ELEC3441 HW1

JIANG Feiyu 3035770800

Part A

A.1 Iron Law

A.1.1 √

Accodring to the definition of Cycle Per Instruction(CPI),

$CPI = \frac{CycleCount}{InstructionCount}$

while according to the diagram provided:

- **program P**:
  - processor A
    $CPI = \frac{1000000\times1+300000\times4+500000\times20}{1000000+300000+500000} = \frac{12200000}{1800000} = 6.7778$
  - processor B
    $CPI = \frac{1000000\times1+300000\times20+500000\times12}{1000000+300000+500000} = \frac{13000000}{1800000} = 7.2222$
- **program Q**:
  - processor A
    $CPI = \frac{500000\times1+250000\times4+125000\times20}{500000+250000+125000} = \frac{4000000}{875000} = 4.5714$
  - processor B
    $CPI = \frac{500000\times1+250000\times20+125000\times12}{500000+250000+125000} = \frac{7000000}{875000} = 8$
- **program R**:
  - processor A
    $CPI = \frac{1000\times1+3000\times4+2000\times20}{1000+3000+2000} = \frac{53000}{6000} = 8.8333$
  - processor B
    $CPI = \frac{1000\times1+3000\times20+2000\times12}{1000+3000+2000} = \frac{85000}{6000} = 14.1667$

A.1.2 √

Accodring to the definition of CPU time,

$CPU\ time_{B,P} = Count\ of\ cycles \times clock\ period$

For processor A, it runs at 1.5 GHz, which has the clock period of $\frac{1}{1.5\times10^9} = 6.667 \times 10^{-10}$

For processor B, it runs at 1.2 GHz, which has the clock period of $\frac{1}{1.2\times10^9} = 8.833 \times 10^{-10}$

- **For program P**
  $CPU\ time_{A,P} = (1000000 \times 1 + 300000 \times 4 + 500000 \times 20) \times 6.67 \times 10^{-10} = 8.13ms$
  $CPU\ time_{B,P} = (1000000 \times 1 + 300000 \times 20 + 500000 \times 12) \times 8.83 \times 10^{-10} = 10.83ms$
  Therefore, $\frac{CPUTime_{B,P}}{CPUTime_{A,P}} = 1.332$. Processor A is 1.332× faster than processor B.
- **For program Q**
  $CPU\ time_{A,Q} = (500000 \times 1 + 250000 \times 4 + 125000 \times 20) \times 6.67 \times 10^{-10} = 2.667ms$
  $CPU\ time_{B,Q} = (500000 \times 1 + 250000 \times 20 + 125000 \times 12) \times 8.83 \times 10^{-10} = 5.833ms$
  Therefore, $\frac{CPUTime_{B,P}}{CPUTime_{A,P}} = 2.188$. Processor A is 2.188× faster than processor B.
- **For program R**
  $CPU\ time_{A,R} = (1000 \times 1 + 3000 \times 4 + 2000 \times 20) \times 6.67 \times 10^{-8} = 0.035ms$
  $CPU\ time_{B,R} = (1000 \times 1 + 3000 \times 20 + 2000 \times 12) \times 8.83 \times 10^{-8} = 0.071ms$
  Therefore, $\frac{CPUTime_{B,P}}{CPUTime_{A,P}} = 2.005$. Processor A is 2.005× faster than processor B.

A.1.3

- When considering the total runtime of these 3 programs on processor A and B, for the geometric mean:
  $Average(geometricmean_A) = (8.13 \times 2.667 \times 0.035)^{\frac{1}{3}} = 0.912,$
  $Average(geometricmean_B) = (10.83 \times 5.833 \times 0.071)^{\frac{1}{3}} = 1.649$
  A is $\frac{1.649}{0.912} = 1.808$ times faster.

A.1.4

Consider the total runtime of these 3 programs on processor A and B:

$CPUTime_{A,total} = CPUTime_{A,P} + CPUTime_{A,Q} + CPUTime_{A,R} = 10.832ms$

$$CPUTime_{B,total} = CPUTime_{B,P} + CPUTime_{B,Q} + CPUTime_{B,R} = 16.734ms$$

Processor A is faster than processor B when considering the whole running time, which indicates we should improve on processor B. For processor B, the number of cycles spent on each class of instruction are:

ALU	Branch/Jumps	Load/Store
1501000	11060000	7524000

According to `Amdahl's law`, $speedup = \frac{1}{(1-P)+\frac{P}{S}}$, we should try to improve the performance of `Branch/Jumps` instructions. Assuming the `Branch/Jumps` instruction can be infinitely fast, the the maximum speed up should be: $\frac{1501000+11060000+7524000}{1501000+7524000} = \frac{20085000}{9025000} = 2.225$

A.1.5

In this case, we should focus on the instruction type that has the highest CPI on the slower processor, since reducing its cycle time will have the largest impact on the overall performance of the processor.

In this case, the CPI for `Branch/Jumps` instructions is the highest on processor B (20 cycles per instruction) compared to processor A (4 cycles per instruction).

Therefore, improving the cycle time of `Branch/Jumps` instructions on processor B would have the largest impact on the overall performance of the processor

A.2 Benchmark Performance

A.2.1 √

1. Arithmetic mean of raw CPU time:
   $A : 46666.667$
   $B : 46666.667$
   $C : 46666.667$
2. Geometric mean of raw CPU time:
   $A : 40000.000$
   $B : 40000.000$
   $C : 40000.000$
3. Arithmetic mean of normalized CPU time:
   $A : 1.0$
   $B : 1.417$
   $C : 1.667$
4. Geometric Mean of Normalized CPU Time:
   $A : 1.0$
   $B : 1.0$
   $C : 1.0$
5. Total raw CPU time:
   $A : 140000$
   $B : 140000$
   $C : 140000$
6. Total normalized CPU time:
   $A : 3.0$
   $B : 4.25$
   $C : 5.0$

A.2.2 √

1. When considering about metrics (1),(2),(4),(5), all the processors are equally fast. When considering about metrics (3),(6), processor A is the fastest.
2. (1),(2),(4),(5)
3. For fair comparison, arithmetic mean has the limitation, thus metric (2) and (4) should be used.

A.2.3

1. Arithmetic mean of raw CPU time:
   $A : 35300$
   $B : 35600$
   $C : 35150$
2. Geometric mean of raw CPU time:
   $A : 16647.166$
   $B : 19796.928$
   $C : 13998.542$
3. Arithmetic mean of normalized CPU time:
   $A : 1.0$
   $B : 1.563$
   $C : 1.375$
4. Geometric Mean of Normalized CPU Time:
   $A : 1.0$
   $B : 1.189$
   $C : 0.841$
5. Total raw CPU time:
   $A : 141200$
   $B : 142400$
   $C : 140600$
6. Total normalized CPU time:
   $A : 4.0$
   $B : 6.25$
   $C : 5.5$

Metrics (1),(2),(4),(5) shows C is the fastest. When considering about (3) and (6), A is the second and overall B is the slowest one.

A.2.4

Consider only P1 to P3 and P5:

1. Arithmetic mean of raw CPU time:
   $A : 3035000$
   $B : 6035000$
   $C : 1535000$

2. Geometric mean of raw CPU time:
   $A: 166471.658$
   $B: 197969.280$
   $C: 139985.420$
3. Arithmetic mean of normalized CPU time:
   $A: 1.0$
   $B: 1.563$
   $C: 1.375$
4. Geometric Mean of Normalized CPU Time:
   $A: 1.0$
   $B: 1.189$
   $C: 0.841$
5. Total raw CPU time:
   $A: 12140000$
   $B: 24140000$
   $C: 6140000$
6. Total normalized CPU time:
   $A: 4.0$
   $B: 6.25$
   $C: 5.5$

Metrics (1),(2),(4),(5) remain the conclusion, while C is the second one in (3) and (6), it's still the fatest one.

A.2.5

I would like to use the metric (2).
It's possible if t_A, t_B, t_C of P6 equals to `[1000000000, 1, 1000000000]`, at this time,

```
Arithmetic Mean:

Proc.A: 168690200.0
Proc.B: 4023733.5
Proc.C: 167690100.0

Geometric Mean:
Proc.A: 311953.89391571033
Proc.B: 12428.930023815428
Proc.C: 247597.96968368735
```

This will make processor B the fastest.

A.3 RISC-V Assembly Programming1

A.3.1 Arithmetic and logic operations 2

- `z = z + (x - (y << 1))`

```
slli a1, a1, 1 // Shift y left by 1 bit, equivalent to y*2
sub t0, a0, a1 // Subtract result from x
add a2, a2, t0 // Add the result to z
```

- `y = x + 0x757`

```
addi a1, a0, 0x757
```

- `y = x + 0x7800A757`

```
lui tO, 07800A
addi t0, tO, OxA
add a1, a0, t0
```

- `a[3] = a[2] + 1`

```
// since each int is 4 bytes
lw t0, 8(a3)        // Load word from array index 2 (offset by 8 bytes)
addi t0, t0, 1      // Add 1 to the value
sw t0, 12(a3)       // Store word to array index 3 (offset by 12 bytes)
```

- `a[x & 0xF] = y % 16` (Assume a size less than 4byte counts as 4byte, a[16] is an array of ints)

```
andi t0, a0, 0xF // And x with 0xF
slli t0, t0, 2 // Multiply the result by 4 to get the byte offset
add t0, t0, a3 // Add the base address of the array to the offset
andi t1, a1, 0xF // y % 16 is equivalent to y & 0xF
sw t1, 0(t0) // Store the result at the computed address
```

- `z = 4*y + 2*x`

```
slli a2, a1, 2       // Shift y left by 2 bits
slli t0, a0, 1       // Shift x left by 1 bit
add a2, a2, t0       // Add
```

A.3.2 Branches and Jumps

- if-else

```
# if (x > y) then z = x − y else z = y − x

      bge  a1, a0, ELSE       # jump if y>=x
      sub  a2, a0, a1         # x−y
      j    END

ELSE: sub  a2, a1, a0         # y−x
END:  nop
```

- Nested Conditional if-else

```
# if (x < y && x > z) {

      # jump if any of condition not satisfied
      bgeu a0, a1, END        # x>=y
      bgeu a2, a0, END        # z>=x

      slli t0, a0, 2          # calculate the address offset
      add  t0, t0, a3         # compute address
      lw   t1, 0(t0)          # load a[x] in t1
      add  a0, a0, t1         # a[x]+x
END:  add  t0, a1, a2         # y+z
      add  a0, a0, t0         # x+(y+z)
```

- For-loop

```
# for (x = 0; x < 10; x++)
      addi a0, zero, 0        # initialize x to 0
      addi t0, zero, 10

FOR:  bge a0, t0, END         # end if x>=10
      slli t1, a0, 2          # calculate the address offset
      add  t1, t1, a3         # compute address
      sw   a0, 0(t1)          # store the value of a0 at address t1

      addi a0, a0, 1 # x+=1
      j    FOR                # iteration
END:  nop
```

- While-loop

```
# x = 0;
# while(x <= 10) { a[x] = y; y++; x++; }

      addi a0, zero, 0        # initialize x to 0
      addi t0, zero, 10

FOR:  blt t0, a0, END         # end if x > 10
      slli t1, a0, 2          # calculate the address offset
      add  t1, t1, a3         # compute address
      sw   a1, 0(t1)          # store the value of a1 at address t1

      addi a1, a1, 1          # y++
      addi a0, a0, 1          # x++
      j FOR                   # iteration
END: nop
```

A.4 Fibonacci Numbers

A.4.1 Branches and Jumps

Assume `n` is stored in `a0` and `F` is stored in `a1`

```
fib:
    sw        zero,0(a1)          # store F[0] = 0
    li        a5,1
    sw        a5,4(a1)            # store F[1] = 1
    ble        a0,a5,.L2           # base case, jump to .L2 if n <= 1

    mv        a5,a1
    slli a2,a0,2
    add        a2,a2,a1
    addi a2,a2,-4

.L3:                          # entrance of the loop
    lw        a4,4(a5)           # Read the value of
    lw        a3,0(a5)
    add        a4,a4,a3
    sw        a4,8(a5)
    addi  a5,a5,4
    bne        a5,a2,.L3          # Check if a5 is equal to a2; if not, continue

.L2:
    slli  a0,a0,2
    add        a1,a1,a0
    lw        a0,0(a1)           # Read the value of F[n] from a1 and store it in a0
    ret                       # return the value
    .size  fib, .-fib
    .align  2

main:
    addi  sp,sp,-416
    sw        ra,412(sp)
    mv        a1,sp

    # load the value of n into a0, here suppose n = 10
    li        a0,10
    call  fib

    li        a0,0
    lw        ra,412(sp)
    addi sp,sp,416
```

```
    jr        ra
```

A.4.2 Recursive

Assume `n` is stored in `a0`, return value `r` in `a0`

```
.fib:
    addi  sp,sp,-16      # Adjusts the stack pointer
    sw        ra,12(sp)     # Saves the return address on the stack
    sw        s0,8(sp)      # Saves the value of register s0 in the stack
    sw        s1,4(sp)      # Save the value of register s1 in the stack

    # Base Cases
    mv        s0,a0
    li        a5,2          # Load immediate count
    blt        a0,a5,.L1     # If a0 < 2, return r

    # Recurive cases
    addi  a0,a0,-1
    call  fib            # call fib(n-1)
    mv        s1,a0
    addi  a0,s0,-2
    call  fib            # call fib(n-2)
    add  a0,s1,a0

.L1: # this is the base case
    lw        ra,12(sp)
    lw        s0,8(sp)
    lw        s1,4(sp)
    addi sp,sp,16
    jr        ra

main:
    addi  sp,sp,-16
    sw        ra,12(sp)
    li        a0,15          # suppose n=15
    call  fib
    li        a0,0
    lw        ra,12(sp)
    addi  sp,sp,16
    jr  ra
```

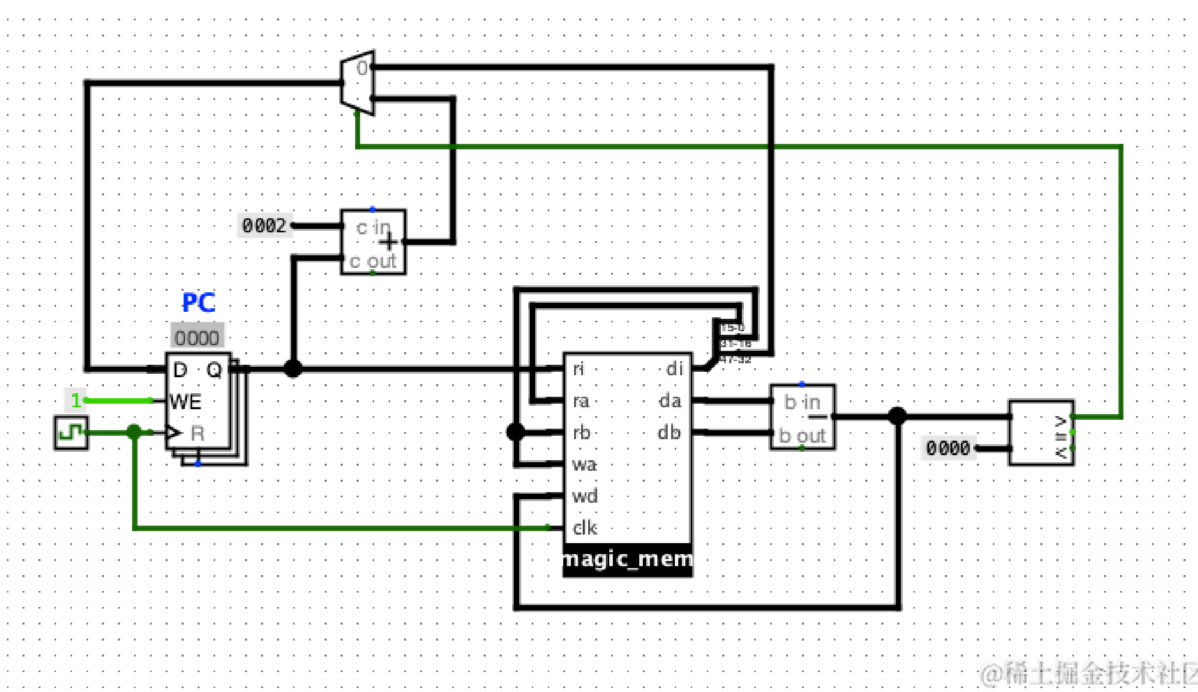
A.5 One Instruction Set Computer

A.5.1 Branches and Jumps

- **When no pipeline is applied**
    - **instruction per program**: `SUBLEQ` requires much more instructions per program.
    - **cycle per instruction**: Both processors run on 1 CPI.
    - **cycle time**: The cycle time of `RISC-V` should be about twice as long as `SUBLEQ`, as `RISC-V` performs two memory operations each cycle, and only one for `SUBLEQ`.

  Depending on the program, when the number of instructions required is more than twice as much as RISC-V, RISC-V will run faster.
- **When pipeline is applied**
    - **instruction per program**
    - **cycle per instruction**: `RISC-V` can run a higher CPI than `SUBLEQ`, because `RISC-V` involves more operations in a single cycle and is easier to pipeline into more stages.
    - **cycle time**: The cycle time should be close. So `RISC-V` will run much faster.

A.5.2 SUBLEQ processor design



A.5.3 SUBLEQ RISC-V implementation

```
subleq:
    sub a1, a1, a0    # memory[b] = memory[b] - memory[a]
    bgtz a1, skip     # branch to skip if memory[b] > 0
    jalr zero, a2     # goto c
skip:
    # Continue with the rest of the program
```