**Conor Simmonds**

**COMPSYS304**

**Assignment 1**

**Student ID: 1728423**

**UID: csim080**

**Question 1:**

**1a)**

First instruction @ 0x0200 0500:

14-25-01-00

First 6 bits is not 0, so instruction is i-Type

6 bit OP : 5 bit Source Reg Address : 5 bit Target Reg Address : 16 bit offset

0001 01.00 | 001.0 0101. | 0000 0001 | 0000 0000

OP = 000101 = 5

Sr = 00001 = 1

St = 00101 = 5

Off = 0000 0001 0000 0000 = 256 // 0x0100, or just 0x100

**EXCECUTION:** bne $1, $5, 256 // branch if not equal, 256 is 0x100 in hex

Jump 0x100 \* 4 = 0x400, as 0x400 = 0x100 instructions + 4 for PC Counter + 0x500 (original location of Program Counter).

(i.e. **(0x100\*4) + 0x4 + 0x400**)

This adds to **0x904** (the location of the start of the next instruction)

Second instruction @ 0x0200 0904:

01-22-38-25

First 6 bits is not 0, so instruction is r-Type

6 bit OP (typically 0) : 5 bit Source Reg Address : 5 bit Target reg Address: 5 bit Destination Reg Address : 5 bit shift amount : 6 bit function code

0000 00.01 | 001.0 0010. | 0011 1.000 | 00.10 0101

OP = 000000 = 0

Rs = 01001 = 9

Rt = 00010 = 4

Rd = 00111 = 7

Shift = 00000 = 0 // not a bitshift operation, typical

Func = 100101 = 37 // 'or' operation in r-type

**EXCECUTION:** or $7, $9, $4

**1b)**

Big endian. In Big endian, the first byte in an instruction (the most significant byte) is at the top. So, where an instruction is contained in addresses 0x0500 -> 0x0503, the most significant bytes will be at memory address 0x0500. This is the case here, as the PC (Program Counter) starts at 0x0200 0500

**Question Two:**

4096 elements is 0x1000 in hex. As four memory locations will be needed per element, memory needed will span **0x3200->0x71FF,** thus the loop that performs the multiplications must be ending at 0x7200.

Lui $a0, 0x1000 # Initialise the register that will point to the next value

# to read. Start by loading the upper half of the word.

Ori $a0, $zero, 0x3200 # Load the seond part of the word.

Lui $a1, 0x1000 # Initialise the register that will point to 1 after last value to

# read. Start by loading the upper half of the word.

Ori $a1, $zero 0x7200 # Load the second part of the word.

Lui $a2, 0x1400 # Initialise the register that will point to the first location to

# write to. There is no lower half, so just load upper.

*Loop*:

Beq $a0, $a1, *Exit* # exit as soon as a1 holds address 0x7200

# will not write to this, last will have been 0x71FF

Lw $t2, 0($a0) # Load from memory what was at the address held by a0

Sll $t2, $t2, 5 # multiply by 2^5 (32) by doing 5 left bitshift operations

Sw $t2, 0($a2) # Store this value to the memory at the address in a2

Addi $a2, $a2, 4 # increment read location by 4

Addi $a0, $a0, 4 # increment write location by 4

J *Loop*

*Exit*

# Complete

**Question Three:**

**3a)**

*Evaluate:*

Lui $t3, [y-upper] # Store the upper value of y first

Ori $t3, $zero, [y-lower] # Store the lower value of y

Ori $t4, $zero, [x] # Store x

Ori, $t5, $zero, 1 # Initialize a register to hold value 1

Ori, $t6, $zero, [n] # Store n

Mult $t4, $t4 # Calculate x^2, store in $LO

Mul $LO, $t4 # Calculate x^3, store in $LO

Ori $t7, $zero, 1 # Initialize running score (eventually holding 2^n)

Ori $t8, $zero, 1 # Initialize counter ( 0 < $t8 < n )

*Loop*:

Sll $t8, $t8, 1 # Multiply by 2 first

Addi $t7, $t7, 1 # Increment the counter

Beq $t7, $t6, *Done* # Branch to ‘*Done*’ when $4 reaches size $3 (which holds n)

J *Loop*

*Done:* # t8 now holds 2^n

Div $t3, $t3, $t8 # evaluate **( y / 2^n )**

Add $t3, $t3, $t5 # evaluate **[ (y / 2^n ) + (1 –x^3) ]**

Jr $ra # Return back

**3b)**

.data # Data (string) goes here

strError: .asciiz “Error: arguments out of range. Exiting…”

.text # Instructions go here

.globl main # Declare main

*Main:*

Ori $t2, $zero, 1 # Set up a variable as 1 (for use in check-if statements)

Lw $t0, x\_address # Load x from memory

Slti $t1, $t0, 10 # Check to make sure x is lower than 10

Beq $t1, $zero, *Kill* # Kill if t1 is 0 (therefore x is greater than 10)

Slti $t1, $t0, 1 # Check to make sure x is higher than 0 (i.e. not less than 1)

Beq $t1, $t2, *Kill* # Kill if t1 is 1 (therefore x is less than 1)

Lw $t0, n\_address # Load n from memory

Slti $t1, $t0, 5 # Check to make sure n is lower than 10

Beq $t1, $zero, *Kill* # Kill if t1 is 0 (therefore n is greater than 10)

Slti $t1, $t0, 1 # Check to make sure n is higher than 0 (i.e. not less than 1)

Beq $t1, $t2, *Kill* # Kill if t1 is 1 (therefore n is less than 1)

Lw $t0, y\_address # Load y from memory

Slti $t1, $t0, 1 # Check to make sure y is higher than 0 (i.e. not less than 1)

Beq $t1, $t2, *Kill* # Kill if t1 is 1 (therefore y is less than 1)

# If reached this point, then all values are conforming to their respective ranges.

J *Evaluate*

*Kill:*

Ori $v0, $zero, 4 # syscall with 4 in v0 will print string whose address is in a0

Lui $a0, strPromptFirst # load higher portion of address of string

Ori $a0, $zero, strPromptFirst # load lower portion of address of string

Syscall # Print string pointed to by a0

Ori $v0, $zero, 10 # syscall with 10 in v0 will exit program

Syscall # Exit program

**Question Four:**

# Each matrix will only take up 25 values maximum (5^2) and the array will only take up 5 values

# maxium (5^1)

# Args are in $a0 -> $a3, the three base addresses and the length

#a0= X, a1 = Y, a2 = Z

*Evaluate:*

Li.d $f1, 1.5 # Store 1.5 in a floating point reg

Li.d $f2, 8 # Store 8 in a floating point reg

Li.d $f2, 16 # Store 8 in a floating point reg

Or $t0, $zero, $a3 # Store array/matrix length in a reg

# Create 4 \* length to use later for iteration purposes

Ori $t3, $zero, 4 # Store 4 for later purposes

Mult $t6, $t3, $t0 # Multiply length by the temporary ‘4’ value

Ori $t1, $zero, $zero # Initialise row counter

*Rows:*

Beq $t0, $t1, *End* # All rows done when t1 has counted to **length**

Ori $t2, $zero, $zero # Initialise column counter

*Columns:*

Beq $t0, $t2, *ColDone* # Cells in a row done when t2 has counted to **length**

# **==== Calculation for EACH cell in matrix ====**

# Calculate memory addresses to use

# Add (4\*length, i.e. contents of t9 reg) for each row passed to base address of each matrix

Mult $t4, $t1, $t6

# Then add cells on this column passed \* 4. This is the offset for reading and writing to the matrices

Mult $t5 $t3, $t2

Add $t4, $t4, $t5

# Load double-precision floating point value from X matrix

L.d $f3, $t4($a0)

# Divide by 8

Div.d $f3, $f3, $f2

# Add 1.5

Add.d $f3, $f3, $f1

# Find memory offset (i.e. row counter \* 4 away from base address) of array y

Mult $t7, $t1, $t3

# Read that value from memory

Lw $f4, $t7($a1)

# Convert integer value to a double precision floating point value

Cvt.d.w $f4, $f4

# Multiply value by 16

Mul.d $f4, $f4, $f2

# Add (16 \* Y[rownumber]) to (1.5 + (X[rownumber][colnumber] / 8))

Add.d $f3, $f3, $f4

# Store this value back in address in matrix Z

# As with X, this uses $t4 for offset, but now uses $a2 for base as a2 contains Z base address

S.d $f3, $t4($a2)

Addi $t2, $t2, 1 # Increment the column counter

J *Columns* # Go to next cell

# **==== END of Calculation for EACH cell in matrix ====**

*ColDone:*

Addi $t1, $t1, 1 # Increment the row counter

J *Rows* # Return to do next row (or exit if finished last row)

*End:*

Jr $ra

**# OPERATION COMPLETE!**