Assignment 2

Alex Clemmer
Student number: u0458675

Problem 1:

2.10.1: Opcodes are the first 6 bits of an instruction. $Opcode_{(a)}$ is `10 1011`$_{two}$, which encodes `sw`. $Opcode_{(b)}$ is `10 0011`$_{two}$, which encodes `lw`.

2.10.2: Both $Instruction_{(a)}$ (`sw`) and $Instruction_{(b)}$ (`lw`) are I-type.

2.10.3: Hex conversions should be done from the least significant bit up towards the most significant bit: $Instruction_{(a)}$ becomes `0xAD100002`, and $Instruction_{(b)}$ becomes `0xFFFFB353`.

2.10.4: Let's first break each instruction into fields, starting with $Instruction_{(a)}$:

opcode	rs	rt	rd	shamt	funct
add	\$t1	\$t2	\$zero		
00 0000	0 1000	0 0000	0 1000	0 0000	10 0000

Split into bytes and converted into hex, we get `0x01004020`. The same follows for $Instruction_{(b)}$:

opcode	rs	rt	constant or address
lw	\$t1	\$s3	4
10 0011	1 0011	0 1001	0000 0000 0000 0100

In hex, that's `0x8E690004`.

2.10.5: $Instruction_{(a)}$ is R-type, while $Instruction_{(b)}$ is I-type.

2.10.6: According to the data we generated in the last few questions, $Instruction_{(a)}$ (which is R-type) has an opcode is `0x00`, an rs of `0x08`, an rt of `0x00`, an rd of `0x08`, a shamt of `0x00` and a funct of `0x20`

$Instruction_{(b)}$ (I-type) has an opcode of `0x23`, an rs of `0x13`, an rt of `0x09`, and an immediate field of `0x0004`

Problem 2:

2.11.1: Conversion from hex to binary is pretty simple. (a)'s `0xAE0BFFFC` becomes `1010 1110 0000 1011 1111 1111 1111 1100`$_{two}$. (b)'s `0x8D08FFC0` becomes `1000 1101 1001 1000 1111 1111 1100 1001`$_{two}$.

2.11.2: (a) converted to a signed decimal is `-1374945284` and (b) is (unsigned) `2375614409`.

2.11.3: Well, technically, (a) represents `sw $s0, 65532($t3)`, but that immediate is huge and would get mitigated into multiple commands, such as the following: `ori $1, $0, 0xfffc; addu $1, $1, $11; sw $16, 0x0000($1)`. [Ed. note: I know there are no semicolons in MIPS; I use them here for conciseness, not because they actually exist.]

(b), is in a similar situation: `lw $t4, 65481($t8)` is the official translation, but this would end up getting mitigated for sure to something like `ori $1, $0, 0xffc9; addu $1, $1, $24; lw $12, 0x0000($1)`.

Problem 3:

2.17.1: Complicated high-level procedures are characteristics of high-level languages. Pursuant to von Neumann's "simplicity" of "equipment", MIPS (and almost every other architecture, ever) doesn't include instructions like `abs`. Two or more smaller, simpler instructions make not only for faster execution, it leasts to much less complicated architecture design.

The reason `sgt` doesn't exist is because it's redundant. In high-level code, it's more expressive and convenient, but in assembly it's neither necessary nor important. What's important is simple architecture and swift execution. So it's excluded.

2.17.2: Both `abs` and `sgt` would be R-type.

2.17.2: My implementation of `sgt` might be something like this:

```
        slt $t1,$t2,$t3
        beq $t1, $zero, SetTrue
        li $t1, 0
        j AfterSetTrue
SetTrue:
        li $t1, 1
AfterSetTrue:
        [...]
```

On the other hand, `abs` is a bit less bookkeep-y:

```
        slt $t2, $t3, $zero
        bne $t2, $zero, IsPositive
        nor $t2, $t3, $zero
        addi $t2, $t2, 1
IsPositive:
        [...]
```

Problem 4:

```
# Author: Alex Clemmer
# Date:   9/4/10
#
# A MIPS program that will convert an integer into a hexadecimal number.
# Students must add the missing lines of code.
```

```
        .data

Prompt:
        .asciiz "Enter an integer: "

Output1:
        .asciiz "The decimal number "

Output2:
        .asciiz " is 0x"

Output3:
        .asciiz " in hexadecimal."


        .text

# Get input from user.  First, issue a prompt using the
# system call that will print out a string of characters.

        la $a0, Prompt  # Put the address of the string in $a0
        li $v0, 4
        syscall

# Next, make the system call that will wait for the user to input
# an integer.

        li $v0, 5  # Code for input integer
        syscall

# Integer input comes back in $v0
# Save the inputted integer in a saved register - important!
# It cannot stay in $v0 as we need to reuse $v0.

        move $s0, $v0

# Next, begin to output the result message.  This is done in several
# steps, including outputting strings and the original integer.

# Output first string.

        la $a0, Output1
        li $v0, 4
        syscall

# Output original integer

        move $a0, $s0  # Remember, $s0 contains the input number.
        li $v0, 1
        syscall

# Output second string.

        la $a0, Output2
```

```
        li $v0, 4
        syscall

# Output the hexadecimal number.  (This is done by isolating four bits
# at a time, adding them to the appropriate ASCII codes, and outputting
# the character.  It is important that the digits are output in
# most-to-least significant bit order.


                  # Set up a loop counter
        li $s1, 8
Loop:
                  # Roll the bits left by four bits - wraps highest bits to lowest bits (where we ne
        rol $s0, $s0, 4 # rol
                  # Mask off low bits (logical AND with 000...01111)
        andi $t0, $s0, 0xF # mask
                  # Determine if the bits represent a number less than 10 (slti)
        slti $t1, $t0, 10 # Is character less than?
                  # If not (if greater than 9), go make a high digit
        beq $t1, $zero, MakeHighDigit

MakeLowDigit:
                  # Combine it with ASCII code for '0', becomes 0..9
        addi $t0, $t0, 48 # add 0
                  # Go output the digit
        j DigitOut

MakeHighDigit:
                  # Subtract 10 from low bits
        subi $t0, $t0, 10
                  # Add them to the code for 'A' (65), becomes a..f
        addi $t0, $t0, 65

DigitOut:
        move $a0, $t0      # Output the ASCII character
        li $v0, 11
        syscall

                  # Decrement loop counter
        subi $s1, $s1, 1 # decrement loop count
                  # Keep looping if loop counter is not zero
        bne $s1, $zero, Loop

# Output third string.

        la $a0, Output3
        li $v0, 4
        syscall


# Done, exit the program

        li $v0, 10  # This system call will terminate the program gracefully.
        syscall
```