction to round the accumulator prior to storage.

3.1 Memory Access

There are several memory access modes available on the TI DSP. The following provides a brief summary and examples of the three modes most commonly used. For more information on these addressing modes, refer to SPRU371 55x DSP CPU Reference Guide.

Immediate Addressing: uses the instruction to encode a fixed value

```
MOV #0h, ACO
```

Absolute Addressing uses the instruction to encode a fixed address. The following example stores the value 20 into memory location 1000.

```
MOV #20, #1000
```

Generally speaking this is not a good way to reference memory. If the memory map changes for exampl and location 1000 is no longer available then the code will not work.

Typically, absolute addresssing is performed using a label that has already been assigned to a location in memory. If we assume that `states` is a label to memory then this instruction will store 20 into the location labeled `states`.

```
MOV #20, *(#states)
```

Indirect Addressing uses address registers as pointers to access memory. These address registers are maintained by the address generation unit on the DSP. These registers include the auxiliary registers AR0 .. AR7 acting as "pointers" to data memory, and control registers such as the circular buffer registers (BK03 BK47 BKC).

Consider the following example:

```
mov #states, AR1
add *AR1, ACO
```

In the first line the auxiliary register AR1 is being initialized to the lcation in memory labeled `states`. The second line is an `add` instruction and uses the AR1 auxiliary register to "point" to the desired location in memory. When the instruction is executed, the processor will add the contents of where AR1 is pointing (i.e. the contents of the memory location labeled `states`) to the accumulator. (This could have been performed with a absolute memory reference using "`add*(#states), ACO`".) In addition to allowing for faster memory access, address registers can be modified with the instruction. For example,

```
add *AR1+, ACO
```

will first perform the addition as discussed and then automatically increment AR1 to the next memory location.

One special form of address modification is for circular addressing. Suppose after incrementing an auxiliary register we would like to wrap back to the beginning of some buffer in memory. Such buffers are referred to as circular buffers. Before circular addressing can be applied, we must first specify the length of the buffer by assigning it to the corresponding circular buffer size register (BK03 for AR0 - AR3, BK47 for AR4 - AR7, or BKC for CDP). Next, you need to set up the buffer start address by setting the corresponding BSA register with the starting memory location of the circular buffer (BSA01 for AR0/AR1, BSA23 for AR2/AR3, BSA45 for AR4/AR5, BSA67 for AR6/AR7, or BSAC for CDP). With circular addressing turned on, the address generated will be the sum of the corresponding BSA register and the corresponding AR register. Circular addressing is performed when it is "set" for a certain register, as in the example below.

```
mov #buf_size, BK03
mov #states, BSA01
mov #index, AR1
bset AR1LC
add *AR1+, ACO
```

The memory location at label `index` is to store the current location of the desired point in the circular buffer. Since registers are reused by other functions and routines (hardware and software interrupts), the ending value of AR1 needs to be stored so that the correct starting location is used the next time your function runs. Once the lower bits of the AR1 register equal the contents of the BK03 register, then these bits are cleared out forcing the values of AR1 to wrap around.

Note: Due to the use of BSA registers, the start of the circular buffer is not bound by any bit boundary requirements. This is different from the 54x which did not have BSA registers.

4 Two's complement arithmetic

2's complement arithmetic notation is an efficient way of representing signed numbers in microprocessors. It offers the advantage that addition and subtraction can be done with ordinary unsigned operations. When a number is written in 2's complement notation, the most significant bit of the number represents its sign: 0 means that the number is positive, and 1 means the number is negative. A positive number written in 2's complement notation is the same as the number written in unsigned notation (although the most significant bit must be clear). A negative number can be written in 2's complement notation by inverting all of the bits of its absolute value, then adding on to the result.

For example, consider the following 4-bit 2's complement numbers (in binary form): Note that

1 = 0001	-1 = 1110+1 = 1111
2 = 0010	-2 = 1101+1 = 1110
6 = 0110	-6 = 1001+1 = 1010
8 = 1000	-8 = 0111+1 = 1000

1000 represents -8, not 8. This is because the topmost bit (the sign bit) is 1, indicating that the number is negative.

The maximum number that can be represented with a k bit 2's complement notation is $2^k - 1$, and the minimum number that can be represented is -2^k . Since the TI 55x DSP is a 16-bit processor, the maximum integer that can be represented in a 16-bit member register is 32767, and the minimum integer is -32768.