Jordan Wu

U0900517

HOMEWORK ASSIGNMENT #3

1. **Do problem 2.15 from the end of Chapter 2. Encoding an instruction in hexadecimal takes several steps - show your work for these steps. (Note: Type refers to instruction type, such as R-Type, I-Type, etc.)**

2.15

Provide the type and hexadecimal representation of following instruction: sw $t1, 32($t2)

Looking at the green card in the book called "MIPS Reference Data", I can see that sw stands for store word. This type of instruction format is called I-type (for immediate) and is used by the immediate and data transfer instructions. In this case this is used as a data transfer instruction where the word from register goes to memory.

The instruction format for I-type is

opcode	rs	rt	Constant or address
6 bits	5 bits	5 bits	16 bits

(A good example in the book is page 84)

Use this to determine rs and rt -> M[R[rs]+SignExtImm] = R[rt]

The opcode for sw is 43_{ten} (from the example and green card)

Looking in Appendix A on page A-24, there is a figure with the MIPS registers and usage convention. This tell me that $t1 has the value 9_{ten} and $t2 has the value 10_{ten} (rs is $t2 and rt is $t1)

The Offset is 32_{ten}

Now we have to convert these values into a hexadecimal representation.

opcode	rs	rt	Constant or address
43_{ten}	10_{ten}	9_{ten}	32_{ten}
101011	01010	01001	0000000000100000

Now convert this to hexadecimal representation

1010-1101-0100-1001-0000-0000-0010-0000= $AD490020_{hex}$

Now double checking my results in MARS

The instruction code for this in MARS is 0xad490020

This implies that I did it correctly

2. **Repeat question #1 (based on problem 2.15) on this instruction:**

   addi $sp, $sp, -32

addi stands for add Immediate and is in I-type format.

opcode	rs	rt	Constant or address
6 bits	5 bits	5 bits	16 bits

Use this to determine rs and rt -> R[rt] = R[rs] + SignExtImm

The opcode for addi is 8_{ten}

$sp stands for stack pointer and has the value 29_{ten} (rs and rt are the same registers)

The constant is -32_{ten} is a signed number

opcode	rs	rt	Constant or address
8_{ten}	29_{ten}	29_{ten}	-32_{ten}
001000	11101	11101	1111111111100000

Now convert this to hexadecimal representation

0010-0011-1011-1101-1111-1111-1110-0000 = $23BDFFE0_{hex}$

Now double checking my results in MARS

The instruction code for this in MARS is 0x23bdffe0

This implies that I did it correctly

3. **Repeat question #1 (based on problem 2.15) on this instruction:**

   or $v0, $s3, $t4

This or is a R-type instruction which has a different format than I-type

opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Use this to determine rs, rd, and rt -> R[rd] = R[rs]|R[rt]

The opcode for or is 0_{ten} with funct value 37_{ten}

rs value for $s3 is 19_{ten}

rt value for $t4 is 12_{ten}

rd value for $v0 is 2_{ten}

There is no shift so I assume that shamt is 0_{ten}

opcode	rs	rt	rd	shamt	funct
0_{ten}	19_{ten}	12_{ten}	2_{ten}	0_{ten}	37_{ten}
000000	10011	01100	00010	00000	100101

Now convert this to hexadecimal representation

0000-0010-0110-1100-0001-0000-0010-0101 = $026C1025_{hex}$

Now double checking my results in MARS

The instruction code for this in MARS is 0x026c1025

This implies that I did it correctly

4. **Do problem 2.14 from the end of Chapter 2. Decoding an instruction takes several steps - show your work for these steps. Also, don't just write the opcode - write a complete assembly language instruction using the correct names for each register.**

2.14

Provide the type and assembly language instruction for the following binary value:

$0000\ 0010\ 0001\ 0000\ 1000\ 0000\ 0010\ 0000_{two}$

We have to figure what type of format is this instruction, looking at the first 6 MSB we can see that this is an R-type instruction (all R-type instruction has 000000 for their opcode). We just have to put this into the R-type format to decode this instruction

opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
000000	10000	10000	10000	00000	100000

We know that opcode is 000000 and the funct is 100000. The green card states that this is add

Register 10000 is used for rd, rs, and rt which is 16 which is $s0

Now put this together

```
add   $s0, $s0, $s0
```

To check if we decode this instruction correctly, we would have to convert this into hexadecimal representation

$0000\text{-}0010\text{-}0001\text{-}0000\text{-}1000\text{-}0000\text{-}0010\text{-}0000 = 02108020_{hex}$

Now double checking my results in MARS

The instruction code for this in MARS is 0x02108020

This implies that I did it correctly

5. **Repeat question #4 (based on problem 2.14) on this encoded instruction:**

   0x8C90FFFC

First thing is to convert this into a 32-bit binary representation

0x8C90FFFC = 1000-1100-1001-0000-1111-1111-1111-1100

We have to figure what type of format is this instruction, looking at the first 6 MSB 100011, this is load word lw. This has an I-type format which is used as a data transfer instruction. We just have to put this into the I-type format to decode this instruction

opcode	rs	rt	Constant or address
6 bits	5 bits	5 bits	16 bits
100011	00100	10000	1111111111111100

rs has the value 4 which is $a0

rt has the value 16 which is $s0

the address is -4 (signed number)

Now put this together

```
lw    $s0, -4($a0)
```

Now double checking my results in MARS

The instruction code for this in MARS is 0x8c90fffc

This implies that I did it correctly

6. **Repeat question #4 (based on problem 2.14) on this encoded instruction:**

   0x15000003
   Note: You would normally need a label for this instruction - just specify the instruction offset instead.

First thing is to convert this into a 32-bit binary representation

0x15000003 = 0001-0101-0000-0000-0000-0000-0000-0011

We have to figure what type of format is this instruction, looking at the first 6 MSB 000101, which is branch if not equal bne. This is an I-type format, we just have to put this into the I-type format to decode this instruction

opcode	rs	rt	Constant or address
6 bits	5 bits	5 bits	16 bits
000101	01000	00000	0000000000000011

We know that this is bne

rs has the value 8 which is $t0

rt has the value 0 which is $zero

the constant is 3

Now put this together (Using the offset as the label)

```
bne   $t0, $zero, label
```

Now double checking my results in MARS

The instruction code for this in MARS is 0x15000000 (offset by 3) which give us 0x15000003

This implies that I did it correctly

7. **Do problem 2.23 from the end of Chapter 2. Show which instructions execute and show the computation that each instruction does. For example only: if $a1 contained 20, addi $s0, $a1, 10 would compute $s0 = 20 + 10.**

2.23

Assume $t0 holds the value 0x00101000. What is the value of $t2 after the following instructions?

```
        slt   $t2, $0, $t0
        bne   $t2, $0, ELSE
        j     DONE
ELSE:   addi  $t2, $t2, 2
DONE:
```

First we need to know what the hexadecimal value is in this temporary register

0x00101000 = 0000-0000-0001-0000-0001-0000-0000-0000

$= 1 \cdot 2^{12} + 1 \cdot 2^{20}$

= 1052672

The first instruction is called set on less than, or slt

```
slt   $t2, $0, $t0     #     ( $0 < $t0 ) ? 1:0
```

This mean that if $0 < $t0 is true than $t2 will have the value 1, else it will be false and have the value 0

For this case $t2 will have the value 1 because it is true => $t2 = 1

The second instruction is branch if not equal, or bne

```
bne   $t2, $0, ELSE    # go to ELSE if $t2 ≠ $0
```

Since $t2 ≠ 0 it will go to ELSE

The Else will run the following

```
addi $t2, $t2, 2      # $t2 = $t2 + 2
```

This will add/increment by 2 to the current $t2

The final value for $t2 is 3

Note: the jump to DONE was not executed if the value in $t0 is greater than 0

8. **Do problem 2.26 (all three parts) from the end of Chapter 2. (Ignore the typos in the book - there are not 'loops', only a 'loop'.)**

2.26

Consider the following MIPS loop:

```
LOOP:   slt  $t2, $0, $t1
        beq  $t2, $0, DONE
        subi $t1, $t1, 1
        addi $s2, $s2, 2
        j    LOOP
DONE:
```

2.26.1

Assume that the register $t1 is initialized to the value 10. What is the value in register $s2 assuming $s2 is initially zero?

If $t1 has the value 10 then $t2 will have the value 1 since $0 < 10

=> $t2 = 1

The next sequence is the beq where $t2 ≠ 0, so this instruction is skipped

So it will run subi where $t1 will equal 10 – 1 => $t1 = 9

Then the addi will run which we will get $s2 = 0 + 2 => $s2 = 2

Lastly it will rerun the loop

This looks like a while loop or for loop

```
int s = 0;
int i = 10;
while(i > 0)
{
    i--;
    s = s + 2;
}
```

Or

```
int s = 0;
for(int i = 10; i > 0; i--)
`   {
        s = s + 2;
    }
```

So this will run it 10 times => $s2 = 20 before the program is done

2.26.2

For each of the loops above, write the equivalent C code routine. Assume that the registers $s1, $s2, $t1, and $t2 are integers A, B, i, and temp, respectively

There is only one loop

Using the registers $s1, $s2, $t1, and $t2 are integers A, B, i, and temp, respectively. I can get the following C code below (I will use a while loop, think a for loop will work too)

```
i = unknown value (don't know it yet)
B = 0;
while ( i > 0 )
{
    i = i - 1;
    B = B + 2;
}
```

2.26.3

For the loops written in MIPS assembly above, assume that the register $t1 is initialized to the value N. How many MIPS instructions are executed?

Now that $t1 is some value N. This program will keep looping until 0 < N is not satisfied. Each time it loops it is decremented by 1, this program will run N times. We are looking for the number of MIPS instruction that are executed.

It will run the following instruction N times

```
slt   $t2, $0, $t1
beq   $t2, $0, DONE
```

```
subi  $t1, $t1, 1
addi  $s2, $s2, 2
j     LOOP
```

On the last run it will run the following

```
slt   $t2, $0, $t1
beq   $t2, $0, DONE
```

So the number of instructions required is 5·N + 2

9. **Do problem 2.21 from the end of Chapter 2. The problem is asking you to rewrite a pseudo instruction using only MIPS instructions. Explain with an example how your solution implements the not instruction.**

2.21

Provide a minimal set of MIPS instructions that may be used to implement the following pseudo instruction.

```
not   $t1, $t2   // bit-wise invert
```

We have to find a way to write a pseudo instruction using only MIPS instructions. We discussed in class that this is possible by using MIPS logical operations. In the textbook it states that the designers of MIPS decided to include the instruction NOR (NOT OR) instead of NOT. If one operand is zero then it is equivalent to NOT: A NOR 0 = NOT (A OR O) = NOT(A). Assuming that following code does this, $t1 = NOT($t2).

Using the MIPS nor instruction -> R[rd] = ~(R[rs]|R[rt])

```
nor   $t1, $t2, $zero  # $t1 = ~( $t2|$zero) = ~($t2)
```

Example:

Say $t2 has a value (random number) 1564879857

The binary representation is 01011101010001100010101111110001

$zero has the binary representation 00000000000000000000000000000000

If we use the logical operation OR

```
$t2 | $zero = $t2
```

Since the OR places a 1 in the results if either operand bit is a 1, the $zero has no 1's => we get back 01011101010001100010101111110001 the 32-bit value in $t2

Now we will NOT this value, which is the 32-bits inverted

10100010101110011101010000001110 => conclude that $t1 = ~($t2) if $zero is used

10. **Do problem 2.25 (both parts) from the end of Chapter 2. (The functionality of the rpt instruction is shown immediately following the instruction.) Again, do not use pseudo instructions in your answer. You may use $at as a temporary register if needed.**

2.25

The following instruction is not included in the MIPS instruction set:

rpt $t2, loop # if (R[rs]>0) R[rs]=R[rs]-1, PC=PC+4+BranchAddr

2.25.1

If this instruction were to be implemented in the MIPS instruction set, what is the most appropriate instruction format?

Looking at rpt instruction it need the R[rs] register and a BranchdchAddr which is an address. Looking at the different type of formats, the most appropriate instruction format for this will be I-Type since it involves rs register and constant/address.

2.25.2

What is the shortest sequence of MIPS instructions that performs the same operation?

We can use $at as the temporary register

Before writing the MIPS instruction, we need to understand what is going on first. The instruction for rpt is that if $t2 is greater than zero than $t2 = $t2 -1, then we jump to some instruction address.

Shortest Sequence of MIPS instruction:

```
loop:   slt  $at, $zero, $t2      # ($zero < $t2) 1:0 (true/false)
        addi $t2, $t2, -1         #$t2 = $t2 -1
        bne  $at, $zero, loop     # go to loop if $at ≠ 0
```

11. Download and save the following assembly language program:

    Hex converter.s

    Use the Mars MIPS Simulator to finish the program, and then test the program for accuracy. You may download Mars to any computer that has Java on it, or you may use Mars in the CADE lab.

    The program is mostly complete - it prompts the user for an integer and inputs the integer. It should output the hexadecimal representation for the integer, but I have deleted these statements from the program. Add the necessary statements so that the program prints out the user's integer in hexadecimal.

    Do not rewrite the existing code! Also, do not change which registers are used in the existing code. Just finish the program as-is.

    Hints: I deleted the instructions from 12 lines, but I left the comments behind. Your solution may take a few more lines. I also followed the pattern similar to the binary converter that I started in class. Also, I extensively use a move pseudoinstruction. The move pseudoinstruction simply copies a value from one register to another. (Rhetorical: Can you figure out how you could do this without a move instruction?)

    Also - make sure to practice a bit of MIPS prior to attempting to solve this problem!

This was pretty simple problem to solve, since there was the comments that basically told us what to do. The only problem I had was that the program had already define the DigitOut label that will output the ASCII character. This caused me some problem since I used a different temporary register (the program used $t0) but, was a quick fix. Other than that this problem was fun and easy if you understand what is going on.