Quiz 1 (upload it to Gradescope)

Fall 2021

Don't forget to sign-in when you are leaving.

(Additional tables, information, notes are available on pages 4-7.)

Q1 [40 Points]. Consider the following latencies for different units in the (single-cycle) datapath described in Lec. 5 (shown in page 4):

Mem (I and D)	**Reg. File (read)**	**Reg. File (write)**	**Mux (any size)**	**ALU**	**Gate (any)**
380 ps	180 ps	10 ps	25 ps	200 ps	5 ps

Sign. Ext.	**PC (read or write)**	**Adder**	**Control**
45 ps	20 ps	170 ps	60 ps

Consider the addition of a multiplier to the ALU of the CPU described in the lecture with the timing shown above. This addition will add 200 ps to the latency of the ALU (i.e., the new ALU latency will be 400 ps), but will reduce the total number of instructions by 15% (because there will no longer be a need to emulate the multiply instruction).

Answer the following questions (show your work):

a) [20 points] What is the clock cycle time with and without this improvement? (hint: the maximum delay is determined by either LW or BEQ instruction.)

b) [10 points] What is the speedup achieved by this change?

c) [10 points] What is the slowest the new ALU can be that result in improved performance?

Question 1 Answer)

a)

An analysis for the various instructions shows that the longest latency belongs to the LW instruction. Thus, our clock cycle time needs to be greater than the LW instruction. Fig. 1 shows the latency of LW without the multiply function, and Fig. 2 shows the latency of LW with the multiply function.

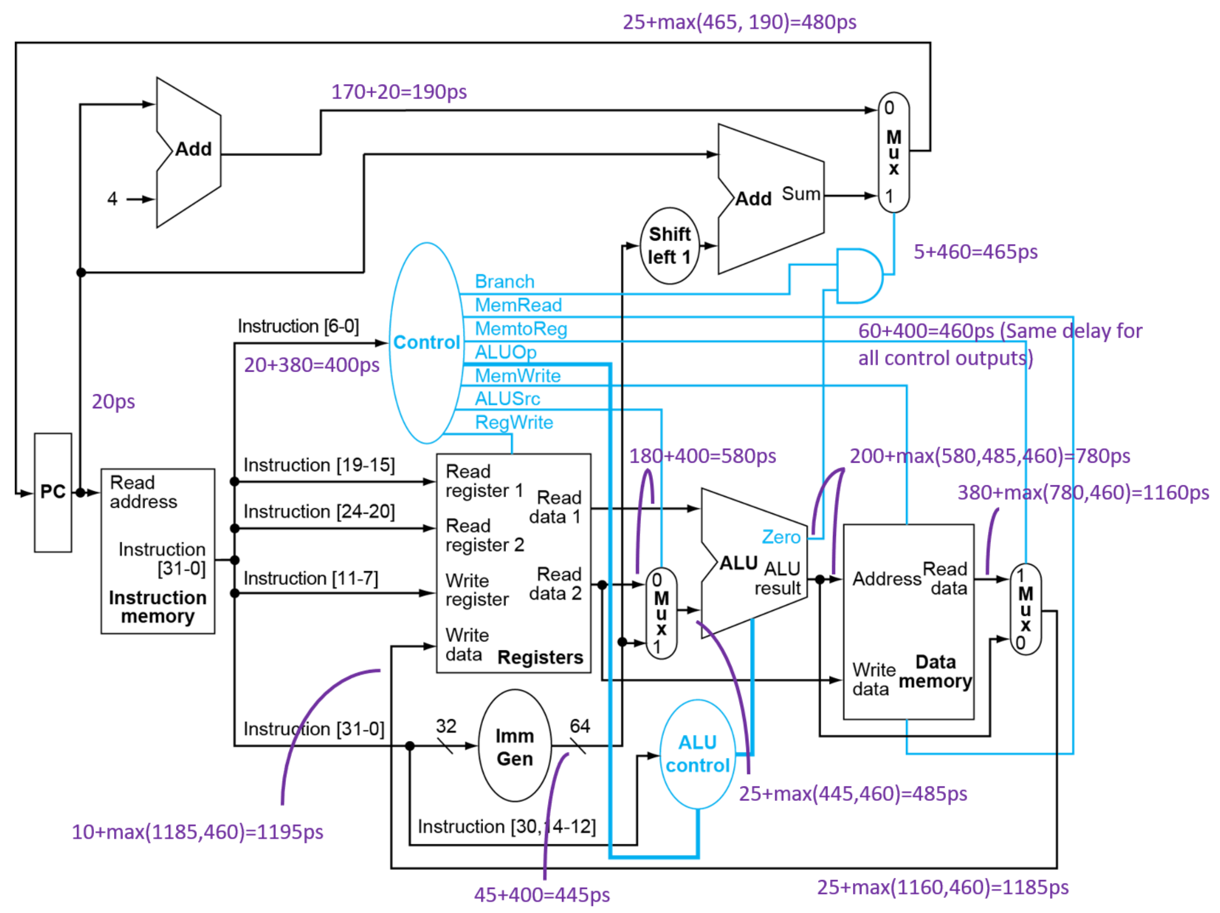

Fig 1: CPU, Normal ALU

Clock Cycle without improvement: > 1195 ps

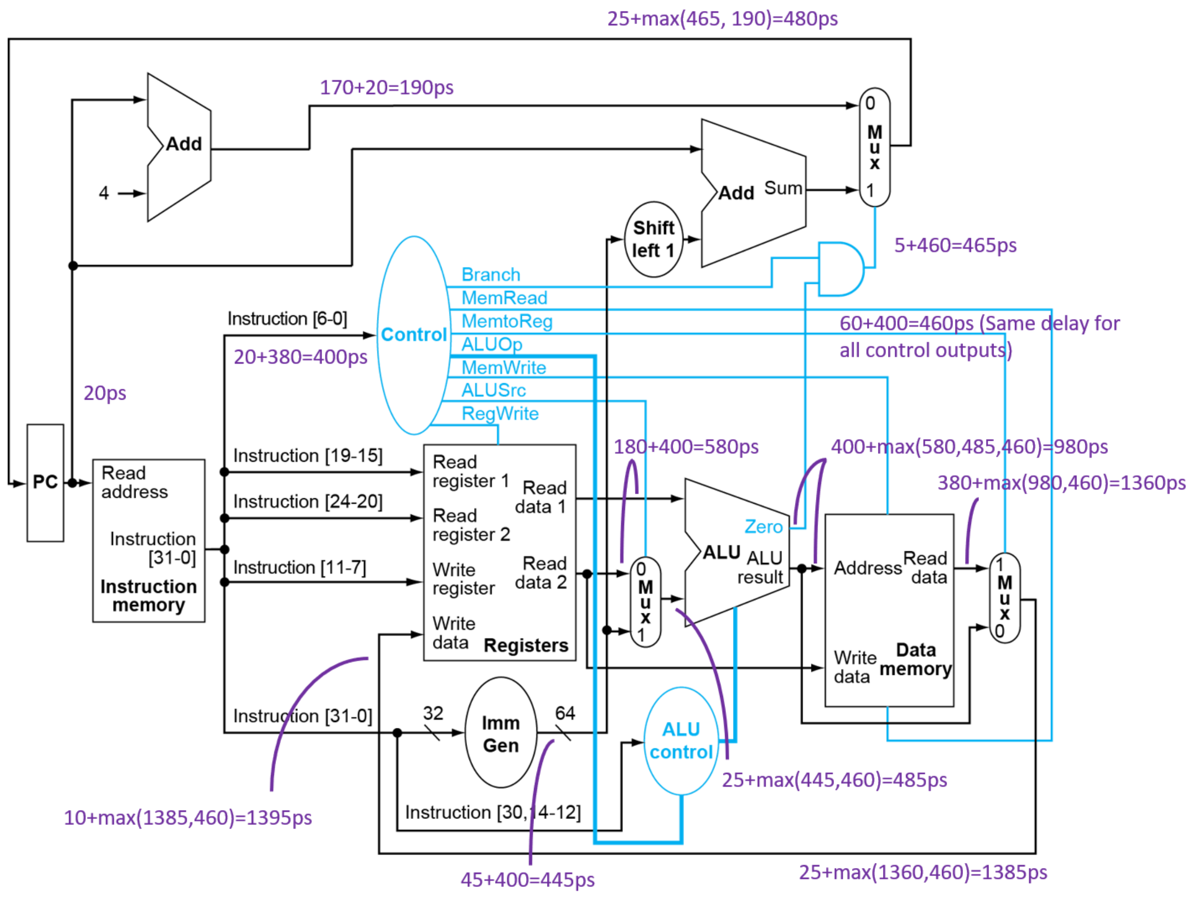

Fig. 2 - ALU with Multiply Function

Clock Cycle with improvement: > 1395 ps

b)

CPU Time Old = 1195 ps * 1xInstructionCount
CPU Time New = 1395 ps * 0.85xInstructionCount

$$\frac{CPU\ Time\ Old}{CPU\ Time\ New} = \text{Speedup: } \frac{1195\ ps * 1xInstructionCount}{1395\ ps * 0.85xInstructionCount} = 1.0078\text{x speedup}$$

c)

$$\frac{1195\ ps * 1xInstructionCount}{[clock\ speed\ for\ 1x\ speedup] * 0.85xInstructionCount} = 1\text{xspeedup}$$

Clock speed for 1x speedup = 1405.88 ps

Since we want improved performance, we want the clock speed to be less than the result above (greater than 1x speedup).

We then subtract all the latencies in our critical path except for our ALU to find the max latency our ALU can have.

ALU Latency < 1405ps – (1195 ps – 200 ps)

Answer: ALU Latency < 410 ps

Q2. [40 Points] What is the final value in registers `a0` and `a1`? What does this function do? (show your work by showing the intermediate values in `a0`, `a1`, and `x1/ra`)

(hint: first figure out what IF does, then track the values in each step for CF...)

```
main:
   li a0, 2
   li a1, 3
   li a2, 4
   li a3, 5
   jal ra, CF
   ret
IF:
   li t0, 32
   li t3, 0
 start:
   mv   t1, a1
   andi t1, t1, 1
   beq  t1, x0, shift
   add  t3, t3, a0

 shift:
   slli a0, a0, 1
   srai a1, a1, 1
   addi t0, t0, -1
   bnez t0, start
   mv   a0, t3
   jr x1

CF:
   addi sp, sp, -28
   sw x0, 24(sp)
   sw x0, 20(sp)
   sw ra, 16(sp)
   sw a0, 12(sp)
   sw a1, 8(sp)
   sw a2, 4(sp)
   sw a3, 0(sp)

   # Step 1
   mv a1, a2
```

```
jal ra, IF
sw a0, 20(sp)

# Step 2
lw  a0, 8(sp)
lw  a1, 0(sp)
jal ra, IF

# Step 3
lw  t0, 20(sp)
sub t2, t0, a0
sw  t2, 20(sp)

# Step 4
lw  a0, 12(sp)
lw  a1, 0(sp)
jal ra, IF
sw a0, 24(sp)

# Step 5
lw a0, 8(sp)
lw a1, 4(sp)
jal ra, IF
mv a1, a0
# Step 6
lw  t0, 24(sp)
add a1, t0, a1
lw  a0, 20(sp)
lw   ra, 16(sp)
addi sp, sp, 28
ret
```

Question 2 Answer)

The functionality of this assembly code is a multiplier (specifically the "IF" portion).

Take this example of binary multiplication by hand:

```
    111
   x101
 ------
    111
   000
  111
 ------
 100011
```

Fig. 3: Multiplication by hand

We take each bit in our multiplier (101), and multiply it by the multiplicand (111), with each new intermediate result shifted to the left one extra position (multiply by 2). After we have multiplied each bit in the multiplier by the multiplicand, we add all of our intermediate results together to get our product. This is why a shifter and an adder make a multiplier.

Fig 4 shows a flowchart for the multiplication algorithm. In each cycle, we check if the lowest bit of the multiplier = 1 (bitwise "and" with "1"). If it is, then we add to the product the multiplicand (since 1 times a number = the number itself) before we shift the multiplicand left by 1 (to shift each intermediate result to the left by one position, a multiplication by 2), shift the multiplier right by 1 (so we can check the next bit in the next cycle), and decrement the counter. If the counter = 0, we stop, else we process the next bit of the multiplier and restart the cycle.

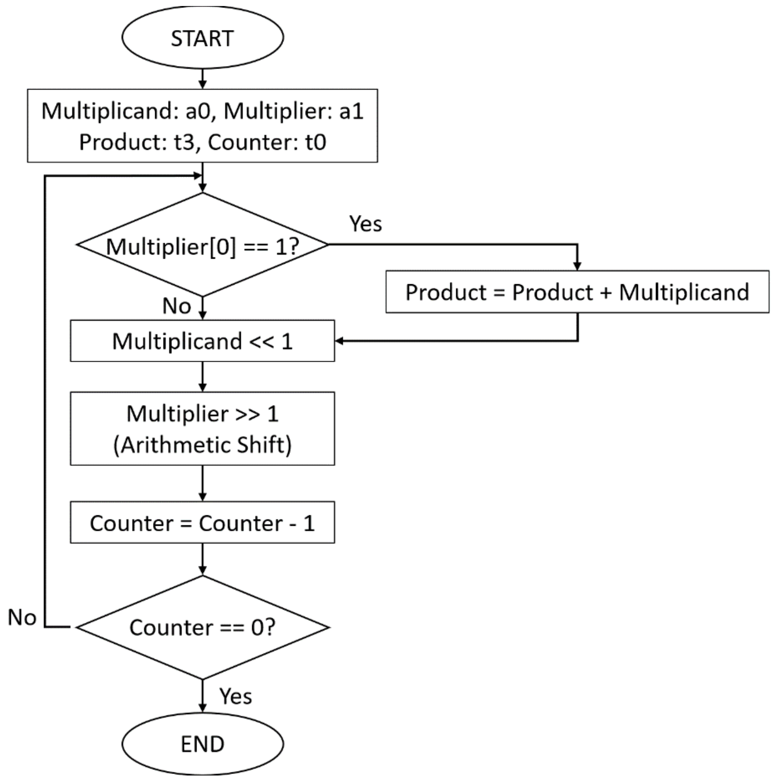

Fig. 4: Multiplication Algorithm

In our code, our variables are initialized as such:

Multiplicand = a0 = 2
Multiplier = a1 = 4
Product = t3 = 0
Counter = t0 = 32

<u>Step 1</u>: with a1 = 4 and a0 = 2, we apply multiplication. Each iteration, we shift a0 to the left by 1 (multiply by 2), and a1 to the right by 1. After 2 loops have passed (a0 has changed from 2 to 8, and a1 now has a 1 in the lowest bit position), we add 8 to our product, t3. Once we have completed this loop, there is nothing more to multiply (a1 = 0), so we store our product (8) into a0. ra points to "sw a0, 20(sp) in Step 1.

(Note: t1 is used as a temporary value for a1. We apply andi to this result, which thus overwrites the value of t1 with the andi result. Thus, we apply the srai to a1, and store the updated value of a1 back into t1 at the start of the next cycle).

Step 2: a0 is set to 3 (value at 8(sp)) and a1 is set to 5 (value at 0(sp))). Thus, our final result is a0 = 15, and a1 = 0. ra points to "lw t0, 20(sp) in Step 3.

Step 3: t0 is set to 8 (value at 20(sp), which was set in Step 1 – this was our Step 1 result). We do t0 – a0, and since a0 = 15 (from Step 2), we store 8 – 15 = -7 into 20(sp)

Step 4: a0 is set to 2 (value at 12(sp)), and a1 is set to 5 (value at 0(sp)). Thus, our final result is a0 = 10, and a1 = 0. ra points to "sw a0, 24(sp)" in Step 4.

Step 5: a0 is set to 3 (value at 8(sp)), and a1 is set to 4 (value at 4(sp)). Our product is 12. At the end of Step 5, we set a1 = a0. Thus, our final result is a0 = 12, and a1 = 12. ra points to "mv a1, a0" in Step 5.

Step 6: t0 is set to 10 (value at 24(sp), which was set in Step 4 – this was our Step 4 result). We add t0 (10) and a1 (12) into a1, so a1 = 22. We set a0 to -7 (value at 20(sp), which was set in Step 3 – this was our Step 3 result). We set ra to the original ra at the start of "Main", pointing to "ret". We add 28 back to the step pointer and end execution.

Final Result: a0 = -7, a1 = 22.

Notes:

This link contains what the assembly could look like in code:
https://gist.github.com/LDelhez/fd6304d04db1e9dabacd65e6108c532c

You can also use an online RISC-V interpreter (such as this:
https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/) to help understand what is happening at each step.

Q3. [20 points] State true (T) or false (F) for each of the following statements (if the statement is False, explain why):

a) A given C program can be compiled into different binaries using different ISAs.
b) According to Dennard's scaling law, the frequency of a transistor/circuit is increased by about 30% from one generation to the next.
c) According to Amdahl's law, if 25% of a program is sequential (non-parallelizable), then the maximum speed up that can be achieved for this program is 1.33 (show your work).
d) For a processor, P1, that has a 3 GHz clock rate and a CPI of 1.5 and executes a program in 1 millisecond, the IC is equal to 20 million (show your work).

Question 3 Answer)

a)

True. RISC-V ISA is just one type of ISA. Other ISA's such as ARM have different ways to encode assembly instructions into machine language (for example, the opcode location in RISC-V is at the lower bits of an instruction, while the opcode location in ARM is at the higher bits of an instruction). Thus, the resulting binaries will look different in different ISA's, even for the same C program.

b)

False. Dennard's scaling law states that the frequency of a transistor/circuit is increased by about 40% from one generation to the next.

c)

Speed Up = $\frac{1}{(1-p) + p/s}$, with p being the percentage of parallelizable content, and s being the factor for improvement (eg: more cores for parallelization).

Since 25% of this program is sequential, 75% is parallelizable, so p = 0.75.

s is the speed up gained by the newly parallelizable content. For example, s=1.6 if the number of cores went from 5 to 8 (1.6 times as fast). Let's plug in s=1.6 into our equation to test what the speedup we get is, with p = 0.75:

Speed Up = $\frac{1}{(1-0.75) + 0.75/1.6}$= 1.39

We can actually obtain a speedup > 1.33 with this percentage of sequential and parallelizable, as long as s > 1.5 (with s = 1.5, speed up = 1.33)

False, speedup of greater than 1.33x can be obtained

d)

Iron Law of Processor Performance:

CPU Time = InstructionCount * CPI * Cycle Time

1 ms = InstructionCount * 1.5 * $\frac{1}{3\ GHz}$

0.001 s = InstructionCount * 1.5 * $\frac{1}{3e9\ Hz}$

InstructionCount = 2 million

False, the instruction count for this setup is 2 million instructions, not 20 million instructions.