02155 - Computer architecture and Engineering
Fall 2022

Assignment 1

Group 31

Daniel F. Hauge (`s201186`)

This report contains 7 pages

October 2, 2022

A1.1

Figure 1: Problem a.

```
ADDI t0, x0, 8       // Sets temporary register to 8
REM t0, x4, t0       // Uses signed remainder ops similar to modulus.
BNE x0, t0, skip     // If x4 mod 8 is not 0, then x4 is not a multiple of 8.
ADD x1, x0, x0       // Set x1 to 0 when x4 mod 8 was actully 0.
skip:
NOP
```

Figure 2: Problem b. Bit mask way (Personal preference)

```
ANDI x1, x5, 31       // Applies an immediate bit mask on x5 stored in x1
```

Figure 3: Problem b. Remainder way

```
ADDI t0, x0, 32      // Sets temporary register to 32 for REM.
REM x1, x4, t0       // Uses signed remainder ops similar to modulus.
```

A1.2

Figure 4: Vector reader program

```
ADDI t0, x0, 0x10000    // Using t0 as read address for number in vector
ADD t1, x0, x0          // Using t1 as index i starting at 0
ADDI t2, x0, 100        // Using t2 for stop condition
ADDI a0, x0, 0x8000     // Init smalest number as largest value  32767
ADDI a1, x0, 0x7FFF     // Init largest number as smalest value -32768

LOOP:
BEG t1, t2, END         // End when i reach 100, meaning all numbers are read
LW t3, 0(t0)            // Load number from vector

BGE t3, a1, BIGGER      // Branch if number is largest so far
MaybeSmaller:
BLT t3, a0, SMALLER     // Branch if number is smalest so far
BNE x0, x0, CONTINUE

BIGGER:
ADDI a1, x0, t3         // Set largest number seen so far
BNE x0, x0, MaybeSmaller

SMALLER:
ADDI a0, x0, t3         // Set smalest number seen so far

CONTINUE:
ADD t1, t1, 1           // Increment i by one (i++)
ADDI t0, t0, 4          // Increment address to next number (4 bytes)
BNE x0, x0, LOOP        // Always branch to start of loop.

END:
NOP
```

I considered the case where the hundred numbers is always incrementing when reading the numbers sequentialy. To accomodate for this, going back with "MaybeSmaller" is used to also check if number is smaller than what has been observed.

A1.3

a.

Determining the execution time of the program can be found by finding total amount of cycles and dividing by the clock rate using the following equation.

$$\frac{\textbf{CPU Clock Cycles}}{\textbf{Clock rate}} = \textbf{CPU Time}$$

Using given information to get execution time.

$$\frac{10^6 \cdot 0.35 \cdot 3 + 10^6 \cdot 0.4 \cdot 4 + 10^6 \cdot 0.25 \cdot 5}{2000000000} = 0.00195$$

b.

Using same equation, with new given information to get execution time.

$$\frac{10^6 \cdot 0.35 \cdot 3 + 10^6 \cdot 0.35 \cdot 4 + 10^6 \cdot 0.25 \cdot 5 + 10^6 \cdot 0.05 \cdot 6}{2000000000} = 0.002$$

With a 500 micro seconds longer execution time, the modifications are not advantageous for the benchmark.

A1.4

x20 18 x22 16 x23 4 x24 2

168 -> [?] 240 -> [?][18,16,4,2] 260 -> [?][18,16,4,2] 360 -> [?][18,16,4,2][18,16,4,4] 364 -> [?][18,16,4,2][18,16,4,4] 388 -> [?][18,16,4,2] 280 ->

176 -> [?]

A1.5

Figure 5: Complex multiplication program

```
        . data
aa :    . word a # Re part of z
bb :    . word b # Im part of z
cc :    . word c # Re part of w
dd :    . word d # Im part of w

        . text
        . globl main
main :
    lw a0, aa
    lw a1, bb
    lw a2, cc
    lw a3, dd
    jal ra, complexMul # Multiply z and w
    nop
    j end # Jump to end of program
    nop

complexMul:
    MUL t0, a0, a2
    MUL t1, a1, a3
    MUL t2, a0, a3
    MUL t3, a1, a2
    SUB a0, t0, T1
    ADD a1, t2, t3
    jalr x0, 0(ra)

end:
    nop
```

a.

The procedure *complexMul* is a leaf procedure, as it does not call any other procedures.

b.

The stack was not used, as complex multiplication was achievable with reasonable convinience without the need to retain local variables that a non-leaf procedure needs. For example, no need to store return address or other temporary values that will be needed after secondary procedure call returns.

c.

4 MUL instructions contribute 8 cycles, sub, add and jalr contribute 3 cycles, which totals 11 cycles.

A1.6

Figure 6: Alternative Ccmplex multiplication program

```
        . data
aa :    . word a # Re part of z
bb :    . word b # Im part of z
cc :    . word c # Re part of w
dd :    . word d # Im part of w

        . text
        . globl main
main :
    lw a0, aa
    lw a1, bb
    lw a2, cc
    lw a3, dd
    jal ra, altComplexMul # Multiply z and w
    nop
    j end # Jump to end of program
    nop

altComplexMul:
    ADD t0, a0, a1
    MUL t0, t0, a2
    SUB t1, a3, a2
    MUL t1, t1, a0
    ADD t2, a2, a3
    MUL t2, t2, a1
    SUB a0, t0, t2
    ADD a1, t0, t1
    jalr x0, 0(ra)

end:
    nop
```

a.

altComplexMul is a leaf procedure as it does not call another procedure.

b.

The stack is not used as there was no need to save return address, temporaries or any values from registers that might be overwritten in a nested procedure call.

c.

3 MUL instructions contribute 6 cycles, 3 ADD instructions contribute 3 cycles, 2 SUB instructions contribute 2 cycles and jalr contribute one cycle, totaling 12 cycles

A1.7

a.

If RISC-V only had 2 argument registers and one result register, I would store the arguments in memory and pass the address of the arguments rather than the actual values. To return the results, again the results could be stored in memory and address to the results would be given instead of actual values. This way composite structures like imaginary numbers, but also other structures that contain multiple values like strings can be used. Both techniques are fundamental when looking at procedure call patterns: **Call By Reference** and **Call By Value** that is used in high level programming languages.

b.

No registers are saved across a procedure call without additional instructions. Only registers that is put in the stack or stored elsewhere in memory with instructions will be saved. It is convention to save and restore return address and used temporaries to x8-x9 and x18-x27 registers from the stack.