Design Assignment 2:
Assembly Language Programming

THE COLLEGE OF NEW JERSEY

Alexis Adie and Madison Mastroberte

ELC 411-01: Embedded Systems

Submitted:
October 11th 2017

Design Assignment 2: Assembly Language Programming

Alexis Adie and Madison Mastroberte

Department of Electrical and Computer Engineering
The College of New Jersey
2000 Pennington Road, Ewing, NJ 08618, USA
(adiea1, mastrom7)@tcnj.edu

I. Introduction

The lab consisted of a dual implementation of ‘C’ and assembly codes that produce the same results. The programs were coded to simulate the dot product, inner product, of two 16-bit data arrays. A ‘C’ code was provided to students and the assembly was first created by students. Utilizing the debugger tool in PSoC Creator, students tested and checked the logical behavior of their assembly code. Once the debugger values matched the ‘C’ language results, this portion of the lab was completed. Next students retrieved the compiler’s assembly version of the ‘C’ code, and analyzed the code. Overall, the lab introduced students to the debugging design environment, while showing how to generate a specific assembly code to mimic a ‘C’ subroutine code.

II. Methodology

A. *Pre-Class: Manual Assembly Code*

Students began by generating assembly code by mimicking the ‘C’ code given in the first step. The parameters were passed in the R0, R1, and R2 registers, however return values were deposited in R0.

B. *Part I: Debugging Manual Assembly Code*

Students began by downloading the canvas file, main_for_asm_project.c and pasting the code into a main.c file. Once the initial code was replaced, a breakpoint was set at line 34; the location is where the sum is declared. The compiler was then set to an optimization strategy of ‘None.’ Once set, the project was built and run under debug mode. The debugger will run each step, using F5, until the breakpoint. Next the disassembled code was viewed. The code was then copied and then pasted into “inner_prod_gcc.s.”

C. *Part II: Analysis of Compiled Assembly Code*

Students began the second portion of the lab by creating a new assembly language source file (GNU ARM Assembly file). The initial boilerplate code was changed to implement the inner_prod_asm code. Once tested, and put into debug mode, the six variables were place on the Watchlist. These values are shown in Figure 1.

Once completed, two digital output pins were added to the design. Then using these pins, the time spent in the ‘C’ code and manual assembly code was measured. The pins were then used to measure triggered waveforms and time for the signal.

Lastly, the code was recompiled with a ‘Speed’ optimization level. Then the time spent in the optimized ‘C’ code and manual assembly code was remeasured.

III. Results

A. *Part I*

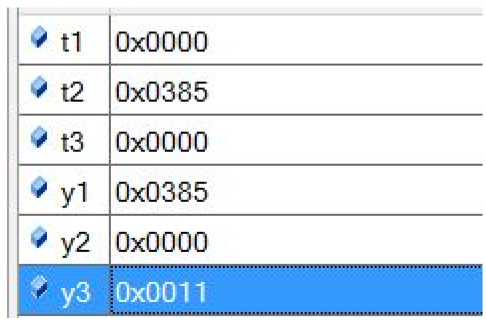

Figure 1. Screenshot watchlist for debugged assembly code, including six variables.

Due to errors within the code created, the developers were unable to achieve a match between the variables as expected. The issues stemmed from the loop section of the code, which the team confirmed by changing the code slightly with each trial.

Table 1. Time spent in the ‘C’ version of the function with varying compiler strategies.

Opt. Strategy	‘C’ Time (ms)	Assem. Time (ms)
None		
Speed		
Manual		

B. *Part II*

In order to implement the correct outputs, the students utilized the button and pull-up resistor configuration. Figure 2 depicts the schematic of the PSoC 5LP on-board button. The button acts as an open circuit when unpressed, and ground when pressed. In order to implement it on the PSoC the input must be set to have a pull-up resistor. This forces its output to high when unpressed rather than leaving it as an open circuit. Since the output of the button is high when unpressed and low when pressed its value must be inverted when used in code.

Figure 2. Measured positive time of dual scope trace on rising edge of signal.

Figure 3a-b, depict the several states of the LEDs. Figure 3a is the state when the first LED consistently blinks and the secondary LED is unlit since the button is unpressed. Figure 3b is the state when the first LED remains constantly blinking and the button is pressed. By pressing the button, the secondary LED is lit.

IV. Discussion

A. Part I

Figures 1a-c show the output waveforms from the digital out port of the PSoC board. Each output signal is a square wave at a different frequency with some abnormalities as the amplitude changes. These over dampened oscillations are most likely due to the speed that the PSoC is trying to change its output. When it flips it drives the signal to the new state as quickly as it can, but the signal overshoots due to the speed. At lower frequencies these abnormalities become less apparent.

Different compiler optimization settings were also tested to determine if the maximum frequency changed with different settings. Three different settings were selected, 'none', 'speed', and 'minimal'. None compiles the code as is, and the compiler doesn't take advantage of ways to save space or increase speed. For this setting the PSoC board was able to generate a maximum frequency of 197.4kHz. The next optimizer setting tested was speed. In this case the compiler found the best way to generate machine code for it to run the quickest. This setting generated the highest frequency at 473.7kHz. The last setting that was tested was minimal. This generated a lower frequency at 450.7kHz. These three frequencies were calculated from the microseconds per division lines. The measured values from the National Instruments box are inaccurate.

B. Part II

One LED was set to blink once every second while the other was set to change states whenever a button was held down. This functioned as intended but there is one small detail to note. When the button was pressed, the LED associated with it would not change state until the set cycle LED changed. This is due to the nature of the infinite loop in the code. An interrupt would be necessary to make the button LED change exactly as the button is pressed.

V. Conclusion

Overall, the lab demonstrated the ability to interface 'C' and assembly to successfully perform the dot product and interact with another. In Part I, students implemented two codes, 'C' and assembly, that produce the dot product. Students then debugged the code to match output values of the 'C' code. Next, students analyzed the various outputted assembly codes with differing optimization strategies. Overall, the lab introduced students to the debugging design environment, while showing how to utilize assembly code to create the same behaviors as a 'C' subroutine. The lab was successful and gave students insight regarding the capabilities of PSoC Creator.

VI. Appendix

Part I: Madison Mastroberte's Assembly Code (Prior to Lab)

```
;initialize the variables in registers
 Assume  R0 = i
   R1 = sum
   R2 = n
   R3 = h
   R4 = x
;set registers to 0
 MOV R0, #0 ; i = 0
 MOV R1, #0 ; sum = 0

loop
 CMP R0, R1 ; i < n
 LDRSH R3, [R0]
 LDRSH R4, [R0]
 MUL R1, R3, R4   ; sum += (h[i] * x[i])
 ADD R0, #1 ; ++i
 RSR R1 #16 ; sum = sum >> 16

 BX LR
end

r0 = h , r1 = x, r2 = n

;initialize the variables in registers
 Assume  R0 = h
   R1 = x
   R2 = n
   R3 = i
   R4 = sum

;set registers to 0
 MOV R3, #0 ; i = 0
 MOV R4, #0 ; sum = 0
 PUSH R4

loop:
 CMP R0, R1 ; i < n
 LDRSH R3, [R0]
 LDRSH R4, [R0]
 MUL R1, R3, R4   ; sum += (h[i] * x[i])
 ADD R0, #1 ; ++i
 RSR R1 #16 ; sum = sum >> 16

 POP R4
 BX LR
end
```

r0 = h , r1 = x, r2 = n

Part II: Alexis Adie's Assembly Code (Prior to Lab)

```
 /*   Assume
            R0 = i
            R1 = x
            R2 = h
            R3 = n
            R4 = sum
            R5 = temp
*/
//set registers to 0
      MOV R3, #0         //i = 0
      MOV R4, #0         //sum = 0

      PUSH {R4}
      PUSH {R5}
loop:
      CMP R0, R3         //i < n
      LDRSH R2, [R0]    //h[i]
      LDRSH R1, [R0]    //x[i]
      MUL R5, R1, R2    //temp=(h[i] * x[i])
      ADD R4, R4, R5     //sum += (h[i] * x[i])
      ADD R0, #1         //++i
      LSR R1, #16        //sum = sum >> 16
      BX LR
//end
      .endfunc
      .end
```

Part III: Debugged Assembly Code

```
 //initialize the variables in registers
 /*   Assume
            R0 = h
            R1 = x
            R2 = n
            R3 = i
            R4 = sum
            R5 = temp
*/

//set registers to 0
      MOV R3, #0         //i = 0
```

```
        MOV R4, #0          //sum = 0
//  B test

        PUSH {R4,R5}
loop:
        CMP R0, R1          //i < n
        //BEQ
        LDRSH R3, [R0]     //h[1]
        LDRSH R1, [R0]     //x[i]
        MUL R5, R1, R3     //temp=(h[i] * x[i])
        ADD R4, R4, R5      //sum += (h[i] * x[i])
        ADD R0, #1          //++i
        ASR R1, #16         //sum = sum >> 16
        B loop

        POP {R4}
        BX LR
```

Part IV: inner_prod_gcc.s Code with Comments

```
0x00000084 <inner_prod>:

    31: // Inputs:    h - pointer to array of int16_t values, length n
    32: //            x - pointer to array of int16_t values, length n
    33: // Returns:   [x (dot) h] >> 16, as an int16_t value
    34: int16_t inner_prod( int16_t *h, int16_t *x, int n )
    35: {

0x00000084 push    {r7}          //Stores r7 to the top of the stack
0x00000086 sub     sp, #1c       //Creates stack frame
0x00000088 add     r7, sp, #0    //Uses r7 as the “frame pointer”
0x0000008A str     r0, [r7, #c] //Stores r7 into array h with offset of 0x0C
0x0000008C str     r1, [r7, #8] //Stores r7 into array x with offset of 0x08
0x0000008E str     r2, [r7, #4] //Stores r7 into array n with offset of 0x04

    36:     int i;
    37:     int32_t sum = 0;

0x00000090 movs    r3, #0       //r3=sum=0
0x00000092 str     r3, [r7, #10]//Stores sum at offset of 0x10

    38:
    39:     for (i = 0; i < n; ++i)

0x00000094 movs    r3, #0       //r3=i=0
0x00000096 str     r3, [r7, #14]//Stores i at offset of 0x14
0x00000098 b.n     c4 <CYDEV_PICU_SIZE+0x14>//ignored as per instructions
```

```
    40:     {
    41:     sum += (h[i] * x[i]);

0x0000009A ldr     r3, [r7, #14]//Loads i from frame offset 0x14 to r3
0x0000009C lsls    r3, r3, #1   //Logical shift left r3 by 1
0x0000009E ldr     r2, [r7, #c] //Loads h from frame offset 0x0C to r2
0x000000A0 add     r3, r2       //h[i]
0x000000A2 ldrsh.w r3, [r3]     //Loads halfword r3
0x000000A6 mov     r1, r3       //r1=[h[i]
0x000000A8 ldr     r3, [r7, #14]//Loads i from frame offset 0x14 to r3
0x000000AA lsls    r3, r3, #1   //Logical shift left by 1
0x000000AC ldr     r2, [r7, #8] //Loads x from frame offset 0x08 to r2
0x000000AE add     r3, r2       //x[i]
0x000000B0 ldrsh.w r3, [r3]     //Loads halfword r3
0x000000B4 mul.w   r3, r3, r1   //r3=h[i]*x[i]
0x000000B8 ldr     r2, [r7, #10]//Loads sum with offset 0x10 to r2
0x000000BA add     r3, r2       //r3=sum+(h[i]*x[i])
0x000000BC str     r3, [r7, #10]//Stores new sum into array r3 with offset 0x10

    34: int16_t inner_prod( int16_t *h, int16_t *x, int n )
    35: {
    36:     int i;
    37:     int32_t sum = 0;
    38:
    39:     for (i = 0; i < n; ++i)

0x000000BE ldr     r3, [r7, #14]//Loads i from r3 with offset of 0x14
0x000000C0 adds    r3, #1       //i=i+1
0x000000C2 str     r3, [r7, #14]//Stores i back into r3
0x000000C4 ldr     r2, [r7, #14]//Loads i
0x000000C6 ldr     r3, [r7, #4] //Loads n
0x000000C8 cmp     r2, r3       //Compares i and n
0x000000CA blt.n   9a <inner_prod+0x16>//i<n

    40:     {
    41:     sum += (h[i] * x[i]);   // accumulate each of the 'n' product terms
    42:     }
    43:     sum = sum >> 16;         // right shift to normalize

0x000000CC ldr     r3, [r7, #10]//Loads sum from r3
0x000000CE asrs    r3, r3, #10  //Arithmetic shift right by 16
0x000000D0 str     r3, [r7, #10]//Stores the new sum into r3

    44:
    45:     return (int16_t) sum;

0x000000D2 ldr     r3, [r7, #10]//Loads sum from r3
0x000000D4 sxth    r3, r3        //Returns sum
```

```
    46: }

0x000000D6 mov     r0, r3       //Set r0 to sum
0x000000D8 adds    r7, #1c      //Add 1c to r7
0x000000DA mov     sp, r7       //sp=r7
0x000000DC pop     {r7}         //takes r7 from the top of the stack
0x000000DE bx      lr           //branch
```

Part V: inner_prod_asm.s Code with Comments

```
            .syntax unified
            .text

            .global inner_prod_asm
            .func inner_prod_asm, inner_prod_asm
            .thumb_func

      inner_prod_asm:
       //initialize the variables in registers
       /*   Assume
                  R0 = h
                  R1 = x
                  R2 = n
                  R3 = i
                  R4 = sum
                  R5 = temp
      */

      //set registers to 0
            MOV R3, #0        //i = 0
            MOV R4, #0        //sum = 0
      //  B test

            PUSH {R4}
            PUSH {R5}
      loop:
            CMP R0, R1        //i < n
            LDRSH R3, [R0]    //h[1]
            LDRSH R1, [R0]    //x[i]
            MUL R5, R1, R3    //temp=(h[i] * x[i])
            ADD R4, R4, R5    //sum += (h[i] * x[i])
            ADD R0, #1        //++i
            ASR R1, #16       //sum = sum >> 16

            POP {R4}
            BX LR
      //end
            .endfunc
```

```
            .end

//set registers to 0

      PUSH  {R4,R5,R6}         //Stores regs to the top of the stack
      MOV   R3, #0      //sum = 0
      MOV   R4, #0      //i=0


   B test

loop:

      ADD   R0, R0, R4         //h[i]
      ADD   R1, R1, R4         //x[i]
      MUL   R5, R0, R1  //h[i]*x[i]
      ADD   R3, R3, R5  //sum= sum + (h[i]*x[i])
      ADD   R4, R4, #1         //i++
      STR   R4, [R4]
      STR   R3, [R3]
   B test


test:
      CMP   R4, R2      //i < n
      BLT   loop
      ASR   R3, #16     //shift right by 16
      MOV   R0, R3      //r0=sum
      POP   { R4, R5, R6}
      SXTH   R0, R3
      BX    LR
//end
      .endfunc


      .end
```