Quiz 2 - Solution

Fall 2021

(Additional tables, information, notes are available on pages 3-5.)

Q1. Assume that we have the following sequence of instructions in our RISCV pipelined processor:

```
begin:
     addi x1, x0, 15
     ori  x2, x0, 2
     andi x3, x1, 3
     sub  x4, x1, x0
     lw   x5, 4(x0)
     add  x5, x5, x3
     beq  x5, x4, end   # assume that this should be taken
     ori  x3, x1, 7
     addi x5, x5, 1
end:
     mul x0,x0,x0
```

Answer the following questions:

a. If our processor has no forwarding and/or branch prediction (i.e., branches are resolved at MEM stage, and we should stall the processor until it is resolved), fill the following table. Write (`f,d,e,m,w`) for each column. If the processor has been stalled, show it by " - " (first two instructions are filled already).

List of potential hazards:

- `andi` *uses the value of* `x1` *produced by* `addi`*: The decode stage of* `andi` *must occur during or after the writeback stage of* `addi`*.*
- `add` *uses the value of* `x5` *produced by* `lw`*: The decode stage of* `add` *must occur during or after the writeback stage of* `lw`*.*
- `beq` *uses the value of* `x5` *produced by* `add`*: The decode stage of* `beq` *must occur during or after the writeback stage of* `add`*.*
- *The address of the instruction to execute after* `beq` *is unknown until* `beq` *moves from the memory stage to the writeback stage. In the meantime, new instructions cannot be fetched.*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
addi	f	d	e	m	w															
ori		f	d	e	m	w														
andi			f	d	-	e	m	w												
sub				f	-	d	e	m	w											
lw						f	d	e	m	w										
add							f	d	-	-	e	m	w							
beq								f	-	-	d	-	-	e	m	w				
mul													-	-	-	f	d	e	m	w

Note that when stalling, the instruction that was in the decode stage effectively stays in that stage. The decode stage is repeated until the operands are ready, which resolves the data hazard. For instance, even though the pipeline diagram suggests otherwise at first glance, the `andi` *instruction is in the decode stage during the fifth clock cycle, and therefore reads the correct value of* `x1` *written at the same time by the* `addi` *instruction.*

The following pipeline diagram may be considered acceptable too, as it shows the correct begin/end cycle of each instruction.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
addi	f	d	e	m	w															
Ori		f	d	e	m	w														
andi			-	f	d	e	m	w												
sub					f	d	e	m	w											
lw						f	d	e	m	w										
add							-	-	f	d	e	m	w							
beq										-	-	f	d	e	m	w				
mul													-	-	-	f	d	e	m	w

However, this second diagram does not reflect exactly the flow of instructions in the pipeline: in reality, instructions need to be fetched and decoded before a data hazard can be detected, and before a stall can be generated.

b. Now assume that we have forwarding and the branch outcome can be known at DECODE stage. Fill the following table: (you still have to stall for branches!)
If there is a forwarding, show it with * (for BOTH the forwarding stage and EX)

For RAW dependencies between two consecutive instructions (`add` to `beq`), data is forwarded from the memory stage of the producing instruction to the execute stage of the consuming instruction.

For RAW dependencies between instructions that are separated by a single instruction (`addi` to `andi`), data is forwarded from the writeback stage of the producing instruction to the execute stage of the consuming instruction.

*For RAW dependencies between a load instruction and the next instruction (`lw` to `add`), data **cannot** be forwarded from the memory stage of the load to the execute stage of the consuming instruction because the memory read only completes at the end of the cycle. Instead, the pipeline needs to stall for one cycle. Data is then forwarded from the writeback stage of the load to the execute stage of the consuming instruction.*

The address of the instruction to execute after `beq` is unknown until `beq` moves from the decode stage to the execute stage. In the meantime, new instructions cannot be fetched.

The correct pipeline diagram is thus the following:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
addi	f	d	e	m	*w											
ori		f	d	e	m	w										
andi			f	d	*e	m	w									
sub				f	d	e	m	w								
lw					f	d	e	m	*w							
add						f	d	-	*e	*m	w					
beq							f	-	d	*e	m	w				
mul									-	f	d	e	m	w		

Or, by abuse of notation:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
addi	f	d	e	m	*w											
ori		f	d	e	m	w										
andi			f	d	*e	m	w									
sub				f	d	e	m	w								
lw					f	d	e	m	*w							
add						-	f	d	*e	*m	w					
beq								f	d	*e	m	w				
mul									-	f	d	e	m	w		

c. What is the speed up in case b (compared to a)?

The execution time in case a is $T_a = 20$ cycles.
The execution time in case b is $T_b = 14$ cycles.

The speedup is thus

$$\frac{T_a}{T_b} = \frac{20 \text{ cycles}}{14 \text{ cycles}} = \frac{10}{7} \approx 1.43$$

Note that, since case b is faster than case a, the speedup is greater than 1.

Q2. Assume our program has this mix of instructions (there is no RAW or load-use data hazard):

R/I- type	BEQ	JAL	LW/SW
60%	20%	5%	15%

a. What is the IPC if we have a branch predictor that does *always-not-taken* with 30% accuracy but with ***no BTB***? (Assume that the branch resolves at DECODE stage).

The execution of R-type, I-type, and LW/SW instructions is not impacted by control hazards. As there are no data hazards, we have

$$CPI_{R/I} = CPI_{lw/sw} = 1$$

Correctly predicted beq instructions account for $20\% \times 30\% = 6\%$ *of all instructions. As they are correctly predicted, they do not cause any time penalty:*

$$CPI_{beq\_good} = 1$$

Incorrectly predicted instructions account for $20\% \times 70\% = 14\%$ *of all instructions. When they execute, one instruction is fetched speculatively and then flushed. There is a one clock cycle penalty:*

$$CPI_{beq\_bad} = 2$$

For jump-and-link instructions, the processor cannot predict the target without a BTB. It needs to stall until the target is known, at the end of the decode stage:

$$CPI_{jal} = 2$$

The average CPI is

$$CPI_a = 60\% \times CPI_{R/i} + 15\% \times CPI_{lw/sw} + 6\% \times CPI_{beq\_good} + 14\% \times CPI_{beq\_bad} + 5\% \times CPI_{jal} = 1.19$$

The average IPC is thus

$$IPC_a = \frac{1}{CPI_a} = \frac{1}{1.19} \approx 0.84$$

b. If we change the branch predictor to a 2-bit branch history table (BHT) with 80% accuracy and add a (perfect) BTB, what would be the new IPC?

With those changes, correctly predicted branches would account for $20\% \times 80\% = 16\%$ *of all instructions. Incorrectly predicted branches would account for* $20\% \times 20\% = 4\%$ *of all instructions.*

With a perfect BTB, the target of jump-and-link instructions is known at the end of the fetch stage, so the next correct instruction can be fetched at the next cycle:

$$CPI_{jal} = 1$$

The average CPI becomes

$$CPI_b = 60\% \times CPI_{R/i} + 15\% \times CPI_{lw/sw} + 16\% \times CPI_{beq\_good} + 4\% \times CPI_{beq\_bad} + 5\% \times CPI_{jal} = 1.04$$

The average IPC is then

$$IPC_b = \frac{1}{CPI_b} = \frac{1}{1.04} \approx 0.96$$

c. What is the speed up for case b?

From the Iron law of computing, we know that

$$speedup = \frac{\#instructions \times CPI_a \times T_{cycle}}{\#instructions \times CPI_b \times T_{cycle}}$$

Since the number of instructions and the clock period are assumed to be the same in case a and case b, we have

$$speedup = \frac{CPI_a}{CPI_b} = \frac{1.19}{1.04} \approx 1.14$$

Q3- State True (T) or False (F) for each statement. Show your work/explain for each statement.

a. BTB is useful for finding the destination address in RET and CALL instructions.

False. The BTB is not used to find the target of ret instructions. They are obtained using a separate structure, known as the return stack buffer.

(Note that it is technically possible to design a processor that does not have a return stack buffer and that uses the BTB to predict the target of ret instructions. Although the BTB would then often produce incorrect predictions, it may occasionally be correct. In that case, the statement would be true.)

b. Ideally, an N-Issue superscalar processor can improve the speed up by a factor of N.

True. (Assuming that the clock frequency and the CPI of each individual instruction are unchanged.)

c. For a 2-bit predictor with initial state 00 (Not Taken), the `NT, T, T, NT, T, T, T, NT, NT, NT` branch pattern, the predictor has 80% accuracy.

False.

NT: predict not taken, which is correct, state stays equal to 00.
T: predict not taken, which is incorrect, state is then updated to 01.
T: predict not taken, which is incorrect, state is then updated to 10.
NT: predict taken, which is incorrect, state is then updated to 01.
T: predict not taken, which is incorrect, state is then updated to 10.
T: predict taken, which is correct, state is then updated to 11.
T: predict taken, which is correct, state stays equal to 11.
NT: predict taken, which is incorrect, state is then updated to 10.
NT: predict taken, which is incorrect, state is then updated to 01.
NT: predict not taken, which is correct, state is then updated to 00.

The accuracy is 40%, which is lower than 80%.

d. Processor A which has a branch predictor that has 95% accuracy with 2 cycles miss penalty has a higher (average) IPC than Processor B which has a branch predictor with 80% accuracy with 1 cycle miss penalty.

True.

The average CPI of branch instructions on A and B are

$$CPI_{A,branch} = 95\% \times 1 + 5\% \times (1 + 2) = 1.1$$
$$CPI_{B,branch} = 80\% \times 1 + 20\% \times (1 + 1) = 1.2$$

Let α be the fraction of instructions that are branches.

If A and B have the same CPI for non-branch instructions, the average CPI of A and B are

$$CPI_A = (1 - \alpha) \times CPI_{non\ branch} + \alpha \times CPI_{A,branch} = (1 - \alpha) \times CPI_{non\ branch} + 1.1\alpha$$
$$CPI_B = (1 - \alpha) \times CPI_{non\ branch} + \alpha \times CPI_{B,branch} = (1 - \alpha) \times CPI_{non\ branch} + 1.2\alpha$$

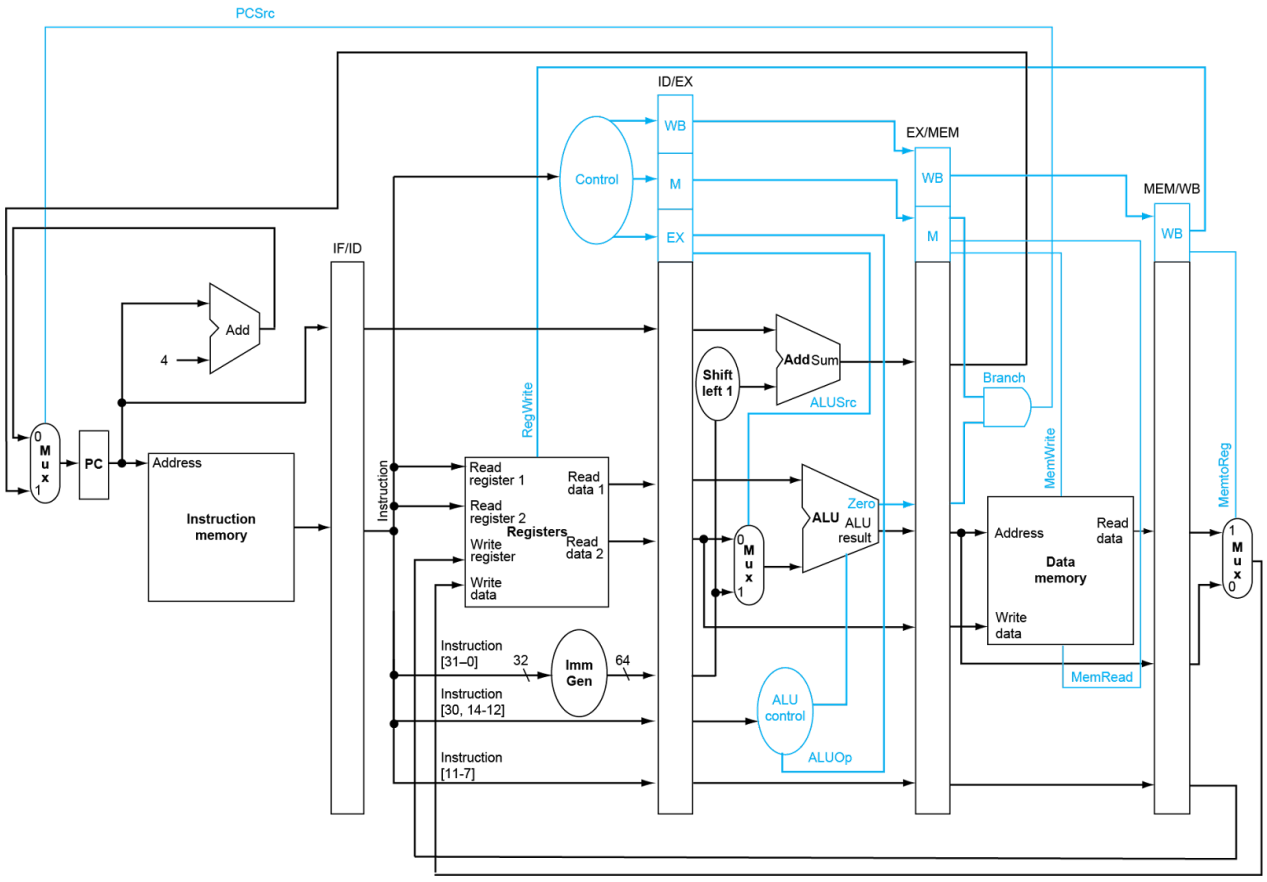
Processor A has a higher IPC if and only if

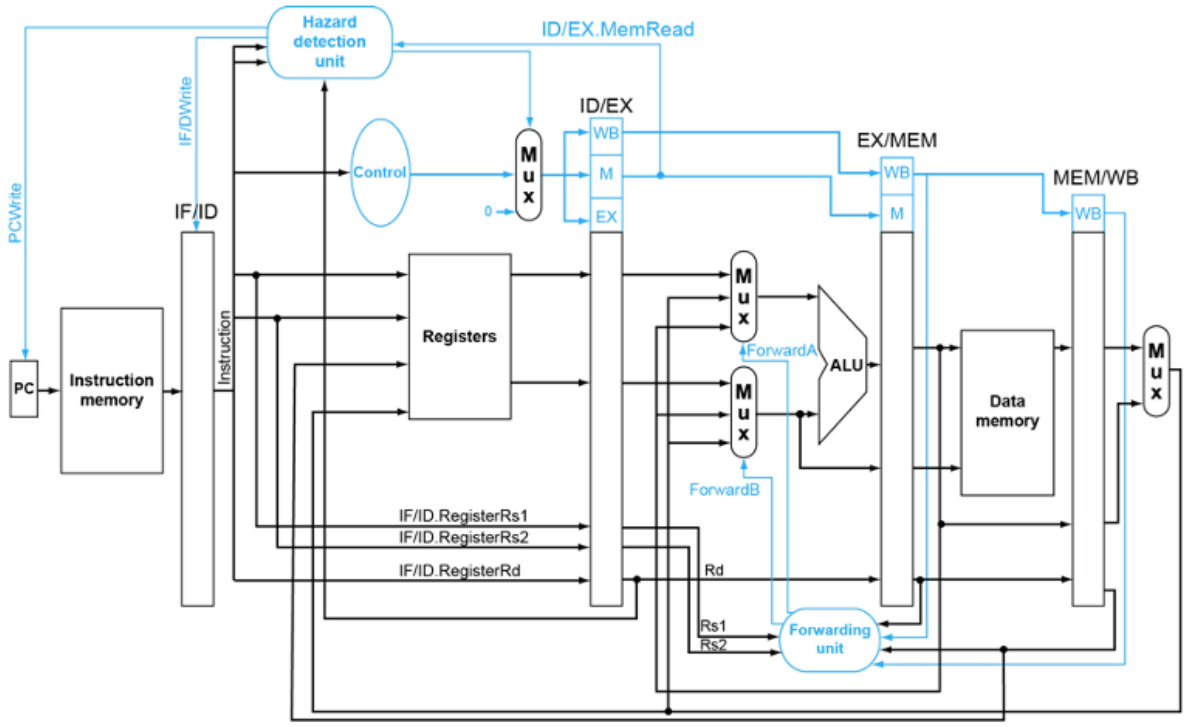
$$CPI_A < CPI_B$$
$$\Leftrightarrow (1 - \alpha) \times CPI_{non\ branch} + 1.1\alpha < (1 - \alpha) \times CPI_{non\ branch} + 1.2\alpha$$
$$\Leftrightarrow 1.1\alpha < 1.2\alpha$$

which is true for all $\alpha > 0$.

Appendix:

You can use the following information to answer the questions. You are NOT allowed to use any other resources (e.g., Google, notes, lectures, Campuswire, etc.). Using Calculator is OK.





ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <s constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <u constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, constant`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « constant`
SRLI	`srli rd, rs1, constant`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »u constant`
SRAI	`srai rd, rs1, constant`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »s constant`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <s reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <u reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »u reg[rs2]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »s reg[rs2]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

Loads	Load Byte	I	LB	rd,rs1,imm	rd = M[rs1+imm][0:7]
	Load Halfword	I	LH	rd,rs1,imm	rd = M[rs1+imm][0:15]
	Load Word	I	LW	rd,rs1,imm	rd = M[rs1+imm][0:31]
	Load Byte Unsigned	I	LBU	rd,rs1,imm	rd = M[rs1+imm][0:7] zero-extends
	Load Half Unsigned	I	LHU	rd,rs1,imm	rd = M[rs1+imm][0:15] zero-extends
Stores	Store Byte	S	SB	rs1,rs2,imm	M[rs1+imm][0:7] = rs2[0:7]
	Store Halfword	S	SH	rs1,rs2,imm	M[rs1+imm][0:15] = rs2[0:15]
	Store Word	S	SW	rs1,rs2,imm	M[rs1+imm][0:31] = rs2[0:31]

JAL	`jal rd, label`	Jump and Link	`reg[rd] <= pc + 4` `pc <= label`
JALR	`jalr rd, offset(rs1)`	Jump and Link Register	`reg[rd] <= pc + 4` `pc <= {(reg[rs1] + offset)[31:1], 1'b0}`
BEQ	`beq rs1, rs2, label`	Branch if $=$	`pc <= (reg[rs1] == reg[rs2]) ? label` `: pc + 4`
BNE	`bne rs1, rs2, label`	Branch if $\neq$	`pc <= (reg[rs1] != reg[rs2]) ? label` `: pc + 4`
BLT	`blt rs1, rs2, label`	Branch if $<$ (Signed)	`pc <= (reg[rs1] <`$_s$ `reg[rs2]) ? label` `: pc + 4`
BGE	`bge rs1, rs2, label`	Branch if $\geq$ (Signed)	`pc <= (reg[rs1] >=`$_s$ `reg[rs2]) ? label` `: pc + 4`
BLTU	`bltu rs1, rs2, label`	Branch if $<$ (Unsigned)	`pc <= (reg[rs1] <`$_u$ `reg[rs2]) ? label` `: pc + 4`
BGEU	`bgeu rs1, rs2, label`	Branch if $\geq$ (Unsigned)	`pc <= (reg[rs1] >=`$_u$ `reg[rs2]) ? label` `: pc + 4`