Rodrigo Becerril Ferreyra

Student ID 017584071

CECS 440 Section 02

04 October 2021

Exercise 1

Note: in this document, I use underscores in binary, octal, or hexadecimal numbers. These underscores are for visual purposes only, like the comma in decimal numbers.

Problem 2.1

```
add $t0, $s2, -5
add $s0, $s1, $t0
```

Problem 2.4

```
B[g] = A[f] + (f + 1);
```

Problem 2.10

MIPS registers cannot hold the values presented in the book (that requires 128 bits while MIPS registers only hold 32 bits), so I will assume that `$s0` holds a value of `0x8000_0000` and `$s1` holds a value of `0xD000_0000`, respectively.

1. The contents of `$t0` are `0x5000_0000`.
2. An overflow has occurred; the answer should be `0x1_5000_0000,` but there are too few bits to represent this result.
3. The contents of `$t0` are `0xB000_0000`.
4. There is no overflow in this result.
5. The contents are `0xD000_0000`.
6. Apart from the overflow in the first instruction, this pair of instructions give the correct result in this case.

Problem 2.13

The code for the instruction given is `101011_01001_01010_0000000000100000b` (or, in decimal, `43_9_10_32`), which is an I-type command.

Problem 2.14

1. The instruction is a `sub` instruction, with a format of R-type.
2. The instruction is `sub $v1, $v1, $v0`.
3. The binary representation is `000000_00011_00010_00011_00000_100010b`.

Problem 2.15

1. The instruction is a `sw` instruction, with a format of I-type.

2. The instruction is `sw $at, 4($v0)`.

3. The binary representation is `100011_00001_00010_0000000000000100b`.

Problem 2.18

The "tricky" part of this problem is the fact that you need to clear bits 31 through 26 by using a mask. This mask takes two instructions to create and one to apply, as opposed to only one instruction if the mask could be applied to lower bits.

```
sll $t2, $t0, 15     # shift all data to MSB position
lui $t3, 0x3FF       # create mask
ori $t3, $t3, 0xFFFF # create mask
and $t1, $t1, $t3    # apply mask
or $t1, $t1, $t2     # replace masked bits
```

Problem 2.26

Below are two solutions to Problem 2.25. The first is the most accurate to the code. The second is only valid if both loops are guaranteed to run at least once.

```
add $t0, $zero, $zero # i = 0

iloop:
    slt $t5, $t0, $s0     # check for iloop
    beq $t5, $zero, exit  # exit if (i < a) is false

    add $t1, $zero, $zero # j = 0

    jloop:
        slt $t5, $t1, $s1     # check for jloop
        beq $t5, $zero, jloopexit # exit only inner loop if (j < b)
is false
        sll $t2, $t1, 4       # t2 = (j * 4) * 4
        add $t3, $s2, $t2     # t3 holds the adjusted address of D

        add $t4, $t0, $t1     # t4 = i + j
        sw $t4, 0($t3)        # D[j * 4] = i + j

        addi $t1, $t1, 1      # j++
        j jloop

    jloopexit:
        addi $t0, $t0, 1 # i++
        j iloop

exit:
```

```
add $t0, $zero, $zero # i = 0

iloop:
    add $t1, $zero, $zero # j = 0

    jloop:
        sll $t2, $t1, 4       # t2 = (j * 4) * 4
        add $t3, $s2, $t2     # t3 holds the adjusted address of D

        add $t4, $t0, $t1     # t4 = i + j
        sw $t4, 0($t3)        # D[j * 4] = i + j

        addi $t1, $t1, 1      # j++
        slt $t5, $t1, $s1     # check for jloop
        bne $t5, $zero, jloop

    addi $t0, $t0, 1  # i++
    slt $t5, $t0, $s0 # check for iloop
    bne $t5, $zero, iloop
```

Solution 1 has 14 instructions, while Solution 2 has 12 instructions. Perhaps if the compiler could not know if both loops would run at least once (for example, if `a` and `b` were user-inputted) it would use the first Solution. However, if `a` and `b` were hard-coded, the compiler would attempt to optimize and use the 12-instruction version instead.

Problem 2.39

The original program requires 900 million instructions to complete. The new program requires 775 million instructions to complete. This is a 13.889% decrease in total number of instructions. This offsets the 10% increase in time that the slower clock cycles add. This is a good design choice.

Problem 2.41

Using scaled offset addressing, I would not need to shift the index by 2 every time I would want to use it. Seeing as how there are two arrays that need indexing, at least two instructions would be saved, and it would make the code a lot more readable.

Problem 2.42

Using scaled offset addressing, just like last time, no shifting of the index is necessary; the hardware does that for you, so you don’t have to worry about it. In this example, there are three arrays to access, so it would save a lot of time and instructions if I didn’t have to shift the index to the right two times. It would be easier on the eyes and on the processor.