ECE 4270: Computer Architecture, Spring 2021

LAB 2: MIPS Assembler

March 4, 2021

Derek Johnson

Chan Kim

Introduction

We have developed an instruction-level simulator for MIPS and test it out from the previous assignment. In this assignment, we will develop an assembler for the MIPS ISA. To do so, we will use the output of our MIPS assembler as an input to MIPS simulator we have developed during the previous lab assignment. Then we write the MIPS assembly program for two algorithms Bubble_Sort, Fibonacci_Number and convert them into MIPS machine code using our MIPS assembler and test them out.

Implementation

Before impellent MIPS assembly program, I wrote a better C code for both algorithms so it will be easier for us to write MIPS assembly program. To implement MIPS assembly program for bubble_sort, we store an array that contains 10 integers into the data segment of the memory before anything else. Once we stored the 10 integers using two instructions lui, and sw, we used two for-loops to sort them in ascending order. During the progress of implementation, we found that we could not use the branch that compare two registers to each other. So, we decided to subtract those two registers and compare the result if it's greater or less than the zero. On the other hand, before we sort the integers in array, we first load the first and second element of array. Comparing to Bubble_sort assembly program we only required to store three registers which are previous, current, and answer. Then we use two branches that if the condition is equal to 0 or not to jump to next instruction. Whenever we calculated the result of Fibonacci number, we used addiu instruction to sum up the answer and store that register to the memory. Then the program will keep looping until it finally met the requirement for branch to exit. The assembler was built in C using many different basic string and integer functions and file streams combined with long if-else ladders. The assembler opens the input file and reads the first string as the instruction. The assembler then compares the string to an if-then ladder to find the instruction. Based on the instruction read, the assembler expects to read a certain number of strings and adds them to a hex number based on what part of the instruction they are related. The assembler has a function for each type of instruction part to determine how much to add based on the strings. The assembler then prints the hex number to the output file and reads the next instruction from the input file. This loops until the end of the input file is reached. Figures 1 and 2 show the MIPS assembly code for the Fibbonacci and bubble sort programs, respectively. Figure 3 and Figure 4 show the implementation of the assembler c-code functions and main.

```
lui $a1, 0xffff
addiu $a1, $a1, 0xffff
lui $zero, 0
addiu $a0, $zero, 0
addiu $a2, $zero, 0
addiu $t0, $zero, 0
addiu $t2, $zero, 0x1
lui $t1, 0x1001
sw $a1, 0 ($t1)
sw $a0, 0x0004 ($t1)
addiu $t3, $zero, 0xA
addiu $t4, $zero, 0
bne $t4, $t0, 0xC
addi $a1, $a1, 0x1
addiu $a0, $a0, 0x1
beq   $t4, $t0, 0x14
add $a2, $a1, $a0
sw  $a2, 0x8, ($t1)
addiu $a1, $a0, 0
addiu $a0, $a2, 0
addiu $t4, $t4, 0x1
bne $t4, $t3, 0x8028`
addiu $v0, $zero, 0xA
syscall
```

Figure 1 - MIPS assembly Fibonacci program

```
lui   $zero, 0
addiu $t0, $zero, 0x5
addiu $t1, $zero, 0x3
addiu $t2, $zero, 0x6
addiu $t3, $zero, 0x8
addiu $t4, $zero, 0x9
addiu $t5, $zero, 0x1
addiu $t6, $zero, 0x4
addiu $t7, $zero, 0x7
addiu $t8, $zero, 0x2
addiu $t9, $zero, 0xa
addiu $a0, $zero, 0
addiu $a2, $zero, 0x9

lui   $a1, 0x1001
sw    $t0, 0 ($a1)
sw    $t1, 0x4 ($a1)
sw    $t2, 0x8 ($a1)
sw    $t3, 0xc ($a1)
sw    $t4, 0x10 ($a1)
sw    $t5, 0x14 ($a1)
sw    $t6, 0x18 ($a1)
sw    $t7, 0x1c ($a1)
sw    $t8, 0x20 ($a1)
sw    $t9, 0x24 ($a1)
sw    $a0, 0x28 ($a1)
sw    $a2, 0x2c ($a1)

addiu $s0, $zero, 0x9
addiu $s1, $zero, 0
addiu $a3, $zero, 0

lw    $t0,  0 ($a1)
lw    $t1,  0x4 ($a1)
sub   $s4, $t0, $t1
bgez  $s4, 0x14
addiu $a0, $t0, 0
addiu $t0, $t1, 0
addiu $t1, $a0, 0
addiu $a3, $a3, 0x1
sub   $s4, $a2, $s1
sub   $s5, $s4, $a3
bgtz  $s5, 0x8028
addiu $t9, $t9, 0x1
bgtz  $s4, 0x8034
addiu $v0, $zero, 0x8
syscall
```

Figure 2 - MIPS assembly bubble sort program

```
int main (int argc, char *argv[]) {

        char *inputfile = argv[1];
        char *outputfile = argv[2];
        printf("%s\n",argv[1]);
        printf("%s\n",argv[2]);

        char instructionID[6];
        int32_t hexInstruction=0;
        char rs[6],rt[6],rd[6];
        char immed[18];
        char target[28];
        uint32_t sa=0;

        FILE* fp=fopen(inputfile,"r");
        if(fp==NULL){
                printf("Error opening input file\n");}

        fscanf(fp,"%s",instructionID);
        printf("Instruction = %s\n",instructionID);

        while (feof(fp)==0){
                hexInstruction=0;

                 if(strcmp(instructionID,"addiu")==0){
                        hexInstruction = hexInstruction + 0x24000000;
                        fscanf(fp, "%s %s %s",rt,rs,immed);
                        addRegister(rs,&hexInstruction,"rs");
                        addRegister(rt,&hexInstruction,"rt");
                        addImmediate(immed,&hexInstruction);}

                else if( strcmp(instructionID,"add")==0 ){
                        hexInstruction = hexInstruction+0x00000020;
                        fscanf(fp,"%s %s %s",rd,rs,rt);
                        addRegister(rs,&hexInstruction,"rs");
                        addRegister(rt,&hexInstruction,"rt");
                        addRegister(rd,&hexInstruction,"rd");}


                else if( strcmp(instructionID,"sll")==0 ){
                        hexInstruction = hexInstruction+0x00000000;
                        fscanf(fp,"%s %s %d",rd,rt,&sa);
                        addRegister(rt,&hexInstruction,"rt");
                        addRegister(rd,&hexInstruction,"rd");
                        hexInstruction=hexInstruction +(sa<<6);
                        }
```

Figure 3 - Assembler main

```
                adder=27;}
        else if(strcmp(reg,"gp")==0){
                adder=28;}
        else if(strcmp(reg,"sp")==0){
                adder=29;}
        else if(strcmp(reg,"fp")==0){
                adder=30;}
        else if(strcmp(reg,"ra")==0){
                adder=31;}
        else{
                printf("Error on register translation %s\n",reg);}

        if(strcmp(regID,"rs")==0){
                adder=adder<<21;}
        else if(strcmp(regID,"rt")==0){
                adder=adder<<16;}
        else if(strcmp(regID,"rd")==0){
                adder=adder<<11;}
        else{
                printf("Error on registerID %s\n",regID);}

        printf("Reg = %s RegID = %s\n", reg, regID);
        printf("Instruction before adder= %x\n",*numberPtr);
        printf("adder= %x\n", adder);
        *numberPtr=*numberPtr+adder;
        printf("Instruction after adder= %x\n",*numberPtr);

        return;
}
void addImmediate(char* immediate,int32_t *numberPtr){
        if (strcmp(immediate,"0")==0){
                *numberPtr = *numberPtr + atoi(immediate);}
        else{
                immediate=strtok(immediate,"0x");
                printf("immediate = %s\n",immediate);
                int32_t immed = (int32_t)strtol(immediate,NULL,16);
                *numberPtr=*numberPtr+immed;}
}
void addTarget(char *target,int32_t *numberPtr){
        if (strcmp(target,"0")==0){
                *numberPtr = *numberPtr + atoi(target);}

        else{
                target=strtok(target,"0x");
                printf("target = %s\n",target);
                int32_t targ = (int32_t)strtol(target,NULL,16);
                *numberPtr= *numberPtr + targ;}
```

Figure 4 - Assembler Functions

Result

Our results for this lab seemed good. Both of our assembly code programs ran through the assembler. Both output files are the same length in non-white space lines as our input files, which means that the assembler seems to be working correctly. The hex numbers appear to be accurate based on the example outputs which we were given in the lab document. Figure 5 shows the output of the Fibonacci program in hexadecimal instructions. Figure 6 shows the output of the bubble sort program in hexadecimal instructions. We could have taken our results further and tested them by running the output instructions through the lab from last week, but we didn't prioritize this in our time enough.

24 lines (24 sloc) | 216 Bytes

```
3c05ffff
24a5ffff
3c000000
24040000
24060000
24080000
240a0001
3c090001
a9250000
a9240004
240b000a
240c0000
1588000c
20a50001
24840001
11880014
00a43020
a9260008
24850000
24c40000
258c0001
158b0008
2402000a
0000000c
```

Figure 5 - Fibonacci hex output

44 lines (44 sloc) | 396 Bytes

```
3c000000
24080005
24090003
240a0006
240b0008
240c0009
240d0001
240e0004
240f0007
24180002
2419000a
24040000
24060009
3c050001
a8a80000
a8a90004
a8aa0008
a8ab000c
a8ac0001
a8ad0014
a8ae0018
a8af001c
a8b80002
a8b90024
a8a40028
a8a6002c
24100009
24110000
24070000
8ca80000
8ca90004
0109a022
06810014
25040000
25280000
24890000
24e70001
00d1a022
0287a822
1ea00008
27390001
1e800008
24020008
0000000c
```

Figure 6 - Bubble sort hex output

Conclusion

Our results were much better than last labs it seems. Our assembler ran completely through both programs and produced an output that seems accurate. Further testing by running our output into last week's lab could have helped us confirm our results. This lab was very challenging. Our workload was more balanced this lab. Derek wrote the entirety of the assembler code, revised the MIPS assembly code to work in the assembler, checked the logic of the assembly code. Chan wrote the two assembly programs and helped write some of the lab report. Overall, the lab was challenging in every step taken. However, the challenge helped us to understand better about this lab. After implementing and writing two algorithms in MIPS assembly program and converting them into MIPS machine code, we enhance understanding of instruction words, MIPS, storing registers into the data segment of the memory, loading right registers from the memory, and using the right branches for if-else statements and for-loops.