ECE 4270 - Computer Architecture

Lab 2 - MIPS Assembly to Hex and Assembly Problem Solving

Brandon Gallo and Jacob Higgins

Group 4

10/8/2020

a. Objective

The objective of this assignment was to convert a text file full of assembly MIPS instructions into their hexadecimal representation. Also a big part of the lab was to write two MIPS assembly programs. One program would perform a bubble sort, and one would find the fibonacci number of some given value. The assembly program can be used as an input to the first part of the lab, and the hex output can be used as an input to the solution to Lab 1.

b. Milestones & Work Distribution

This lab has two fairly distinct parts that are intended to work together once both are complete. For the first part, the assembly to hex converting, the first milestone was correctly reading in strings from a .in file and breaking up all parts of the string removing the '$' and commas from the string. Then it was on to building a 4 byte hex representation of the instruction. The order of milestones for getting this done went as follows: converting the instruction name into its hex representation, then converting each register to its hex representation, then converting the immediate or offset areas to their hex representation. Lastly, the completed hex representation was written to the terminal and to an output file to be used as an input to the MIPS simulator from lab 1.

As far as who did what, help and discussion was had between the two parts of the lab throughout, though we split up the two parts for one to focus more on. I, Jacob H, focused mainly on the reading, parsing, converting, and writing, of assembly instruction strings from the input text file, to writing them to the output.txt file. This included all below mentioned implementations regarding converting the instruction to hex representation. I, Brandon G, focused on the assembly language instructions for the bubble sort and fibonacci problems that were then converted into hex and read by the simulator. This involved converting a C algorithm into assembly language based on the MIPS instruction set architecture. For testing, I converted each instruction individually into hex format and ran it through the working simulation program to see what was going on in memory and the registers.

c. Implementations

Implementations for the assembly to hexadecimal portions of the lab first consisted of converting the instruction name into its hex representation. This was done by many 'if statements' checking for a string match of all instructions with the current string representation of the instruction name. If a match was found, the 4 byte hex representation was OR'd with the proper value that represented that instruction in hex. While the many 'if statements' could have been used again for scanning for the register string name, instead a faster method of using a hash table was implemented. So a hash table is created and then initialized in main with all 32 register

names and their corresponding hex value, so that when it's time to search for a register string name, e.g. “$v0”, it can be hashed and mapped in close to O(n) time, instead of many ‘if statements’. These were the two important implementations revolving around handling the parsed string representations of the instruction from the text file.

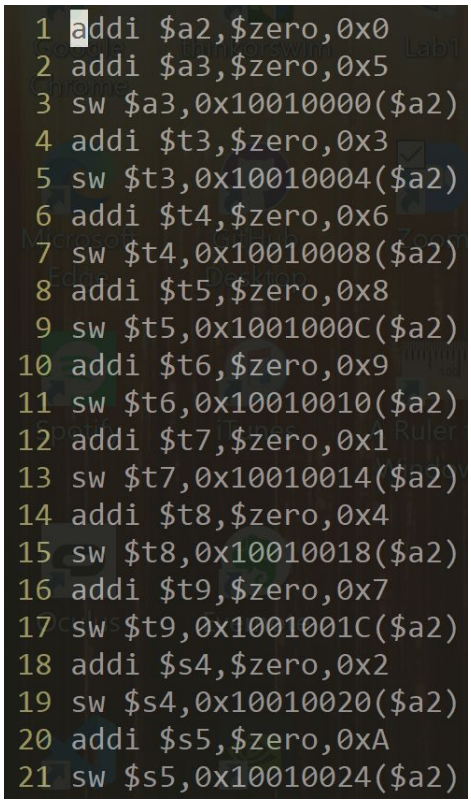
The Problem 1 and 2 implementations involved creating a bubble sort algorithm and fibonacci sequence using the MIPS instruction set architecture. By using I,J, and R instructions, this was achieved by loading and storing variables into memory and registers respectively, in addition to taking advantage of branch instruction to act as if statements in a traditional C program. With a good understanding of how registers work with memory (and vice versa), in addition to using the R4400 MIPS manual for the functionality of each instruction, a working code was able to be implemented.
Below is the implementation of the Bubble Sort Algorithm in MIPS assembly language.
The pseudocode shows step by step what this program should do as each instruction is called.

This first picture shows the implementation storing the array values sequentially in memory at address location 0x0 and initializing the base to 0 in line 1.

```
 1 addi $a2, $zero,0x0 //initialize the base
 2 addi $a3, $zero,0x5 //load first array value into register a3
 3 sw $a3, 0x0($a2)    //store array values sequentially in memory
 4 addi $t3,$zero,0x3
 5 sw $t3,0x4($a2)
 6 addi $t4,$zero,0x6
 7 sw $t4,0x8($a2)
 8 addi $t5,$zero,0x8
 9 sw $t5,0xC($a2)
10 addi $t6,$zero,0x9
11 sw $t6,0x10($a2)
12 addi $t7,$zero,0x1
13 sw $t7,0x14($a2)
14 addi $t8,$zero,0x4
15 sw $t8,0x18($a2)
16 addi $t9,$zero,0x7
17 sw $t9,0x1C($a2)
18 addi $s4,$zero,0x2
19 sw $s4,0x100($a2)
20 addi $s5,$zero,0xA
21 sw $s5,0x104($a2)
```

Figure 1

Because the values were not showing up in memory when running ‘mdump’ in the simulator, we tried to change the offset for the store word instructions to sequentially store values into the data section of memory starting at 0x10010000, but this was not working with our assembler.



```
 1 addi $a2,$zero,0x0
 2 addi $a3,$zero,0x5
 3 sw $a3,0x10010000($a2)
 4 addi $t3,$zero,0x3
 5 sw $t3,0x10010004($a2)
 6 addi $t4,$zero,0x6
 7 sw $t4,0x10010008($a2)
 8 addi $t5,$zero,0x8
 9 sw $t5,0x1001000C($a2)
10 addi $t6,$zero,0x9
11 sw $t6,0x10010010($a2)
12 addi $t7,$zero,0x1
13 sw $t7,0x10010014($a2)
14 addi $t8,$zero,0x4
15 sw $t8,0x10010018($a2)
16 addi $t9,$zero,0x7
17 sw $t9,0x1001001C($a2)
18 addi $s4,$zero,0x2
19 sw $s4,0x10010020($a2)
20 addi $s5,$zero,0xA
21 sw $s5,0x10010024($a2)
```

Figure 2

:

This third picture shows the rest of the bubble sort algorithm broken up into functions for the sake of readability. This representation was especially helpful when trying to debug the code in and determining where to offset each of the branches and jump instructions (either forward or backward using sign extended shifting and the two's complement respectively).

```
24 MAIN_LOOP:
25 addi t1,t1,0xA //load the size of the array and into t1
26 addi t0,t0,0x1 //load the value '1' in t0 to be used in sub
27 sub a1,t1,t0 //t1-t0->a1 get the count for the current pass(must be < 10)
28 blez $a1,(OFFSET TO MAIN_DONE) //branch to main_done if we are done. (if a1<=0)
29 add $a0,$zero,$t0 //load the array element into a0
30 lui $t2,0 //clear the "did swap" flag
31 jal (OFFSET TO PASS_LOOP) //do a single sort pass, address of this instruction goes in $ra
32 blez $t2,(OFFSET TO MAIN_DONE) //branch on 0,if there are no swaps on current pass, we are done
33 sub $t1,t1,t0 //taking adavantage of t0 being 1. bump down remaining passes
34 j (OFFSET TO MAIN_LOOP) //loop until MAIN_DONE is called
35
36 PASS_LOOP:
37 lw $s1,0x0($a0) //load first element into register s0
38 lw $s2,0x4($a0) //load second element into register s1
39 bgt $s1,$s2,(BRANCH TO SWAP) //if s1 > s2 swap elements
40
41 PASS_NEXT:
42 addi $a0,$a0,0x4 //move pointer to next element in s1 -> a0 = a0+4
43 subi $a1,$a1,0x1 //decrement the number of loops remaining -> a1 = a1-1
44 bgtz $a1,(OFFSET TO PASS_LOOP) //branch if there are still loops remaining (a1>0)
45 jr $ra          //return address
46
47 SWAP:
48 sw $s1,0x4($a0) //store [i+1] into s1 memory address
49 sw $s2,0x0($a0) //store [i] into s2 memory address
50 addi $t2,$t2,0x1 //tell main that we did a swap
51 j (OFFSET OF PASS_NEXT)
52
53 MAIN_DONE:
54 addi $v0,$v0,0xA
55 syscall
```

Figure 3

Here is the process in which the branch and jump instruction offsets were determined. Depending on if the instruction required a backward offset led to the use of two's complement. Both forward and backwards offsets required the use of a right shift by two bits as stated in the R4400 manual.

+72 offset from
Line 28 to MAIN_DONE

0000 0000 0100 1000

Right shift 2

0000 0000 0001 0010 = 0012

+16 offset from
31 to PASS_LOOP

0000 0000 0001 0000

Right shift 2

0000 0000 0000 0100 = 0004

+56 offset from
32 to MAIN_DONE

0000 0000 0011 1000

Right shift 2

0000 0000 0000 1110 = 000E

-36 offset from
Line 34 to MAIN_LOOP

0000 0000 0010 0100

2s comp

1111 1111 1101 1011

Add 1

1111 1111 1101 1100

Shift right 2

1111 1111 1111 0111 = FFF7

+20 offset from
Line 39 to SWAP

0000 0000 0001 0100

Right shift 2

0000 0000 0000 0101 = 0005

-20 offset from
Line 44 to PASS_LOOP

0000 0000 0001 0100

2s comp

1111 1111 1110 1011

Add 1

1111 1111 1110 1100

Shift right 2

1111 1111 1111 1011 = FFFB

-28 offset from
Line 51 to PASS_NEXT

0000 0000 0001 1100

2s comp

1111 1111 1110 0011

Add 1

1111 1111 1110 0100

Shift right 2

1111 1111 1111 1001 = FFF9

Figure 4

This picture shows the final working code for the bubble sort algorithm. Finally the implementation for each offset is placed next to branch and jump instructions allowing for the program to run properly.

```
 1 addi $a2,$zero,0x0
 2 addi $a3,$zero,0x5
 3 sw $a3,0x0($a2)
 4 addi $t3,$zero,0x3
 5 sw $t3,0x4($a2)
 6 addi $t4,$zero,0x6
 7 sw $t4,0x8($a2)
 8 addi $t5,$zero,0x8
 9 sw $t5,0xC($a2)
10 addi $t6,$zero,0x9
11 sw $t6,0x10($a2)
12 addi $t7,$zero,0x1
13 sw $t7,0x14($a2)
14 addi $t8,$zero,0x4
15 sw $t8,0x18($a2)
16 addi $t9,$zero,0x7
17 sw $t9,0x1C($a2)
18 addi $s4,$zero,0x2
19 sw $s4,0x100($a2)
20 addi $s5,$zero,0xA
21 sw $s5,0x104($a2)
22 addi t1,t1,0xA
23 addi t0,t0,0x1
24 sub a1,t1,t0
25 blez $a1,0x12
26 add $a0,$zero,$t0
27 lui $t2,0x0
28 jal 0x4
29 blez $t2,0xE
30 sub $t1,t1,t0
31 j 0xFFF7
32 lw $s1,0x0($a0)
33 lw $s2,0x4($a0)
34 bgt $s1,$s2,0x5
35 addi $a0,$a0,0x4
36 subi $a1,$a1,0x1
37 bgtz $a1,0xFFFB
38 jr $ra
39 sw $s1,0x4($a0)
40 sw $s2,0x0($a0)
41 addi $t2,$t2,0x1
42 j 0xFFF9
43 addi $v0,$v0,0xA
44 syscall
```

Figure 5

For Problem 2, we followed a similar process of breaking the machine code up into functions for the sake of readability and so that it would be easier to test branches and jump offsets. Figure 6 shows the Fibonacci sequence broken up into functions with pseudocode explaining each step.

```
 1 main:
 2 addi $a0,$zero,0x1 //starting value for fib
 3 jal 0x3
 4 addi $v0,$zero,0xA //exit program
 5 syscall
 6
 7 vfib:
 8 addi $t0,$zero,0x1 //set register $t0 to 1
 9 beq $a0,$zero,BRANCH TO FIB0    //go to return 0 if i=0
10 beq $a0,$t0,BRANCH TO FIB1  //go to return 1 if i=1
11 jr JUMP TO FIB
12
13 fib0:
14 lui $v0,0x0 //return 0
15 jr $ra
16
17 fib1:
18 addi $v0,$zero,0x1 //return 1
19 jr $ra
20
21 fib:
22 addi $sp,$sp,-16 //make room for 4 elements in the stack
23 sw $ra,0($sp)   //store return address
24 sw $a0,4($sp)   //store i
25
26 //calculate (fib(n-1))
27 addi $a0,$a0,-1 //decrement i
28 jal JUMP TO VFIB  //recurse for (fib(n-1))
29 sw $v0,8($sp) //save value of (fib(n-1))
30
31 //calculate (fib(n-2))
32 lw $a0,4($sp) //restore value of i from stack
33 addi $s0,$s0,-2 //decrement i twice
34 jal BRANCH TO VFIB //recurse through fib(n-2)
35 sw $v0,12($sp)  //save result in fib(n-2)
36
37 //restore from stack and sum
38 lw $ra,0($sp) //load return address
39 lw $t0,8($sp) //load fib(n-1)
40 lw $t1,12($sp) //load fib(n-2)
41 addi $sp,$sp,16 //free 4 elements on stack
42 add $v0,$t0,$t1 //sum fib(n-1)+fib(n-2)
```

Figure 6

Here is the offset calculations for the branch and jump instructions for Problem 2.

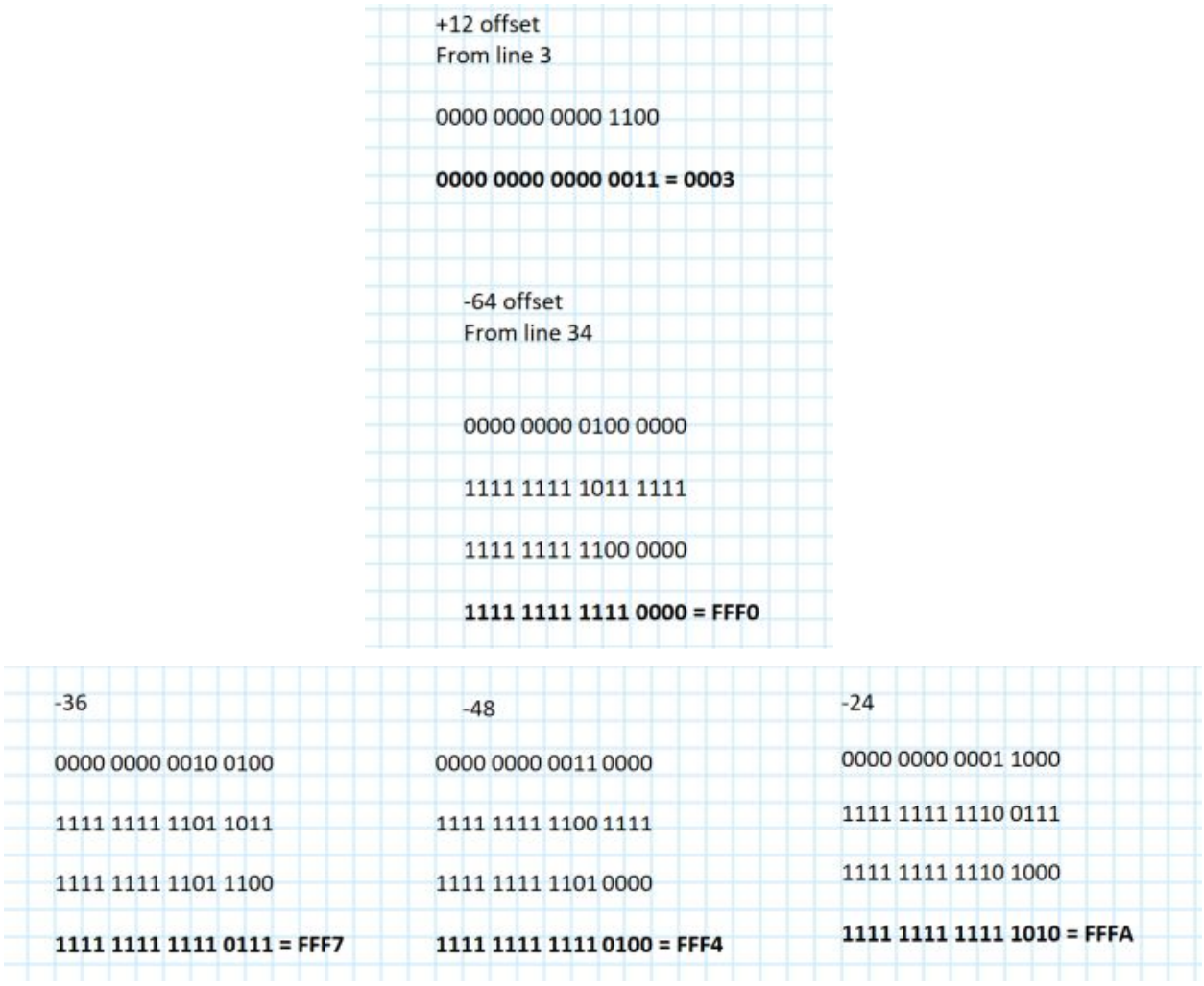
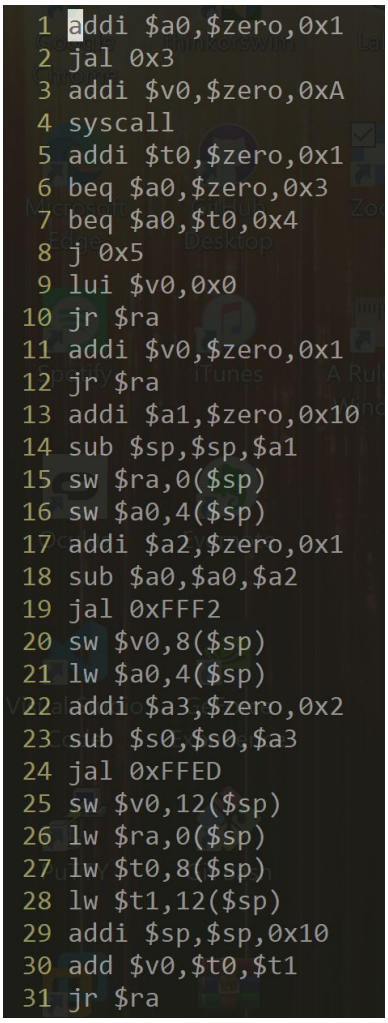


Figure 7

Here is the final code for the fibonacci sequence. All branch and jump offsets are calculated in figure 7 and the code is condensed into this format for the assembler.



```
addi $a0,$zero,0x1
jal 0x3
addi $v0,$zero,0xA
syscall
addi $t0,$zero,0x1
beq $a0,$zero,0x3
beq $a0,$t0,0x4
j 0x5
lui $v0,0x0
jr $ra
addi $v0,$zero,0x1
jr $ra
addi $a1,$zero,0x10
sub $sp,$sp,$a1
sw $ra,0($sp)
sw $a0,4($sp)
addi $a2,$zero,0x1
sub $a0,$a0,$a2
jal 0xFFF2
sw $v0,8($sp)
lw $a0,4($sp)
addi $a3,$zero,0x2
sub $s0,$s0,$a3
jal 0xFFED
sw $v0,12($sp)
lw $ra,0($sp)
lw $t0,8($sp)
lw $t1,12($sp)
addi $sp,$sp,0x10
add $v0,$t0,$t1
jr $ra
```