ELEC3441: Computer Architecture
Second Semester, 2023–24

# Homework 1 (r1.0)

# *Part A: Problem Set*

## A.1 Iron Law

### A.1.1

$$CPI_{A,P} = \frac{\#\text{cycles}}{\#\text{instructions}} = \frac{1 \times 1000000 + 4 \times 300000 + 20 \times 500000}{1000000 + 300000 + 500000} = 6.78$$

Similarly:

$$CPI_{A,Q} = 1.58$$
$$CPI_{A,R} = 8.83$$
$$CPI_{B,P} = 7.22$$
$$CPI_{B,Q} = 2.14$$
$$CPI_{B,R} = 14.17$$

**A.1.2** To determine which processor results in higher performance for the programs P, Q, and R, we need to calculate the execution time for each program on both processors. First, we need to calculate the total number of cycles for each program by multiplying the number of instructions by the cycles per instruction for each class (ALU, Branch/Jumps, Load/Store), then sum these products to find the total cycles for the program. We can convert the clock rate from GHz to cycles per second for each processor.

$$\begin{aligned}\text{CPUTime}_{A,P} &= \#\text{cycles} \times \text{clock period} \\ &= (1 \times 1000000 + 4 \times 300000 + 20 \times 500000) \times 6.67 \times 10^{-8} \\ &= 8133.33\,\mu\text{s} \\ \text{CPUTime}_{B,P} &= 10\,833.33\,\mu\text{s}\end{aligned}$$

Therefore,

$$\frac{\text{CPUTime}_{B,P}}{\text{CPUTime}_{A,P}} = 1.33$$

Processor $A$ is 1.33× faster than processor $B$ for program P.

$$\text{CPUTime}_{A,Q} = 2666.67\,\mu\text{s}$$
$$\text{CPUTime}_{B,Q} = 5833.33\,\mu\text{s}$$

Therefore,

$$\frac{\text{CPUTime}_{B,Q}}{\text{CPUTime}_{A,Q}} = 2.19$$

Processor $A$ is 2.19× faster than processor $B$ for program Q.

$$\text{CPUTime}_{A,R} = 35.33\,\mu\text{s}$$
$$\text{CPUTime}_{B,R} = 70.83\,\mu\text{s}$$

Therefore,

$$\frac{\text{CPUTime}_{B,R}}{\text{CPUTime}_{A,R}} = 2.00$$

Processor $A$ is 2.00× faster than processor $B$ for program R. With the provided clock rate, Processor $A$ is now faster for all programs compared to Processor $B$.

**A.1.3** To determine which processor is faster when considering all three programs as a whole, and by how much faster on average (using the geometric mean), you can calculate the ratio of execution times between the two processors and the geometric mean of them.

The geometirc mean is calculated by multiplying all the ratios together and then taking nth root of the product,where n is the number of ratios.

$$\text{GM} = (\prod_{i=1}^{n} x_i)^{\frac{1}{n}}$$

The ratio of execution times have been given in the last question, and the geometric mean of these ratios is approximately 1.80. Indicating on average, across all three programs, Processor $A$ is 1.80× faster than processor $B$.

**A.1.4** Consider the total runtime of these 3 programs on processor A and B:

$$\text{CPUTime}_{A,total} = \text{CPUTime}_{A,P} + \text{CPUTime}_{A,Q} + \text{CPUTime}_{A,R} = 10\,835.33\,\mu\text{s}$$
$$\text{CPUTime}_{B,total} = \text{CPUTime}_{B,P} + \text{CPUTime}_{B,Q} + \text{CPUTime}_{B,R} = 16\,737.49\,\mu\text{s}$$

Processor $A$ is faster than processor $B$ when using total runtime as metric. Therefore, we'd like to improve the performance of processor $B$. To calculate the maximum speedup possible by improving one of the instruction types (ALU, Branch/Jump, Load/Store) of the slower processor (Processor B in this case), we must first determine which type of instruction, if improved, would have the greatest impact on the total execution time.

Now consider processor $B$, the number of cycles spent on each class of instruction are:

| | |
|---|---|
| ALU | 1501000 |
| Load/Store | 7524000 |
| Branch/Jump | 11060000 |

According to Amdahl's law, we should try to improve the performance of Branch/Jump instructions. The speedup for any enhancement is defined as (Amdahl's Law):

$$\text{Speedup} = \frac{1}{(1-P) + \frac{P}{S}}$$

Assuming you can make Branch/Jump instructions infinitely fast, the cycles per instruction for the improved instruction type will be reduced. Then the maximum speed up is:

$$\frac{1501000 + 7524000 + 11060000}{1501000 + 7524000} = 2.23$$

Therefore, the maximum speedup possible when using the total runtime (not geometric mean) for the three programs would be achieved by improving the Branch/Jump instructions, which yields a speedup factor of approximately 2.23.

**A.1.5** The geometric mean of speedup across all three programs when improving each type is:

$$\text{Geometric Mean of Speedup} = \sqrt[3]{\text{Speedup for P} \times \text{Speedup for Q} \times \text{Speedup for R}}$$

When we improve the performance of Branch/Jump instructions infinitely fast.

For program $P$:

$$\text{Speedup}_P = \frac{\text{Orignial Total Runtime}_P}{\text{New Total Runtime}_P} = \frac{13000000}{7000000} = 1.857$$

For program $Q$:

$$\text{Speedup}_Q = \frac{\text{Orignial Total Runtime}_Q}{\text{New Total Runtime}_Q} = \frac{7000000}{2000000} = 3.5$$

For program $R$:

$$\text{Speedup}_R = \frac{\text{Orignial Total Runtime}_R}{\text{New Total Runtime}_R} = \frac{85000}{25000} = 3.4$$

For the geometric mean of the speedups:

$$\text{Geometric Mean of Speedup} = \sqrt[3]{\text{Speedup}_P \times \text{Speedup}_Q \times \text{Speedup}_R} = 2.81$$

Similarly, following the same process, you can get 1.06 for the ALU instructions, and 1.488 for the Load/Store instructions.

## A.2 Benchmark Performance

### A.2.1

(i) Arithmetic mean of *raw CPU time*:

$A$ : 46666.67 $B$ : 46666.67 $C$ : 46666.67

(ii) Geometric mean of *raw CPU time*:

$A$ : 40000.00 $B$ : 40000.0 $C$ : 40000.0

(iii) Arithmetic mean of *normalized CPU time*:

$A$ : 1.00 $B$ : 1.42 $C$ : 1.67

(iv) Geometric mean of *normalized CPU time*:

$A$ : 1.00 $B$ : 1.00 $C$ : 1.00

(v) Total raw CPU time:

$A$ : 140000 $B$ : 140000 $C$ : 140000

(vi) Total normalized CPU time:

$A$ : 3 $B$ : 4.25 $C$ : 5.00

### A.2.2

1. When consider the metrics (iii), (vi), the processor A will be the fastest. Consider the metrics (i), (ii), (iv), (v), all the processors are equally fast.
2. Metric (i), (ii), (iv) and (v)
3. For fair comparison, metric (ii) and (iv) should be used. The Geometric mean of normalized CPU time is typically the best choice for a fair comparison of the performance of the processors when dealing with normalized values, as it equalizes the scale of performance variance and is less affected by extreme values in the data set. However, since it yields a value of 1 for all processors, it indicates no relative difference when normalized against Processor $A$. In the case where all processors complete the benchmarks in exactly the same total time, and the geometric mean of their performance ratios is also equal, either of these metrics would be suitable to claim that their performance is equivalent. Given that the arithmetic mean of the raw CPU time and the total raw CPU time both suggest the processors are equally good, they could also be considered fair.

**A.2.3** GM shows C is fastest.

For $P1$ to $P4$:

(i) Arithmetic mean of *raw CPU time*:

$A : 35300.00 \quad B : 35600.00 \quad C : 35150.00$

(ii) Geometric mean of *raw CPU time*:

$A : 16647.17 \quad B : 19796.93 \quad C : 13998.54$

(iii) Arithmetic mean of *normalized CPU time*:

$A : 1.00 \quad B : 1.56 \quad C : 1.38$

(iv) Geometric mean of *normalized CPU time*:

$A : 1.00 \quad B : 1.19 \quad C : 0.84$

(v) Total raw CPU time:

$A : 141200.00 \quad B : 142400.00 \quad C : 140600.00$

(vi) Total normalized CPU time:

$A : 4.00 \quad B : 6.25 \quad C : 5.50$

Considering these results, the metrics that suggest Processor $C$ is the fastest, through the arithmetic mean of raw CPU time, and the total raw CPU time. However, if considering all six different ways of comparison, Processor $A$ could also be argued as the best performance based on the total normalized CPU time, consistently performing with a normalized score of 1. Processor $C$ could be seen as more efficient in certain benchmarks, but Processor $A$ demonstrates strong, consistent performance across all benchmarks.

**A.2.4** For $P1$ to $P5$:

(i) Arithmetic mean of *raw CPU time*:

$A : 3035000.00 \quad B : 6035000.00 \quad C : 1535000.00$

(ii) Geometric mean of *raw CPU time*:

$A : 166471.66 \quad B : 197969.28 \quad C : 139985.42$

(iii) Arithmetic mean of *normalized CPU time*:

$A : 1.00 \quad B : 1.56 \quad C : 1.38$

(iv) Geometric mean of *normalized CPU time*:

$A : 1.00 \quad B : 1.19 \quad C : 0.84$

(v) Total raw CPU time:

$A : 12140000.00 \quad B : 24140000.00 \quad C : 6140000.00$

(vi) Total normalized CPU time:

$A : 4.00 \quad B : 6.25 \quad C : 5.50$

Results (i), (ii), (iv) and (v) have same conclusion. Processor $C$ is fastest. Processor $A$ is second. Processor $B$ is slowest. Results (iii) and (vi) are meaningless.

**A.2.5** To make Processor $B$ stand out as the fastest with an additional benchmark $P6$, we would want to ensure that Processor $B$ has a significantly better runtime for $P6$ than Processors $A$ and $C$.

## A.3 RISC-V Assembly Programming

In this question, you will practice writing assembly programs using the RISC-V assembly code. In all cases, assume the C variables are stored in the following registers:

| C Declaration | Variable | Register |
|---|---|---|
| `unsigned x` | `x` | `a0` |
| `unsigned y` | `y` | `a1` |
| `unsigned z` | `z` | `a2` |
| `unsigned a[16]` | `a` | `a3` |

**A.3.1 Arithmetic and logic operations** Implement the following C code segment as RISC-V assembly code. When in doubt, check with the RISC-V gcc toolchains you learned in Lab 1.

- `z = z + (x - (y << 1))`
- `y = x + 0x757`
- `y = x + 0x7800A757`
- `a[3] = a[2] + 1`
- `a[x & 0xF] = y % 16` *Hint:* you don't need to use modulo operator.
- `z = 4*y + 2*x` *Hint:* you don't need to use multiplication.

- 
```
slli t0, a1, 1
sub  t0, a0, t0
add  a2, a2, t0
```

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

- 
```
lui  t0, 0x7800A
addi t0, t0, 0x757
add  a1, a0, t0
```

- 
```
lw   t0, 8(a3)
addi t0, t0, 1
sw   t0, 12(a3)
```

- 
```
andi t0, a0, 0xF
slli t0, t0, 2
add  t0, a3, t0
andi t1, a1, 0xF
sw   t1, 0(t0)
```

- 
```
slli t0, a1, 2
slli t1, a0, 1
```

```
    add  a2, t0, t1
```

**A.3.2 Branches and Jumps** Implement the following C code segment as RISC-V assembly code.

- ```
  if (x > y) {
      z = x - y;
  } else {
      z = y - x;
  }
  ```

- ```
  if (x < y && x > z) {
      x = x + a[x];
  }
  x = x + y + z;
  ```

- ```
  for (x = 0; x < 10; x++) {
      a[x] = x;
  }
  ```

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

- ```
          bgeu a1, a0, ELSE
          sub  a2, a0, a1
          j    END
    ELSE: sub  a2, a1, a0
    END:  nop
  ```

- ```
          bgeu a0, a1, END
          bgeu a2, a0, END
          slli t0, a0, 2
          add  t0, t0, a3    # compute address of a[x]
          lw   t1, 0(t0)     # load a[x] in t1
          add  a0, a0, t1
    END:  add  t0, a1, a2
          add  a0, a0, t0
  ```

- ```
          addi a0, zero, 0
          addi t0, zero, 10
    FOR:  bgeu a0, t0, END
  ```
  ```

```
        slli t1, a0, 2
        add  t1, t1, a3
        sw   a0, 0(t1)
        addi a0, a0, 1
        j    FOR
END:    nop
```

- 

```
        addi a0, zero, 0
        addi t0, zero, 10
FOR:    bltu t0, a0, END
        slli t1, a0, 2
        add  t1, t1, a3
        sw   a1, 0(t1)
        addi a1, a1, 1
        addi a0, a0, 1
        j    FOR
END:    nop
```

## A.4 Fibonacci Numbers

### A.4.1

```
Init:
  addi  s0,  a0,  1;    # set s0 to be n+1
  sw    s0,  0(a1);     # store 0 in F[0]
  li    t3,  1;
  sw    t3,  4(a1);     # store 1 in F[1]
  li    t0,  2;         # loop index, it ranges from 2 to n

Loop:
  lw    t1,  0(a1);     # load F[i-2] to t1
  lw    t2,  4(a1);     # load F[i-1] to t2
  add   t3,  t1,  t2;   # calculate F[i]
  sw    t3,  8(a1);     # store result in F[i]
  addi  t0,  t0,  1;    # loop index update
  addi  a1,  a1,  4;    # address updates for next iteration
  bne   t0,  s0,  Loop; # Loop continue or not
  addi  a0,  t3,  0;    # copy result to a0
```

### A.4.2

```
  fib:
    addi  sp,   sp,   -16     # creat stack
    sw    ra,   12(sp)
    sw    s0,   8(sp)
    sw    s1,   4(sp)
    addi  s0,   a0,   0       # save n to s0
    addi  t0,   zero, 2
    blt   a0,   t0,   Return  # compare n and 2
    addi  a0,   a0,   -1
    jal   ra,   fib           # call fib(n-1)
    addi  s1,   a0,   0       # save return value to s1
    addi  a0,   s0,   -2
    jal   ra,   fib           # call fib(n-2)
    add   a0,   s1,   a0
  Return:
    lw    ra,   12(sp)
    lw    s0,   8(sp)
    lw    s1,   4(sp)
    addi  sp,   sp,   16
    jalr  zero, 0(ra)         # return
```



## A.5 One Instruction Set Computer

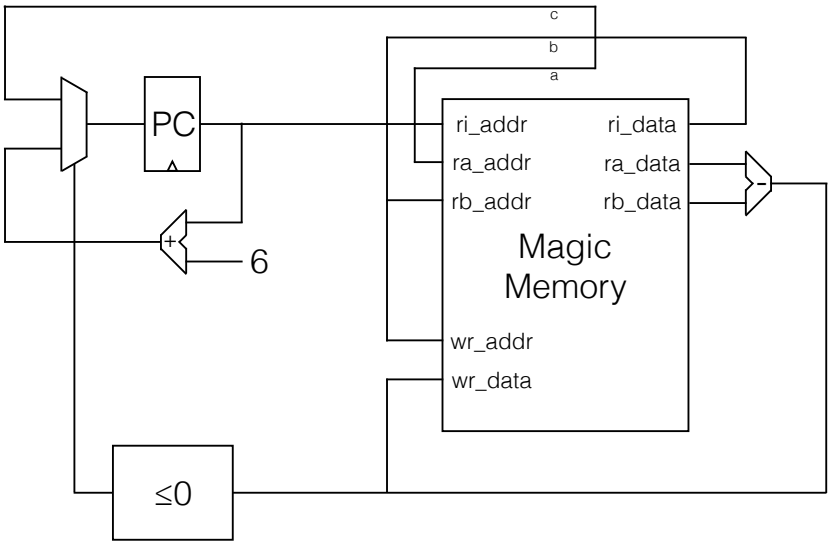
**A.5.1** (i) Any program will require many more SUBLEQ instructions to implement the same functionality as a RISC-V, since any operation not a load/store or subtraction will have to be implemented as a complex sequence of SUBLEQ instructions.

(ii) Executing each instruction requires 3 memory accesses: 2 reads and 1 write. If we assume a magic memory with enough ports to fulfil both instruction and data loads in a single cycle without any resource conflicts (similar to a register file but larger), then CPI will not be affected much compared to RISC-V

by memory. However, each instruction is also a branch, so in the pipelined case, CPI will be higher due to the larger effect of branch misprediction.

(iii) Cycle time would be slightly lower for SUBLEQ as there are fewer different cases to handle, so there is no logic required for decoding or choosing which data path to use.

SUBLEQ can win only on term (iii), and there only by a small amount. However the penalty for (i) is going to be huge, so that overall a SUBLEQ machine will be much slower than a RISC-V at performing the same functionality.




**A.5.2**

**A.5.3**