REAL-TIME DSP LABORATORY 3:
Assembly and Linear Assembly on the C6713 DSK

Contents

1 Introduction 1
2 Assembly Language on the C6713 1
3 Calculating a Finite Sum using C and Assembly Code 3
4 Calculating a Finite Sum using Linear Assembly* 9
4.1 Assignment 1 11
5 Multiply and Accumulate in Assembly* 11
5.1 Assignment 2 14
5.2 Assignment 3 14
5.3 Assignment 4 14
6 Optimizing Assembly Code 15
7 End Notes 16

Note: Starred sections contain assigned tasks to be written up in the report.

1 Introduction

DSP programming in assembly code is essential for the efficient coding of algorithms in DSP hardware. Moreover, it brings insight into how these algorithms actually work at the register level. Much of this insight is concealed in C programs.

In general, we will use a C program to communicate with the DSK board, but with a C program, we can call C-coded functions, assembly-coded functions, or linear assembly-coded functions to implement DSP algorithms.

The examples in this lab will not have any real-time constraints, but future labs in digital filtering will, so we will explore the idea of efficient coding of functions using assembly and linear assembly coding. In this lab, you will study

- assembly and linear assembly coding of functions that can be called from a C program,
- multiplication and accumulation in assembly and linear assembly, and
- optimization of assembly language code.

2 Assembly Language on the C6713

Using assembly code, unlike C code, you have direct control over the flow of data through the DSP core. Along with optimization advantages, however, this brings the disadvantage that the programming language used is specific to the hardware and the assembler[1]. In our case, the assembler that we use is specific to TI's C6000 series of DSP's and to TI's CCS. A second cost of assembly programming is that development is slower than coding in C. Therefore, it

[1]This is not an issue in C, since the American National Standards Institute (ANSI) created a standard for the C programming language called ANSI C, which is used virtually everywhere (including CCS).

is normally reserved for time-critical operations, particularly those used over and over such as multiply-accumulate (MAC).

The DSP Core

In the C6713 DSP core, there are two *Data Path Register Files* labelled *Data Path A* and *Data Path B* (See Figure 1 below). In each path, there are four *Functional Units*. These include

- `.L` for logical and arithmetic operations,
- `.S` for branch, bit manipulations and arithmetic operations,
- `.M` for multiply operations, and
- `.D` for data transfers (loading/storing) and arithmetic operations.

In hardware, the eight functional units consist of

- the four fixed/floating-point *arithmetic logic units* (ALUs): `.L1`, `.L2`, `.S1`, and `.S2`,
- the two fixed/floating-point multipliers: `.M1` and `.M2`, and
- the two fixed-point ALUs: `.D1` and `.D2`.

These functional units can execute instructions in parallel during one clock cycle. There are a total of 8 possible instructions (4 in each data path) that can be executed in one clock cycle. Since the CPU is operating at 225MHz, it is possible to execute, in principle, 1800 million instructions per second (MIPS), provided 8 instructions are operating in parallel. The functional units `.S`, `.L`, and `.M` can handle floating point operations, which gives a maximum of 1350 million floating-point operations per second (MFLOPS)[6]. These terms are often associated with DSP chips, but they are often misleading since they are maximum processing rates that can only be achieved under special circumstances. The importance of the different functional units and data paths will be seen in assembly language programs we write, where the specific path may be designated for each instruction. We will explore this in assembly language programs that follow.

In each data path (A or B) on the DSP chip, there is a set of sixteen 32-bit registers, namely A0-A15 for path A and B0-B15 for path B. Any of these registers can be used to store values during execution of a program. By convention, when an assembly function is called from C, the values passed to the function will be stored in specific registers. The first 10 arguments passed to an assembly function will be stored (in the following order) to registers `A4`, `B4`, `A6`, `B6`, `A8`, `B8`, `A10`, `B10`, `A12`, `B12`. Any additional arguments will be stored in a stack. The even registers are used when 32-bits of data (or less) are being passed to each register. When a 64-bit (double precision floating-point number) is passed to a function, it is stored in adjoining registers (e.g. `A4:A5`, `B4:B5`, `A6:A7`, etc.) [5]. Upon returning from a called function, only one value may be returned. By convention, the value in register `A4` will be returned. If no value is to be returned, then the C program that called the function will ignore the value in register `A4`. When a project is built and loaded onto the DSK, the program instructions will be organized in blocks of memory for each function. Therefore, when a function is called, the program will branch (jump) to a different part of the program memory. When a branch to a function (or interrupt) is made, the current execution state is saved. This is done by the DSP assembler, which stores the current memory location (in the program memory) to the register `B3` before the branch occurs. Upon completion of a function (or interrupt), the program will branch back to the calling function by making a branch to the memory address stored in `B3`, thus returning the execution state of the calling function. To illustrate this, assume that a C coded program called the assembly coded function `ans = myfunc(a,b,c)`. In this case, the value `a` would be stored in (DSP core) register `A4`, the value `b` would be stored in register `B4`, and the value `c` would be

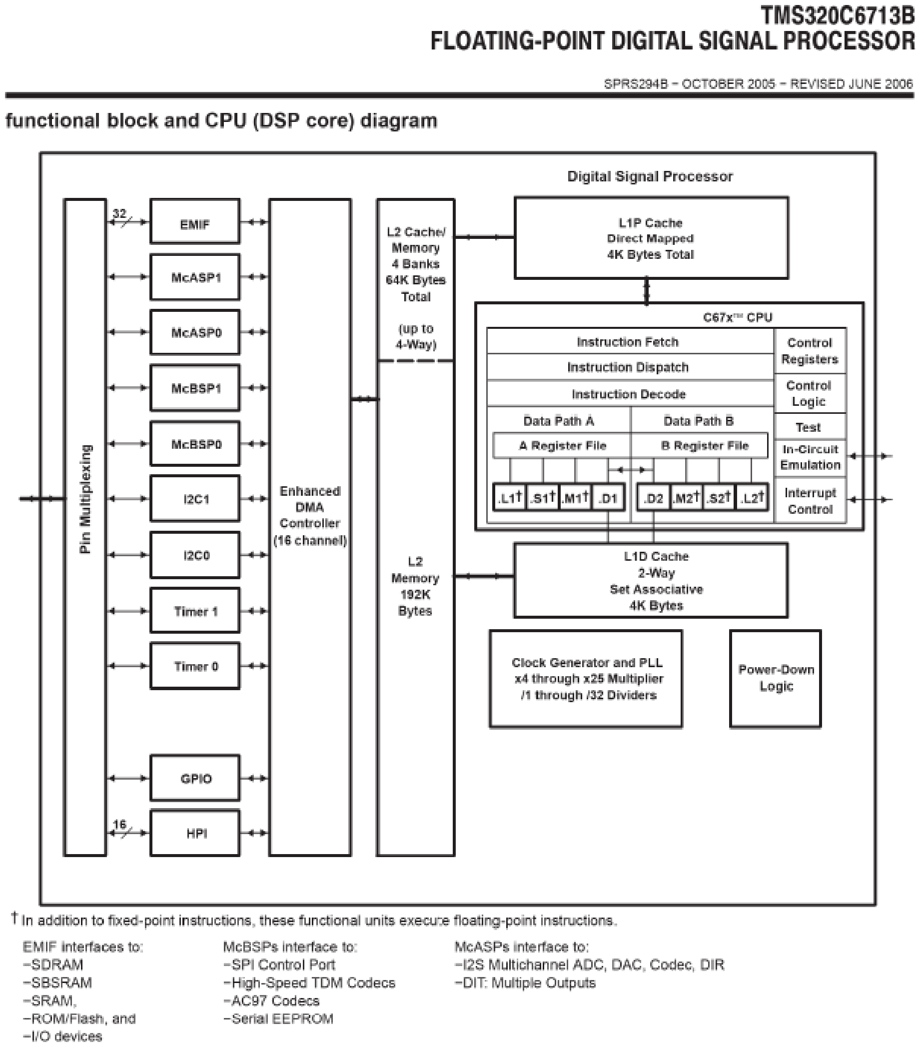

Figure 1: C6713 DSP chip and CPU layout, taken from [6].

stored in register A6. Upon completion of the `myfunc()`, the value stored in register A4 would be stored in the (C program) variable `ans`. The next two programming examples should clarify this discussion.

3 Calculating a Finite Sum using C and Assembly Code

The first assembly language program we shall study computes the finite sum $\sum_{k=1}^{N} k$. To begin, let's examine a C code program that computes a finite sum. Create a project `sum_C` by cloning a clean template project. Delete all the c and asm source files.

Copy or import the C source code file `sum_C.c`. Build this project, load `sum_C.out` onto the DSK, and run this project to verify that the output on the bottom of the screen reads `Sum = 6`.

When the program completes execution, it should halt showing a `Disassembly` window with a bunch of assembly code lines shown (the numbers may differ in your display). The relevant region is:

```
00007400          C$$EXIT, abort:
00007400 00000000          NOP
00007404 00000090          B.S1          0x7404
00007408 00008000          NOP           5
0000740C 00000000          NOP
00007410 00000000          NOP
```

with an arrow showing the *next* line to be executed, pointing at line `0007408`. The program stopped on line `0x7404`; this line is a branch back to itself, i.e. in infinite loop. If you click Run again, the execution point will not move; in order to run the program again, you will need to click `Debug->Restart` and then Run. Notice that before you click Run, the disassembly window will show the program to be in `c_int00`. This system interrupt, not accessible by the user, is the normal starting point for C programs.

Now we look at the actual program. Open up the source code file `sum_C.c` in CCS (shown in Figure 2) and examine it.

The first line of code (after the comments) includes the pre-compiled ANSI C "standard input/output" header file `stdio.h` (By convention, header files have the extension `.h`). This is required for outputting messages to the computer screen. When `sum_C.out` was loaded onto the DSK, CCS automatically created the window `Stdio`, which is where the output appears. The second line of code is a *prototype* of the C function that will be called from the `main()` function[2]. These are required in C because they tell the compiler which functions to expect and what types of data will be passed into and out of these functions. A function prototype must appear in a program before the function is coded. Usually, prototypes appear at the beginning of the program where the global variables and preprocessor directives[3] appear. In the previous labs that used a codec, functions such as `input_sample()` and `output_sample()` (located in the file `dsk6713_bsl.lib`) were used. Each of these functions has a prototype. By convention, when there are many functions in a C program, a header file is used to store the prototypes, which is included in the beginning part of the C program that contains the coded function. For example, open up the file `dsk6713_aic23.h`. You should notice that the only purpose `dsk6713_aic23.h` serves is to hold the prototypes of the codec-related functions coded in the board support library `dsk6713_bsl.lib`. NB: When the word `void` appears as the first word of a prototype, it means that nothing is being returned from these functions.

The `main()` function in this program consists of two lines of code (see lines 25 and 26). The first line passes the value `N` to the function `sum_c_func()` which returns a value that is stored in

[2]A prototype for a C function is simply the first line of the function with a semicolon at the end of it. For example, compare lines 5 and 10 of Figure 2. Prototypes must be declared before the `main()` function and usually appear after the preprocessor directives (`#include` and `#define`).

[3]Preprocessor directives are lines included in the code of programs that are not program statements but directives for the preprocessor. Usually, they are indicated by `#`. The two main ones used in this course are the `#include` directive and the `#define` directive. The `#include` directive tells the preprocessor to replace the line with code from the function given. The `#define` directive tells the preprocessor to replace all instances of variable with the specified value. In both cases, these directives are performed before the code is compiled.

```
/*1 */ // sum_C.c
/*2 */ // calculates a sum by calling a C function
/*3 */
/*4 */ #include <stdio.h>                // required for printing output
/*5 */ short sum_c_func(short k);        // prototype for C function
/*6 */
/*7 */ short N=3;                        // number of integers to sum
/*8 */ short sum;                        // store value returned from sumcfunc
/*9 */
/*10*/ short sum_c_func(short k)
/*11*/ {
/*12*/  short i;
/*13*/  short total = 0;
/*14*/
/*15*/  for (i=k; i>=0; i--)             // sum from top to bottom
/*16*/  {
/*17*/          total += i;
/*18*/  }
/*19*/
/*20*/  return(total);
/*21*/ }
/*22*/
/*23*/ void main()
/*24*/ {
/*25*/   sum = sum_c_func(N);            // call sum_c_func to calculate sum
/*26*/   printf("Sum = %d", sum);        // print result
/*27*/ }
```

Figure 2: Listing of `sum_C.c`

the variable `sum`. The second line prints the result in the window `Stdio` in CCS. To learn more about the syntax of the `printf()` function (and its counterpart `scanf()`) refer to [3].

Functions such as `sum_c_func()` are similar to interrupts in the sense that the program saves the current execution state in register `B3`, branches to the section of code where the function (or interrupt) is located, executes the function (or interrupt), and then returns to the execution state stored in `B3` upon completion. The difference is that a function is executed only after it has been called, whereas an interrupt is processed whenever an interrupt source initiates it. The actual steps involved in executing a function (or interrupt) are not well displayed in C. In assembly, however, this process is drawn out step-by-step. This may initially seem like a drawback, but it leads to reduced code size and invaluable insights about how a process really works. Let's look at an example of how the function `sum_c_func()` could be coded as an assembly function. See Figure 3, which we will annotate shortly.

This function can be called from a C source code, which is how we will use assembly code in this course[4]. The C source that calls this function looks almost identical to the code from `sum_C.c`, except that the function prototype (line 5 of Figure 2)) has been replaced with an *external*

[4] For an example of a complete assembly program on the DSK, see [1, pg 95-96].

```
1.)   ; sum_asm_func.asm
2.)   ; assembly function to find n+(n-1)+...+1+0
3.)
4.)                       .def         _sum_asm_func  ; asm function called from C
5.)   _sum_asm_func:      MV     .L1   A4,A1          ; setup N as loop counter in A1
6.)
7.)   LOOP:               SUB    .S1   A1,1,A1        ; decrement loop counter A1
8.)                       ADD    .L1   A4,A1,A4       ; accumulate in A4
9.)         [A1]          B      .S2   LOOP           ; branch to LOOP if A1#0
10.)                      NOP          5              ; five NOPs for delay slots
11.)                      B      .S2   B3             ; return to calling routine
12.)                      NOP          5              ; five NOPs for delay slots
13.)                      .end
```

Figure 3: Listing of `sum_asm_func.asm`. Adapted from [1].

declaration and the name of the function to be called has been changed. See Figure 4 for a listing of the C code `sum_asm.c` which calls the assembly language function `sum_asm_func()`.

```
// sum_asm.c
// Calculates a sum by calling an assembly function

#include <stdio.h>
extern short sum_asm_func();   // declare external assembly function

short N=3;                     // number of integers to sum
short sum;                     // value returned from sum_asm_func

main()
{
 sum = sum_asm_func(N);        // call sum_asm_func to calculate sum
 printf("Sum = %d", sum);      // print result
}
```

Figure 4: Listing of `sum_asm.c`. Adapted from [1].

Download a Project

Create a workspace for Lab 3 and create a new project `sum_asm` by importing and renaming a clean template project. Delete the old top-level file and replace it with `sum_asm.c` and `sum_asm_func.asm` from the webpage. Delete or exclude from build the `Vectors_intr.asm` and `Vectors_poll.asm` files. In CCS, build this project, load `sum_asm.out` onto the DSK, and run this project to verify that it produces the same result as `sum_C.out`. When `sum_asm.out` was loaded onto the DSK, the global variables were stored at a linker assigned memory location, the compiled and assembled code for the `main()` function were stored at a different linker assigned memory location, and the assembled code for the function `sum_asm_func()` were stored at a separate linker assigned memory location.

Annotating the Assembly Code

Line 4 in `sum_asm_func.asm` defines the starting point of the function `sum_asm_func()`, which is located at line 5. By convention, when an assembly function is called from a C program, the name of the assembly function is preceded by an underscore in its `.asm` file (see line 5 in Figure 3).

When the function call to `sum_asm_func(N)` is made in the C program, the value `N` is moved to register `A4`, and the current execution state of the `main()` function is written to register `B3`. Then, the program branches to the starting point (memory location) of `sum_asm_func()`. The first command, `MV  A4,A1`, copies (moves) the value in register `A4` to register `A1`, so that the value `N` is stored in both registers `A1` and `A4`. This is done for two reasons. First, two registers are required for this algorithm: one register, namely `A4`, is used to accumulate the sum and the second register, namely `A1`, is used to determine the current value to add to the sum and also to terminate the loop. The other reason for selecting register `A1` is that it can be used as a *conditional register*, which will be discussed later. The loop in this assembly coded function executes until `A1` is equal to zero. Now, the value of `N` is located in `A4` and the next value that needs to be added is `N-1`, so `A1` is decremented by one. The command `SUB  A1,1,A1`, reading from left to right, takes the value in register `A1`, subtracts one from that value, and stores the new value in register `A1`.

Next, we proceed into a loop, which was coded as a *for* loop in our C program. Notice that there are no initial conditions that need to be met before entering the loop; therefore, this loop will execute at least once[5]. The loop is coded in four lines. The first line (line 8 in Figure 3) adds the current value in `A1` to the sum in `A4`. Then, `A1` is decremented by one. Finally, a decision is made using a branch (`B`) statement. If `[A1]` is not equal to zero, the program will branch back to the line with the label `LOOP` and continue to execute from there[6]. When the program returns to the branch statement again, it re-tests `A1` to see if it equals zero. Once A1 reaches zero, the program will no longer branch back to the label `LOOP`. Instead, it will continue on to the next instruction after the branch statement. Branch statements take six clock cycles to execute, so they have five no operation (`NOP`) instructions following them. This is to ensure that no other part of the program tries to use the `.S2` data path during the branch. As a result, the program execution will wait five clock cycles until the hardware decides whether to branch or not.

At this point, the function has calculated the sum and it is ready to return it to the function from which is was called. The value being returned by this function must be in register `A4`. The last instruction branches back to the calling function by branching to the memory address stored in register `B3`, where `B3` holds the execution state for the calling function.

Column Structure of Assembly Code

Assembly language programs have a structure that must be preserved. There are separate columns for each part of the code. The basic column structure is [1]

```
Label  ||  [ ]  Instruction  Unit  Operands  ; Comments.
```

The first column contains labels, such as `LOOP`. The second column designates instructions executing in parallel, which we will explore later in this lab. The third column specifies conditional registers associated with an instruction. The only registers that may be used as conditional registers are `B0`, `B1`, `B2`, `A1`, and `A2`. All of the instructions in assembly may be conditional. This means that any instruction (operating on any register) may or may not execute based on a value in a conditional register. The two conditions that may be tested are whether the value in

[5] In older styles of programming such as BASIC, loops that executed at least once were called *do-while* loops.

[6] To set up a loop that branches only when `A1` is equal to zero, replace `[A1]` with `[!A1]`.

a conditional register is equal to zero or not equal to zero. The fourth column contains assembly commands (instructions) like `ADD`, `SUB`, `MPY`, `MV`, etc. The fifth column specifies the functional unit, such as `.L1`, that the instruction will use. This column is optional. If you do not specify the data paths, the assembler will choose them for you, but it is good practice to assign these yourself. The sixth column specifies the registers that the instructions will operate on. Depending on the instruction, indirect addressing may be used to address the registers. Indirect addressing includes pre- and post-incrementing the value in a register and memory location offsets. The latter being used when the contents of a register point to the memory location of the data to be processed. For a more comprehensive treatment of indirect addressing, see either [1] or [7]. The last column is for appending comments. Any text on a line after a semicolon (`;`) in assembly code is a comment and will be ignored by the assembler. These columns do not have to line up perfectly, but at least one blank space must be left for each column that is empty. For ease of readability, we will vertically align the columns as appropriate.

Use of Functional Units

The functional units in `sum_asm_func.asm` are chosen based on the location of the registers being accessed. When the registers being accessed are located in data path A, the functional units receive the suffix 1 (as in `.L1`, `.S1`, `.M1`, or `.D1`). Similarly, when the registers being accessed are located in data path B, the functional units receive the suffix 2 (as in `.L2`, `.S2`, `.M2`, or `.D2`). Some instructions can only be implemented on certain data paths. For example the branch to register `B3` (return from the function) can only be done by the functional unit `.S2`. This is the case for most branch functions. The only exception is that branches to a label (such as `B  LOOP` in `sum_asm_func.asm`) can be done using either `.S1` or `.S2`. All other branch instructions must be done on `.S2`. For a list of instructions that can be implemented with each function unit, see [5].

Break Points for Program Debugging

In CCS, the current values of the data path registers may be observed while the program is executing. Reload the executable file `sum_asm.out` onto the DSK again to re-initialize the hardware. This can be done by selecting the 'File' pull down menu and then selecting 'Reload Program'. Open up the file `sum_asm_func.asm` in CCS. In the grey margin on the left double-click the mouse on lines 6 and 8. Red dots should appear on these lines, signifying break points. An executing program will stop at these break points, which will allow you to view the contents of the registers at these points in the program. Go to the 'View' pull-down menu, select 'CPU Registers', then select 'Core registers'. A window should pop up on the lower right part of the screen that shows the contents of the registers. Run the program. When execution stops, the value 3 should appear in registers `A1` and `A4`. Also, register `B3` should contain the execution state of the `main()` function at the time the program branched. In the window 'Disassembly' that was opened when you loaded the program, you should see the current program memory location. This should agree with the program counter (PC) register in your 'DSP core registers' window. Continue to step through the program. Pay attention at each break point. Make sure the correct value is stored in each register. Keep in mind that the instruction on the line of the break point has not executed yet. This method of stepping through code, line-by-line, can be very helpful when trying to debug an algorithm.

Use of Data Cross-Paths

Two data *cross-paths* are located in the DSP core that allow the registers in data paths A and B to share data (one data path in each direction). These are denoted by an `x` at the end of the functional units. For example, if you wanted to add the contents of `A4` to `B4` and store the result in `A7`, you would use the following code:

```
ADD  .L1x  A4,B4,A7; note the use of spaces for the
                   ; Label, ||, [ ], and operand columns
```

Here, the `1x` suffix is used to tell the assembler that the final result will be stored in a register located in data path A. If the result were to be stored in `B7` instead of `A7`, the following code would be used:

```
ADD  .L2x  A4,B4,B7
```

where the `2x` suffix tells the assembler that the result will be stored in a register in data path B. Since there are only two cross-data paths, there is a maximum of two instructions per cycle that can use cross-paths. In the case where two cross-path instructions are being perform in one clock cycle, they must write to different data paths (i.e. one of the functional units must have the suffix `1x` and the other must have the suffix `2x`). Figure 5 implements the assembly code of Figure 3 using cross-paths. For your own amusement, you can modify `sum_asm_func.asm` to use cross-paths and verify that you get the same result. Also, try stepping through the code line-by-line as before. Can you trace the algorithm through the different registers?

```
; sum_asm_func.asm using cross-paths and alternative branch instruction
; assembly function to find n+(n-1)+...+1+0

                  .def        _sum_asm_func  ; asm function called from C
_sum_asm_func:    MV    .L2x  A4,B1          ; setup N as loop counter in A1

LOOP:             SUB   .S2   B1,1,B1        ; decrement loop counter A1
                  ADD   .L1x  A4,B1,A4       ; accumulate in A4
      [B1]        B     .S1   LOOP           ; alternative branch unit .S1
                  NOP         5              ; five NOPs for delay slots
                  B     .S2   B3             ; return to calling routine
                  NOP         5              ; .S2 must be used here
                  .end
```

Figure 5: Listing of `sum_asm_func.asm` using data cross-paths and an alternate branch instruction

4 Calculating a Finite Sum using Linear Assembly*

Linear assembly code is very similar to assembly code except that data registers (A0-A15 and B0-B15) and `NOP` instructions are not specified by the programmer. Instead, they assigned by a highly optimized compiler. Figure 6 contains the linear assembly code for `sum_asm_func.asm` of Figure 3.

As with an assembly program, the first line (not including comments) of a linear assembly program begins with a definition (`.def`). Two new preprocessor directives are being used. They are `.cproc`, which signifies the beginning of the linear assembly function, and `.endproc`, which ends the linear assembly function. Line 5 of the function `sum_sa_func()` (`.cproc  N`) tells the

```
1.)   ; sum_sa_func.sa
2.)   ; linear assembly function to find n+(n-1)+...+1+0
3.)
4.)                     .def        _sum_sa_func   ; linear asm func called from C
5.)   _sum_sa_func:     .cproc      N              ; start of linear asm function
6.)                     .reg        sum,ctr        ; asm optimizer directive
7.)                     MV          N,ctr          ; setup loop counter in ctr
8.)                     MV          N,sum          ; accumulate total in sum
9.)
10.)  LOOP:             SUB         ctr,1,ctr      ; decrement loop counter
11.)                    ADD         sum,ctr,sum    ; accumulate in sum
12.)        [ctr]       B           LOOP           ; branch to loop if ctr # 0
13.)                    .return     sum            ; return sum to calling func
14.)                    .endproc                   ; end of linear asm function
```

Figure 6: Listing of `sum_sa_func.sa`. Adapted from [1].

compiler that one argument will be passed into the function and it will be labelled `N`. Line 6 tells the compiler that two registers will be needed to hold data and it labels these registers `sum` and `ctr`. The next two lines store the argument `N` into DSP core registers. From here, the function proceeds exactly as in `sum_asm_func.asm`. The commands work exactly like they do in assembly. The differences are that you do not have to specify the specific data registers, and you do not have to account for delay slots (NOPs). A highly optimized compiler will assign the best possible data register to each instruction and it will insert NOPs where appropriate. Here, the programmer is still allowed to direct the data through DSP core registers, but is not forced to specify the registers at each point. This will reduce development time. In more complicated programs, the choice of using C, linear assembly, or assembly code becomes a trade-off between coding effort and coding efficiency. Linear assembly in many cases can provide a good balance between the two.

As a side note on linear assembly code, the labels associated with the registers and names of the variables being passed to a linear assembly function are up to the programmer to choose. The same algorithm in Figure 6 could be coded as in Figure 7 below, taken from [1].

```
  ; sum_sa_func.sa  Linear assembly function called from C to find sum

                    .def        _sum_sa_func   ; Linear ASM func called from C
  _sum_sa_func:     .cproc      number         ; start of linear ASM function
                    .reg        a,b            ; asm optimizer directive
                    mv          number,b       ; set-up loop counter in b
                    mv          number,a       ; move number to a

 loop:              sub         b,1,b          ; decrement loop counter
                    add         a,b,a          ; n + (n-1)
       [b]          b           loop           ; branch to loop if count # 0
                    .return     sum            ; return sum to calling funct
                    .endproc                   ; end of linear asm function
```

Figure 7: Alternative way of coding `sum_sa_func.sa`. Adapted from [1].

Notice that the assembly commands do not have to be capitalized. Also, the names of the registers and the variables being passed can be labelled however you want to label them (provided you do not use any invalid characters). It is, however, recommended that you choose meaningful names for these registers so that your program is easy to follow. For example, in Figure 6, the register `ctr` is for the loop counter, the register `sum` holds the accumulated sum, and the input variable `N` is the number `N` being passed by the calling function.

4.1 Assignment 1

The factorial of a number N is defined to be $N! = N*(N-1)*\ldots*1$. Create a C program that calls a C function to calculate $N!$. Then, write the factorial function in both assembly and linear assembly. Explain your programs and include copies of your code. (HINT: Refer to [5] for the instruction `MPY`, which will multiply two numbers in assembly.)

5 Multiply and Accumulate in Assembly*

The process of multiplying two numbers and adding them to a sum is known as a multiply and accumulate (MAC) operation. In digital signal processing, most algorithms require MAC operations. In the previous example, we introduced C callable assembly functions that accumulated a value in a DSP core register. The next step is to multiply two values and accumulate them in a register (MAC). To introduce this idea of multiply and accumulate (MAC), we will explore a C callable assembly function that will calculate the *standard inner product* of two vectors.

We use the notation $\underline{x} \in \Re^n$ to describe the column vector $\underline{x}$ that contains n real numbers. Given two vectors $\underline{x}, \underline{y} \in \Re^n$, we define the standard inner product to be[7]

$$\langle \underline{x}, \underline{y} \rangle = \underline{x}^T \underline{y} = \begin{bmatrix} x_1 \ldots x_n \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = x_1 y_1 + \ldots + x_n y_n = \sum_{i=1}^{n} x_i y_i \quad (1)$$

Thus, the inner product of two vectors is the sum of the products of their elements.

A C Program for Calculating an Inner Product on the C6713 DSK

A vector in an inner product space is an array in software and a set of memory registers in hardware. An array of data is stored in adjacent memory registers on the DSP chip, so the idea of a vector being a collection of elements in a specific order is similar to an array, which is a collection of elements stored in memory registers in a specific order. Lets explore this idea on the DSK.

Examine the C code in Figure 8 for computing the inner product $\langle \underline{x}, \underline{y} \rangle$ for $\underline{x} = [1, 2, 3, 4]^T$ and $\underline{y} = [0, 2, 4, 6]^T$. Pay attention to lines 7 and 8, which define the two data arrays `x` and `y`. In this example, the function `inprod_C_func()` is being passed three arguments and will return one. When a function is called, only one value for each argument can be passed to the function. The memory locations of `x[0]` and `y[0]` are passed to the function. This fact is hidden in C programming, since the function `inprod_C_func()` treats the variables `a` and `b` as though they contain the entire array. As we will see in the assembly code implementation of this C function, this is not the case. Within the function `inprod_C_func()`, there are two local variables, namely `sum` and `i`. The variable `i` is used as a loop counter and array index for computing `ncount` multiply and accumulate operations. The variable `sum` is used to accumulate. Notice that the

[7] The term inner product is used to describe a function of two vectors that has certain properties. The inner product $\langle \underline{x}, \underline{y} \rangle = \underline{x}^T \underline{y}$ is referred to as the *usual inner product* or *standard inner product*. The notation $\underline{x}, \underline{y} \in \Re^n$ indicates that each of the vectors $\underline{x}$ and $\underline{y}$ contains n real numbers. That is $\underline{x} = [x_1 \ldots x_n]^T$ belongs to $\Re^n$, the set of real n-tuples. For a more comprehensive treatment of inner products, see [2].

```
/*1 */ // inner_product_C.c
/*2 */ // Calculates the inner product of two vectors
/*3 */
/*4 */ int inprod_C_func(short *a, short *b, int ncount);  // function prototype
/*5 */ #include <stdio.h>                                  // for printf
/*6 */ #define N 4                                        // # of data in each nx1 vector
/*7 */ short x[N] = {1,2,3,4};                            // define elements in 1st vector
/*8 */ short y[N] = {0,2,4,6};                            // define elements in 2nd vector
/*9 */ short inner_product;                               // to store inner product
/*10*/
/*11*/ int inprod_C_func(short *a, short *b, int ncount)   // inner product function
/*12*/ {
/*13*/   int sum = 0;                                     // init sum
/*14*/   int i;                                           // local var used in for loop
/*15*/
/*16*/   for (i = 0; i < ncount; i++)
/*17*/   {
/*18*/      sum += a[i] * b[i];                           // sum of products
/*19*/   }
/*20*/   return(sum);                                     // return sum as result
/*21*/ }
/*22*/
/*23*/ main()
/*24*/ {
/*25*/   inner_product = inprod_C_func(x,y,N);            // call inner_prod function
/*26*/   printf("<x,y> = %d (decimal) \n", inner_product); // print result
/*27*/ }
```

Figure 8: Listing of `inner_product_C.c`. Adapted from [1].

multiply and accumulate operation is done in one line (see line 18 of Figure 8)[8]. Once the total has been accumulated in the variable `sum`, the value in `sum` is returned to the calling function. NB: The variables `a`, `b`, and `ncount` are local variables. A local variable of a function is a variable that can only be accessed while executing the function.

An Assembly Program for Computing an Inner Product on the C6713 DSK

Consider Figure 9 below, which is an assembly version of the C function `inprod_C_func()` in Figure 8. Create a project `inner_product_asm`' and download the accompanying files from the webpage. In CCS, build this project, reset the DSK, load `inner_product_asm.out` onto the DSK, and run this project to verify that if gives the same results as `inner_product_C.out`. Open up and examine the file `inprod_asm_func.asm` (shown in Figure 9).

When an array is passed to an asm function, the first memory location of the array is passed to the function. Registers `A4` and `B4` hold the first memory locations for arrays `x` and `y` respectively.

[8] The command `sum += a[i] * b[i];` is shorthand notation for the multiply and accumulate operation `sum = sum + (a[i] * b[i]);`. The MAC operation may be coded either way in C.

```
1.)    ; inprod_asm_func.asm
2.)    ; Multiply two arrays. Called from inner_prod_asm.c
3.)    ; A4=x address,B4=y address,A6=count(size of array),B3=return address
4.)
5.)                        .def         _inprod_asm     ; inner product function
6.)                        .text                        ; text section
7.)    _inprod_asm         MV           A6,A1           ; move loop count to A1
8.)                        ZERO         A7              ; init A7 for accumulation
9.)
10.)   LOOP                LDH          *A4++,A2        ; A2=(x). A4 as address pointer
11.)                       LDH          *B4++,B2        ; B2=(y). B4 as address pointer
12.)                       NOP          4               ; 4 delay slots for LDH
13.)                       MPY   .M1x   B2,A2,A3        ; A3 = x * y
14.)                       NOP                          ; 1 delay slot for MPY
15.)                       ADD          A3,A7,A7        ; sum of products in A7
16.)                       SUB          A1,1,A1         ; decrement loop counter
17.)       [A1]            B            LOOP            ; branch back to LOOP until A1=0
18.)                       NOP          5               ; 5 delay slots for branch
19.)
20.)                       MV           A7,A4           ; A4=result A4=return register
21.)                       B            B3              ; return from func to addr in B3
22.)                       NOP          5               ; 5 delay slots for branch
```

Figure 9: Listing of the function `inprod_asm_func.asm`. Adapted from [1].

The third argument passed to the function is the number of elements in each array. Since this is not an array being passed to the function, the predefined value `N` = 4 is being passed to register `A6`. Arrays are stored in contiguous blocks of memory, so accessing an array element may be done by locating the the memory address of the first element, plus an offset. The offset value will depend on the type of data being accessed. The memory on the C6713 chip is byte addressable (i.e. split into 8-bit blocks). In the case of 16-bit shorts, a single 16-bit number will be stored in two adjacent memory locations, with the most significant 8 bits in one location and the least significant 8 bits in another. For example, if `A4` holds the memory location of the first element of an array of 16-bit numbers, then the memory address of the second element would be `A4+(2)`, the memory address of the third element would be `A4+(4)`, and so on. To access the *contents* of a memory location, *pointers* are used. In assembly code the notation `*A4` points to the contents of the memory location specified by the address (number) in `A4`. By convention, the `*` operator is used to point to a memory location and `&` is used to refer to the address of a variable. In ANSI C, the two operators are used together to indirectly address data. In CCS, the `&` does not need to proceed an array variable that is being passed to a function. (NB: A register that holds the memory *location* of the desired value, but not the *value* itself is referred to as a *pointer*.) This will be illustrated shortly.

You should recognize most of the commands in Figure 9. Here, register `A3` is used to hold the result of the multiply operation and register `A7` is used to accumulate the sum of products. Register `A7` is the equivalent of the variable `sum` in `inprod_C_func`, see Figure 8. Once the loop is completed, the final result is moved from `A7` to `A4` so that it will be returned to the calling function. Finally, a branch back to the calling routine (branch to `B3`) is made.

In Figure 9, within the *loop*, the lines 10 and 11 may be new to you. These two lines load the current values in arrays `x` and `y` into temporary registers. Consider line 10: 'LDH   *A4++,A2'. The command 'LDH' stands for *load half word* or load 16 bits worth of data. Recall, the DSP core registers are 32 bit registers, so the (full) word length is 32 bits. Here, `A4` contains the memory location of the current value in the array `x`, so the notation `*A4++,A2` tells the assembler to point to the memory location stored in `A4`, and load the value that is stored in the memory address of A4 into the register A2. The ++ after A4 tells the assembler to increment the memory address in A4 to the next valid memory address. Since the values that are stored in the array x are of type short (16-bit signed integers), they take up two bytes (8 bits = 1 byte) of memory[9]. Therefore, this increment operator adds two to the current value of A4. Since the ++ is placed after the register A4, the assembler will post-increment the value in A4. This means the value in A4 is incremented after the load instruction is executed. This command could have been split into two lines as follows:

```
LDH  *A4,A2
ADD  A4,2,A4
```

If the ++ had been placed before the register A4 (e.g. LDH *++A4,A2), then the address in A4 would have been incremented before the load (pre-incremented). This could be split into two lines of code as follows:

```
ADD  A4,2,A4
LDH  *A4,A2
```

The use of the ++ prefix/suffix is common practice when using indirect addressing. For a more comprehensive treatment of indirect addressing, see [1]. Line 11 in Figure 9 is similar to line 10 except that it loads the data in vector y, by pointing to the memory address stored in register B4. Line 13 uses the cross data paths between registers A and B. Since the final value of the multiply operation will be stored in data path A, the functional unit is .M1x, rather than .M2x.

5.1 Assignment 2

Examine the function that is listed in Figure 9. Do the following:

1. add functional units to each line of inprod_asm_func.asm where applicable; turn in your updated code.
2. explain the function inprod_asm_func.asm line-by-line in your own words.

5.2 Assignment 3

Re-load the project inner_product_asm onto the DSK. Open the file inprod_asm_func.asm and insert break points at lines 10,13, and 21. Keep track of the contents in registers A2, A3, A4, A7, B2, and B4 at each loop iteration. Explain what is happening in the hardware.

5.3 Assignment 4

Download the linear assembly version of the inner product function from the class webpage (project inner_product_sa). Implement it on the DSK. Look at the code for the function inner_prod_sa_func(). Explain the similarities and differences between the linear assembly function and the assembly function for calculating an inner product.

[9] Memory is broken up into 8 bit blocks known as bytes. Since there are 2^{32} 32-bit addressable memory locations and each location is one byte, there are $4 * 2^{30}$ bytes = 4Gb of addressable memory. NB: 2^{10} bytes = 1kb, 2^{20} bytes = 1Mb, and 2^{30} bytes = 1Gb, with the understanding that 1kb = 1024 bits, 1Mb = $(1024)^2$, and so on.

6 Optimizing Assembly Code

Optimizing assembly code means scheduling events so that the DSP core is being utilized efficiently on every clock cycle. In general, this will be difficult and time consuming, especially for complicated algorithms. In this section, we will explore two optimization techniques: assigning instructions to execute in parallel (during the same clock cycle on different functional units) and inserting instructions into clock cycles that are designated no-operation (NOP). Other more powerful and complicated procedures are available. So far, we have included all the NOP instructions required to idle the hardware while instructions that take more than one clock cycle are executed. These are required so that a given functional unit is not accessed while it is executing a multiple clock cycle instruction from a previous clock cycle. Without this protection, an instruction that tries to access a functional unit that is in use will be ignored, which will alter the intended algorithm and may cause the program to crash.

Since code optimization can be an advanced topic, we will only go through the basics, using one example. Consider the function in Figure 9 and refer to Figure 10. There are five things that we can do to optimize this code.

```
1.)   ; inprod_asm_func_opt.asm
2.)   ; Multiply two arrays. Called from inner_prod_asm.c
3.)   ; A4=x address,B4=y address,A6=count(size of array),B3=return address
4.)   ; uses parallel instructions to optimize code.
5.)
6.)                             .def        _inprod_asm_func_opt  ; inner product function
7.)                             .text                             ; text section
8.)   _inprod_asm_func_opt      MV     .L1   A6,A1                ; move loop count -->A1
9.)           ||                ZERO   .S1   A7                   ; init A7 for accumulation
10.)
11.)  LOOP                      LDH    .D1   *A4++,A2             ; A2=(x). A4 as address pointer
12.)          ||                LDH    .D2   *B4++,B2             ; B2=(y). B4 as address pointer
13.)                            SUB    .S1   A1,1,A1              ; decrement loop counter
14.)              [A1]          B      .S2   LOOP                 ; branch to LOOP after add
15.)                            NOP          2                    ; 2 extra delay slots for LDH
16.)                            MPY    .M1x  B2,A2,A3             ; A3 = x * y
17.)                            NOP                               ; 1 delay slot for MPY
18.)                            ADD    .L1   A3,A7,A7             ; accum. in A7, then branch
19.)
20.)                            B      .S2   B3                   ; return from func to addr in B3
21.)                            MV     .L1   A7,A4                ; A4=result A4=return register
22.)                            NOP          4                    ; 4 extra delay slots for branch
```

Figure 10: Listing of an optimized `inprod_asm_func.asm`. Adapted from [1].

The first set of modifications to Figure 9 designates instructions to execute in parallel. Instructions that have double bars (`||`) in their second column are executed in parallel with the instruction(s) on the previous line(s). The following instructions are designed to operate in parallel:

1. The move instruction (`MV` on line 8) sets up the loop counter in `A1` and the zeroing instruction (`ZERO` in line 9) initializes (sets to zero) the accumulating register `A7`. Each of these instructions takes only one clock cycle and is being executed on a different functional unit, so these instructions can execute in parallel.

2. Each of the two load half-word instructions (`LDH`) takes five clock cycles and operates on a different data path, so these instructions can execute in parallel. (NB: Only the `.D` functional unit can be used to load/store data, so a maximum of two load/store operations is allowed to execute during a given clock cycle.)

The second set of modifications utilizes the delay slots (`NOP`) when possible. In particular, they take advantage of the delay slots associated with branch instructions. Branch instructions take six clock cycles to execute, and the branch will not occur until the sixth clock cycle has been completed. This leads to the following rearrangements:

1. The loop counter (`A1`) needs to decrement each time the loop is executed, but where in the loop cycle is not important. Since the decrementing of the loop counter is done by functional unit `.S1` and only takes one clock cycle, it can execute during one of the five clock cycles needed to load a half-word using the `.D` functional unit.

2. The branch instruction will take six clock cycles to execute. Since the branch is a conditional branch based on the decremented value of `A1` it must occur after `A1` is decremented. Therefore, it is moved to line 14. Now, two of four delay slots have be filled, so only 2 more `NOP` instruction are required

3. The final branch statement back to the calling function will also take six clock cycles, but the final moving of the contents from the accumulation register `A7` to the returning register `A4` will take only one cycle. The value in `A4` after the sixth clock cycle will be the value returned to the calling function. Therefore, the branch statement is started in line 20 of Figure 10. Then, the final value is moved into `A4` (the predetermined return register) in line 21, which saves one clock cycle.

Now, let's take a closer look at the *optimized loop* in Figure 10. The two load half-word commands will begin during one clock cycle and it will be four more clock cycles before registers `A4`, `B4`, `A2`, and `B2` or functional units `.D1` and `.D2` will be available. During this time, the loop counter (`A1`) will be updated on `.S1`, then a six clock cycle branch will begin to process on `.S2`. After an additional two `NOP` instructions, the load instructions will have finished, and registers `A2` and `B2` will be available for multiplying. During the next two clock cycles, the two registers `A2` and `B2` will be multiplied together and the result will be stored in `A3`. Finally, in one clock cycle, the new value in `A3` will be accumulated in `A7` and the branch instruction will take effect, sending the program back to the two load half-word instructions. The loop will repeat itself until it is completed. It should be noted that the `NOP` instructions on lines 15 and 17 are required since the instructions on the lines following them use registers that will not be ready until after the clock cycles assigned by the `NOP` instructions have been exhausted.

This optimization requires careful planning of register use and tracking of branch statements. To optimize larger programs, flow charts, dependency graphs, and schedule tables need to be constructed. Examples of these can be found in [1] and [4]. Other optimization methods not described here include the use of a C optimized compiler, intrinsic C functions, and software pipelining. Intrinsic functions and the optimized compiler are for C programs only, and software pipelining can only be used in assembly code. In future labs, we will use intrinsic functions, but will not explore them. If you feel ambitious, you may use C compiler optimizations on your programs.

7 End Notes

An FFT consists of the efficient computation of N inner products, in parallel, in lab or real-time. An FIR filter consists of an unbounded number of inner products, computed sequentially in real-time. Consequently, the insight we have gained in this lab for computing inner products will be invaluable in subsequent labs.

Lab Suggestions for Course Project

- The most advanced and most efficient way to optimize code is to schedule events using software pipelining. An advanced lab would explain software pipelining and require the student to optimize all assembly language programs in the course, using software pipelining. For information on software pipelining, see [1], [7], or [5].
- Another way to schedule events, as well as interface the DSP chip in real-time, is to use the DSP/BIOS. The DSP/BIOS is a software kernel that can be used to schedule events on the DSK and can access any part of the internal memory without interfering with a running application. There is much information available from TI's website and in the CCS help menus on using the DSP/BIOS.

References

[1] R. Chassaing, *Digital Signal Processing and Applications with the C6713 and C6416 DSK*. Wiley, New Jersey, 2005.

[2] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 1985.

[3] Greg Perry. *Absolute Beginners Guide to C*. Pearson Education, 1984.

[4] Texas Instruments, Dallas, TX. *Guidelines for Software Development Efficiency on the TMS320C6000 VelociTI Architecture, SPRA434*, 1998.

[5] Texas Instruments, Dallas, TX, *TMS320C6000 CPU and Instruction Set Reference, SPRU189G*, 2006.

[6] Texas Instruments, Dallas, TX. *TMS320C6713B Floating-Point Digital Signal Processor, SPRS294B*, 2005.

[7] Steven A. Tretter. *Communication Design Using DSP Algorithms: With Laboratory Experiments for the TMS320C6701 and TMS320C6711*. Kluwer Academic/Plenum Publishers, New York, 2003.