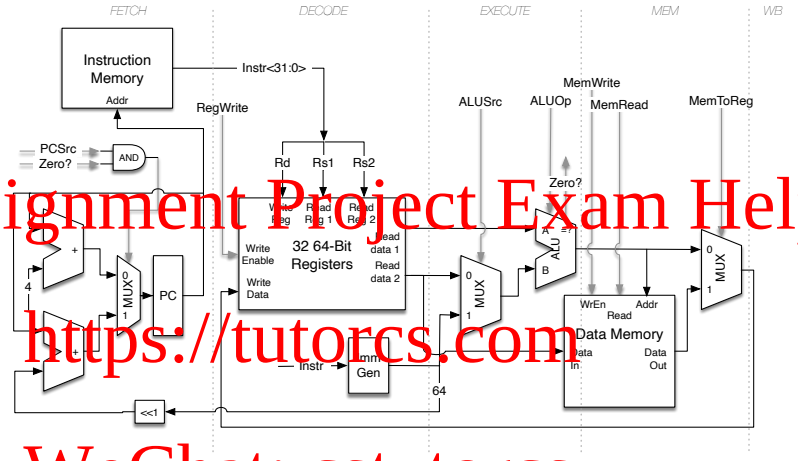
UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering

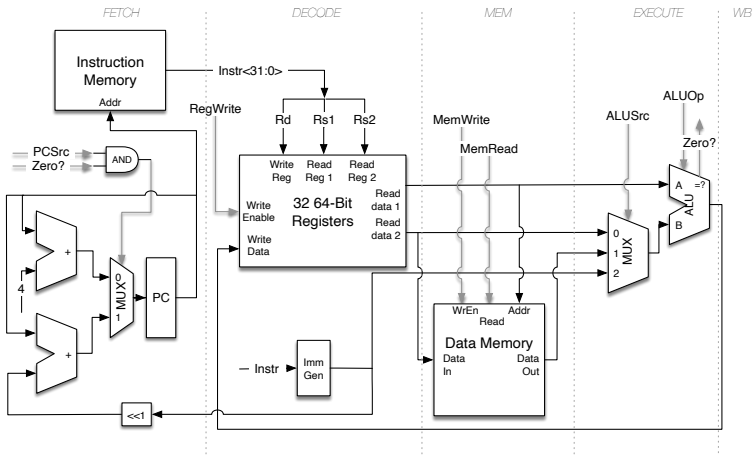
EEC 170 Introduction to Computer Architecture Winter 2021

Quiz 4 Solutions

1. Following is the "baseline" class datapath that will be used in Question 1. This is identical to the diagram we have used in lecture, although some elements have slightly changed positions compared to the lecture diagram. Black lines are datapath signals, gray lines are control signals.



Screech hates load-use hazards. He changes the pipelined baseline design to the one below, putting the data memory before the ALU. All control signals are the same except `MemToReg` has disappeared (its functionality is now subsumed by a more capable `ALUSrc`).



(a) (4 points) Specify a pair of RISC-V instructions that:

- Can be run on either datapath;
- Do *not* require a stall cycle between them on the baseline; and datapath
- *Do* require a stall cycle between them on Screech's datapath.

Solution: We want a pair of instructions that has the use-load hazard, where the first instruction produces a value in its ALU stage and the second instruction consumes that value in its memory stage. For instance:

```
add x2, x3, x4
lw  x5, 0(x2)      # offset must be 0 to run on Screech's datapath
```

(b) (4 points) Specify a single RISC-V instruction that *can* run on the baseline datapath but *cannot* run on Screech's datapath.

Solution: Screech's datapath cannot add a register plus an immediate offset and use the result as the address input to a memory operation, e.g., `ld x2, 100(x3)`.

(c) (4 points) **Write a RISC-V sequence of instructions that *can* run on Screech's datapath that implements the instruction you specified in the previous part.** Use `x10` as a temporary if necessary.

If your answer in the previous part is wrong but your sequence implements your previous answer correctly and can run on Screech's datapath, we will give up to half credit on this part.

Solution: We need to replace a memory operation that uses base+offset addressing with an add to compute the address then the memory operation, for instance:

```
addi x10, x3, 100
ld   x2, 0(x10)
```

(d) (4 points) Specify a pair of RISC-V instructions that require two instructions to run on the baseline datapath but can be run with a single instruction on Screech's datapath.

Solution: Here we're looking for an instruction that uses a dataflow that is only available on Screech's datapath, which will be a load followed by a use of that load. The load needs to be from a register, not register+offset, because

register+offset addressing is not available on Screech's datapath. For instance, the following two RISC-V instructions could be combined into one instruction on Screech's datapath:

```
ld  x2, 0(x3)
add x5, x4, x2
```

2. Your team of two junior designers, Zack and A.C., of the new RISC-VI processor meets with you with some bad news. "We can't meet the clock target," they say, "the memory stage of our 5-stage pipeline just isn't fast enough. We both have a different suggestion."

   You will evaluate each of their solutions on the following code benchmark. You will be counting total clock cycles, *not* the total number of instructions/stalls. Note: When you count clock cycles on this problem, count from the beginning of the first instruction (instruction fetch stage) to the completion (writeback stage) of the last instruction. All instructions issue, execute, and complete in order.

   For maximum opportunity for partial credit, be clear in your solution where stalls occur.


```
ld  x2, 0(x10)
add x2, x2, x2
ld  x3, 8(x10)
ld  x4, 16(x10)
add x3, x3, x4
add x5, x2, x3
```


   (a) (4 points) Assuming we ran the benchmark on the baseline 5-stage RISC-V implementation with all forwarding paths enabled, **how many cycles would it take?** Remember to include any necessary stalls.

Solution: In all solutions on this question, the solution shows ** for stalls and then, if appropriate, delays the beginning of the next instruction to allow it to run stall-free and in order. This could be accounted for in other ways (e.g., propagating the stall to future instructions) but the answer is the same each way.

```
#                1  2  3  4  5  6  7  8  9 10 11 12
ld  x2, 0(x10)   IF ID EX ME WB
add x2, x2, x2      IF ID ** EX ME WB  # load-use, x2
ld  x3, 8(x10)            IF ID EX ME WB
ld  x4, 16(x10)              IF ID EX ME WB
```

```
add x3, x3, x4                    IF ID ** EX ME WB  # load-use, x4
add x5, x2, x3                       IF ID EX ME WB
```

12 cycles. The only two stalls here are due to load-use hazards.

(b) (6 points) "I can make the clock target," says Zack, "and implement all possible forwarding paths, but now on all instructions we require three memory stages (`IF ID EX MEM1 MEM2 MEM3 WB`). Results from memory operations would only now be available for forwarding at the end of `MEM3`." **How many cycles would your benchmark take on Zack's implementation?**

Solution:

```
#                 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
ld  x2, 0(x10)    IF ID EX M1 M2 M3 WB
add x2, x2, x2       IF ID ** ** ** EX M1 M2 M3 WB  # load-use, x2
ld  x3, 8(x10)          IF ID ** ** ** EX M1 M2 M3 WB
ld  x4, 16(x10)            IF ID ** ** ** EX M1 M2 M3 WB
add x3, x3, x4                IF ID ** ** ** EX M1 M2 M3 WB # l-u x4
add x5, x2, x3                   IF ID ** ** ** EX M1 M2 M3 WB
```


18 cycles. The effect here is that a load-use hazard now takes 3 stall cycles to resolve.

(c) (6 points) "I can make the clock target," says A.C., "and keep a 5-stage pipeline, but I have to disable any forwarding paths. Any results that would normally be forwarded instead must be written to the register file. We can write to the register file then read from it on the same stage." This pipeline has no additional capabilities or changes to the standard 5-stage pipeline; it simply has forwarding disabled. **How many cycles would your benchmark take on A.C.'s implementation?**

Solution:

```
#                 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
ld  x2, 0(x10)    IF ID EX ME WB
add x2, x2, x2       IF ** ** ID EX ME WB  # x2
ld  x3, 8(x10)             IF ID EX ME WB
ld  x4, 16(x10)               IF ID EX ME WB
add x3, x3, x4                   IF ** ** ID EX ME WB  # x4
add x5, x2, x3                         IF ** ** ID EX ME WB  # x3
```

16 cycles. The effect here is that any data hazard does not resolve until the writeback stage.