Computer Organization

Lab 2

Zhengyu Li

Zlr25

Caleb Lee

Calzy8

Nicholas Bouckaert

Nabyq6

Objective:

The goal of this lab was to create an assembler for the MIPS ISA. The assembler will take a given file comprised of MIPS assembly instructions and convert that into MIPS machine code - a 32-bit hexadecimal value. We will use the output machined code to input into out MIPS simulator and compare the given output with the desired effect of the code.

We will also write two instructions sets on to do a bubble sort and the other to Calculate the Fibonacci number of 10. These are both done to rest out assembler.

Distribution:

For this week's distribution most of the coding was done by Zhengyu Li, with the guidance of Caleb and Nick on the basic pseudo code format on how the code should would. Caleb and Nick worked together to solve come up with the MIPS instruction set to do the bubble sort and the calculate the Fibonacci number to 10. While the report was equally distributed among each member of the group.

Milestones:

For this lab, the first milestone we had was figuring out how to translate MIPS instructions to corresponding machine code. We discussed how to convert instructions to hexadecimal code and we decided to convert instructions to binary code according to figures of instructions in user’s manual. Then we transfer our 32 bits binary stream to hexadecimal machine code. The second milestone we had was figuring out converting the bubble sort and Fibonacci instructions. We tested instructions of these two algorithms by using our assembler; meanwhile, we fixed errors of our assembler to make it works well. Some of the hexadecimal machine codes are not correct when we started to test instructions. That was because we had issue to deal with special cases and we figured them out during our debug.

Implementation:

For assembler of the MIPS ISA that we designed, we had a main function which implements the core function of converting between instructions and their 32 bit binary codes. We had other four functions to convert the binary code to hexadecimal format which is the MIPS machine code and we we had one function to print machine codes. The way we implemented the assembler is to split a whole instruction into small instructions and find the corresponding binary codes for those specific small instructions. Then we connected binary codes of those small instructions in correct order and we filled up the stream to 32 bits by adding 0 or the fixed code of the specific instruction. First of all, we established our corresponding arrays at the very beginning of our code. We had an array contains all of the name of instructions and had another array contains the opcode for each instruction in the same order. Same way to the arguments, we had 2 arrays for each argument, one was the name of register and another one was the binary number of register. After that, we defined functions and declared variables and then started our main function. We read from a file through command line and then we had a while loop includes a getline function to read each line in the file until the end of the file. Inside the loop, we split the whole instruction into small slices by space, ‘$’ and ‘.’. The first slice which was the instruction name stored in arg0; the other slices were stored in arg1, arg2, arg3. For some instructions, there were only one or two arguments and the other argument variables were null in this case. Once we split instructions and stored them into argument variables, we had a lot of if statement to find the specific instruction by using strcmp function to check arg0 which was the instruction name. Then we had strcat function to connect the binary stream of each argument of that specific instruction together following the given order in manuel book.

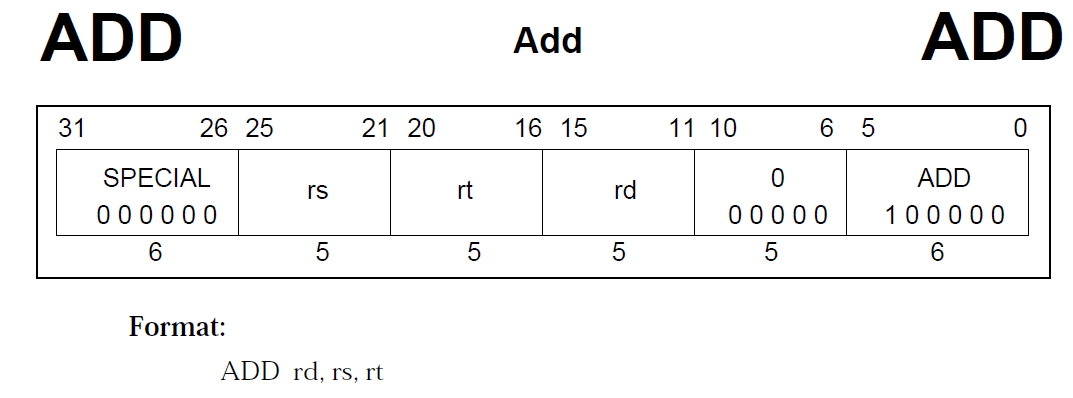


Fig. 1

For example, we knew the format of ADD instruction is “ADD rd, rs, rt” shown above (Fig. 1). The arg0 stored the binary code of instruction which is “000000”, arg1 stored the binary code of rd, arg2 stored the binary code of rs, and arg3 stored the binary code of rt. Then we connected those variables together in opcode, rs, rt and rd order and the expression of our variables are arg0, arg2, arg3, and arg1. Then we added bit 0 to 10 at the end because these 10 bits were fixed.

After these steps above, we would be able to get 32 bits binary stream from the one instruction. And then we converted this stream to hexadecimal machine code by using functions named padBinary, binaryToHex, validate and valueOf. We also had a print function to print the hexadecimal code. At this point, we could get the correct machine code for one instruction and the while loop would make sure that we did the same thing for every instruction till the end. We free pointers at the end to avoid memory leaking and we closed file when we done the assembler.

For both problem 1 and problem 2 we loaded 0x1001 to v0 register by LUI to initialize the registers. For problem 1 we implemented a bubble sort algorithm we added and stored different numbers using ADDIU and SW instruction. We loaded two numbers to registers using LW and we compared them using SUB, BLTZ and BNE instructions. We swapped them using SW instructions if needed. If not, keep going to the next number and compared to the last number within a loop. We called SYSCALL at the end to exit the program. For problem 2, we added 0 to t0 and 0, 1 to t1, t2 registers to initialize the structure of Fibonacci numbers. Then we added the previous two numbers to registers. We had a recursive structure to keep adding numbers by using BNE instruction. We also called SYSCALL at the end to exit the program.

Conclusion:

Overall this lab was fairly straightforward and very informative on how to create a basic assembler that will turn MIPS instructions sets into their hexadecimal format. Then we were give two problems first one was to create a MIPS program that would do a bubble sort and the second was a MIPS program to do the fibonacci number to ten. Once we completed those two instructions sets into our assemble and then took the hexadecimal code and ran it through the MIPS simulation to insure we had accurately accomplished both the hexadecimal code and task in the give problem. We had a few errors within curtain problems however, they were short live and extremely straightforward to fix.

Problem 1: Bubble sort

LUI $v0, 0x1001  
ADDIU $a0, $zero, 0x5  
SW $a0, 0($v0)  
ADDIU $a0, $zero, 0x3  
SW $a0, 4($v0)  
ADDIU $a0, $zero, 0x6  
SW $a0, 8($v0)  
ADDIU $a0, $zero, 0x8  
SW $a0, 0xC($v0)  
ADDIU $a0, $zero, 0x9  
SW $a0, 0x10($v0)  
ADDIU, $a0, $zero, 0x1  
SW $a0, 0x14($v0)  
ADDIU $a0, $zero, 0x4  
SW $a0, 0x18($v0)  
ADDIU $a0, $zero, 0x7  
SW $a0, 0x1C($v0)  
ADDIU, $a0, $zero, 0x2  
SW $a0, 0x20($v0)  
ADDIU $a0, $zero, 0xA  
SW $a0, 0x24($v0)  
  
LUI $t0, 0x1001  
LUI $t3, 0x1001  
ADDIU $a1, $zero, 0xA  
ADDIU $a2, $zero, 0  
SUB $s0, $a1, 0x1  
ADDIU $s1, $zero, 0  
ADDIU $t3, $t3, 0x4  
LW $t1, 0($t0)  
LW $t2, 0($t3)  
SUB $s2, $t1, $t2  
BLTZ $s2, 12  
SW $t2, 0($t0)  
SW $t1, 0($t3)  
ADDIU $s1, $s1, 0x1  
BNE $s0, $s1, -32  
ADDIU $t0, $t0, 0x4  
ADDIU $a2, $a2, 0x1  
ADDIU $s1, $zero, 0  
ADD $t3, $zero, $t0  
SUB $s0, $s0, 0x1  
BNE $s0, $zero, -56  
ADDIU $v0, $zero, 0xA  
Syscall

  
 Fig2: Hexadecimal Code for Bubble sort sending it through our assembler

Problem 2: Fibonacci

LUI $v0, 0x1001  
ADDIU $a0, $zero, 0xA // N = 10  
SW $a0, 0($v0)  
  
ADDIU $t0, $zero, 0  
ADDIU $t1, $zero, 0x1  
ADDIU $t2, $zero, 0  
ADDU $a1, $zero, $a0  
ADDIU $a2, $zero, 0  
ADDIU $t0, $t1, 0  
ADDIU $t1, $t2, 0  
ADDU $t2, $t0, $t1  
ADDIU $a2, $a2, 0x1  
BNE $a1, $a2, -16  
ADDIU $v0, $zero, 0xA  
syscall

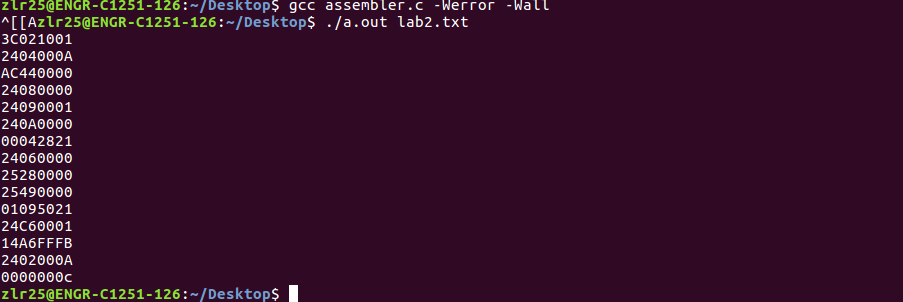


Fig3: Hexadecimal Code for Fibonacci sequence after sending it through our assembler

3C021001

2404000A

AC440000

24080000

24090001

240A00001

00042821

24060000

25280000

25490000

01095021

24C60001

14A6FFFB

2402000A

0000000c

Problem two

3C021001

24040005

AC440000

24040003

AC440000

24040006

AC440000

24040008

AC44000C

24040009

AC440010

24040001

AC440014

24040004

AC440018

24040007

AC440014

24040002

AC440000

2404000A

AC440000

3C081001

3C0B1001

2405000A

24060000

00A18022

24110000

256B0004

8D090000

8D6A0000

012A9022

06400009

AD0A0000

AD690000

26310001

1611FFF7

25080004

24C60001

24110000

00085820

02018022

1600FFF0

2402000A

0000000c

0000000c