**CSC 250 MIPS Assembly Exercises**

Answer questions. Show your work.

1. **(Derived from 2.14)** What is the assembly language instruction for the following binary pattern? What instruction format does it use? Explain each field of the instruction given the instruction format.

**0000 0010 0001 0000 0100 1000 0010 01002**

**(op): 000000, This is a R-format instruction.**

**(rs): 10000, the first register which means $s0**

**(rt): 10000, the second register which means $s0**

**(rd): 01001, the register destination which is $t1**

**(shamt): 00000, the shift amount which is 0**

**(funct): 100100, the function code which means and here.**

1. **(Derived from 2.15)** Given the following instruction:

**lw $s1, 36($t2)**

What instruction format does this instruction use? **\_\_\_\_\_I-format\_\_\_ \_\_\_\_\_\_\_\_\_\_**

What is the opcode in binary and hex? **\_binary: \_1000112\_\_hex:\_0x23\_\_\_**

What is the register numbers for: **$s1 \_\_\_17\_\_\_\_\_ $t2 \_\_\_\_10\_\_\_\_\_**

What decimal value is stored in the 16-bit immediate field? **\_\_\_\_\_36\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

What is the hex value for the 16-bit pattern in the immediate field? **\_\_\_\_\_\_0x24\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

What is the largest positive value, in decimal, that the immediate field could hold?**\_\_ 32767\_\_\_\_**

What is the most negative value, in decimal, that the immediate field could hold? **\_\_\_\_-32768\_\_\_\_**

1. What is the difference between the **j** and the **jr** instructions? In particular: how is the address formed differently (briefly); compare the range of jump addresses.

**j is used to perform an unconditional jump to specific memory address based on current PC. The jump targets could be anywhere in text segment. The range of jump address is address’s length plus 2 bits which is 28 bits. It has range of 2^28**

**jr is a R-format opcode which performs an unconditional jump to an address specified by a register. jr can only jump to the address stored in the register. The range of jump addresses is from 2^31-1 to 2^31 but these addresses need to be first stored in register number.**

1. For a branch instruction, how many instructions in either direction (forward or backward in memory from the current PC) can be branched over?

Target address = (PC+4) + (“address” \* 4)

Forward:

(4) + (2^15-1 \* 4) = 131075

Backward:

(4) + (-2^15 \*4) = -131068

You can go 131075 instruction in forward direction and 131068 instruction for backward direction.

1. 2.25.16 – Parts (a) and (b) are given. For part (c), discuss some advantage(s) and disadvantage(s) of the change suggested.

Assume that we would like to expand the MIPS register file to 128 registers and expand the instruction set to contain four times as many instructions.

My answers to a and b do not currently agree with the text, but I like what I have below.

1. How would this affect the size of each of the bit fields in the R-type instructions?

The opcode would expand from 6 bits to 8. The rs1, rs2, and rd fields would increase from 5 bits to 7 bits. Alternatively, you could expand the function field by 2 bits.

1. How would this affect the size of each of the bit fields in the I-type instructions?

The opcode would expand from 6 bits to 8. The rs1 and rd fields would increase from 5 bits to 7 bits. This change does not affect the imm field per se, but it might force the ISA designer to consider shortening the immediate field to avoid an increase in overall instruction size.

1. How could each of the two proposed changes decrease the size of an MIPS assembly program? On the other hand, how could the proposed change increase the size of an MIPS assembly program?

Advantage:

Because we have increased the size of opcode, we now can specify some new operation codes which can do more tasks without using combination of previous opcodes.

Because we also increase the size of register number, we can use more register number to store addresses. It will help us coding because we can use more temporary variables.

Disadvantages:

One disadvantage is that opcode and register number now take more time to encode and decode. CPU needs more time and power to perform different operations and find the address stored in many register numbers.

Larger instruction sets also means that the assembly code file will be more complicated with more opcodes and register numbers. It will be hard for programmer to understand assembly and make optimization given so many opcodes.

1. 2.25.4 Complete the table below from the problem in the text but also fill in the remaining C statements in the last column. The comment in the last row should be the C statement that this code implements. Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively.

|  |  |  |
| --- | --- | --- |
| sll | $t0, $s0, 2 | # $t0 = f \* 4 |
| add | $t0, $s6, $t0 | # $t0 = &A[f] |
| sll | $t1, $s1, 2 | # $t1 = g \* 4 |
| add | $t1, $s7, $t1 | # $t1 = &B[g] |
| lw | $s0, 0($t0) | # f = A[f] |
| addi | $t2, $t0, 4 | # $t2 = &A[f+1] |
| lw | $t0, 0($t2) | # $t0 = A[f+1] |
| add | $t0, $t0, $s0 | # $t0 = A[f] + A[f+1] |
| sw | $t0, 0($t1) | # B[g] = A[f] + A[f+1] |

C statements: B[g] = A[f] + A[f+1]

1. **(Optional/Advanced)** *For this problem the MARS simulator is required and only suggested for highly interested students who have the time. Do not do this if you have not completed everything else. We may go over it in class.* Consider the MIPS assembler program below.

**.data**

**MEM1: .word 0xAAAAAAAA**

**.text**

**lw $t0, MEM1**

**sll $t2, $t0, 4**

**andi $t2, $t2, -1**

**li $v0,10**

**syscall**

Put this program in the MARS simulator and assemble it. Verify that your assembled code matches the screenshot below.

Graphical user interface, application, table

Description automatically generated

Run the program and enter the final hex values for:

**$t0: \_\_\_\_0xaaaaaaaa\_\_\_\_\_\_\_\_\_\_**

**$t2: \_\_\_\_0xaaaaaaa0 \_\_\_\_\_\_**

Explain what each line is doing by copying each line and putting the explanation with it. Your explanation should lead to the value of **$t2:** Be sure to explain the use of **$1**, the **$at** register. The comments may take some work and Google searching. You do not need to explain the last two instructions, **li** and **syscall**. Here is a start:

**lui $1, 0x00001001**

**It means it first load the upper 16-bits of MEM1 and store it in $1($at)**

**lw $8, 0x00000000($1)**

**It means that lw load the data with address stored in $1. The offset means it loads the first data in that array and store it in register $8.**

**sll $10, $8, 0x00000004**

**This means that $8 does left shift with amount of 4, then stored the value into $10($t2)**

**lui $1, 0xffffffff**

**This means it load the upper 16-bits of -1 and store it in $1**

**ori $1, $1, 0x0000ffff**

**means you compute the bitwise or with $1 and lower 16-bits of -1, 0x0000ffff. $1 | 0x0000ffff and store the result back to $1**

**and $10, $10, $1,**

**means performing bitwise AND with the data stored in $10 and $1 which should be 0xaaaaaaaa \* 16 & 0xffffffff.**

…

The solution was discussed in class. Note that lw first needs to resolve the memory address of MEM1 for which it makes use of the $at ($1) register. Then it does the LW using the address in $at. Similarly for andi, it must make a 32-bit constant for -1. To get this all done, it needs the lui and ori to put the constant into $at then it does the and. I.e., it needed three instructions.

Advanced – check out this program – see below

**.data**

**MEM1: .word 0xAAAAAAAA**

**.text**

**lw $t0, MEM1**

**sll $t2, $t0, 4**

**andi $t2, $t2, 0x0000FFFF**

**li $v0,10**

**syscall**

Try this in MARS. You will see the andi only makes one instruction, the normal andi with a 16-bit constant. That is because the assembler is smart enough to figure out you only need 16 bits and the upper 16 are zeros. All of this is interesting from a couple of perspective. One is how the assembler interprets instructions. Second is how the instructions actually behave in the hardware. Compiler writers must be intimately familiar with all of this.