ECE M116C- CS M151B Computer Architecture Systems - UCLA

Quiz 2

(upload it to Gradescope)

Fall 2021

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

(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).

	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																				
sub																				
lw																				
add																				
beq																				
mul																				

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)

	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																				
sub																				
lw																				
add																				
beq																				
mul																				

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

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%

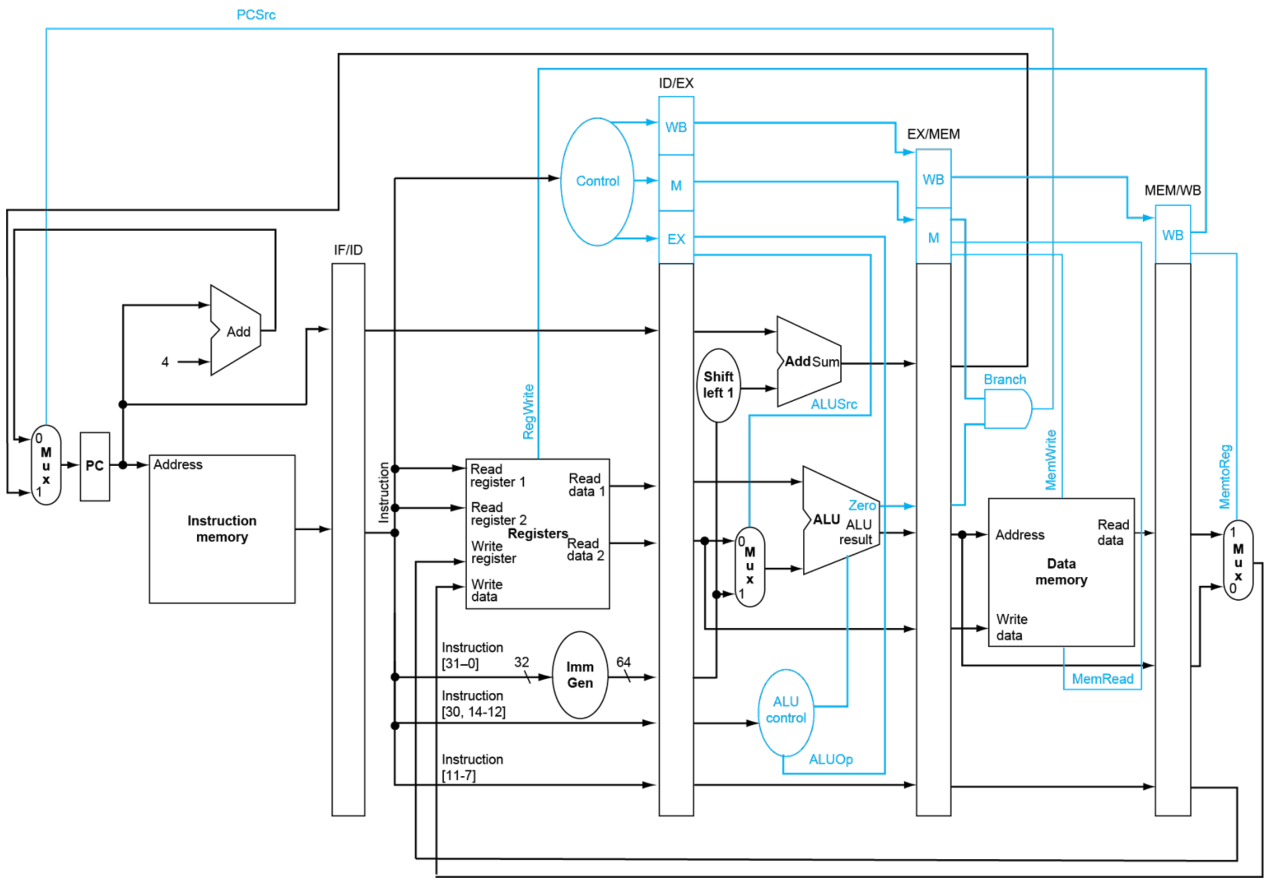
a. What is the IPC if we have a branch predictor that does *always-not-taken* with 30% accuracy but with ***no BTB***? (assume that the branch resolves at DECODE stage).
b. If we change the branch predictor to a 2-bit branch history table (BHT) with 80% accuracy and add a (perfect) BTB, what would be the new IPC?
c. What is the speed up for case b?

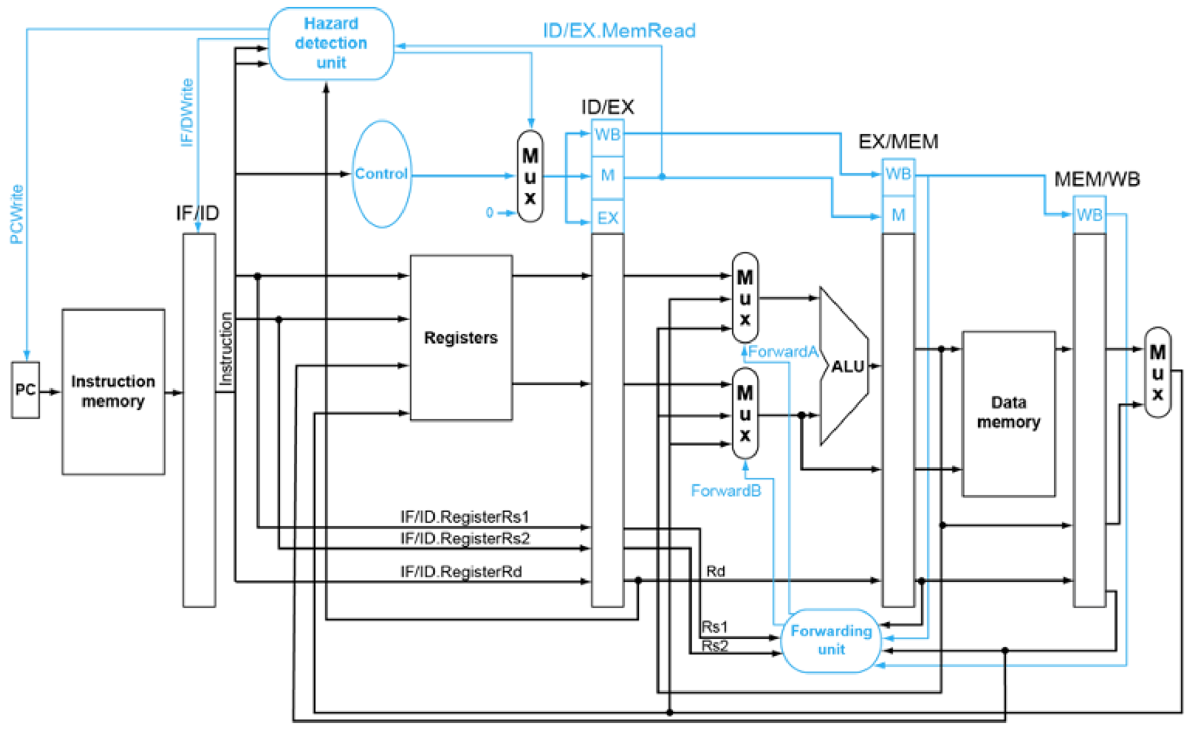
Q3- State True (T) or False (F) for each statement. Show your work/explain for each statement.

a. BTB is useful for finding the destination address in RET and CALL instructions.
b. Ideally, an N-Issue superscalar processor can improve the speed up by a factor of N.
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.
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.

Appendix:

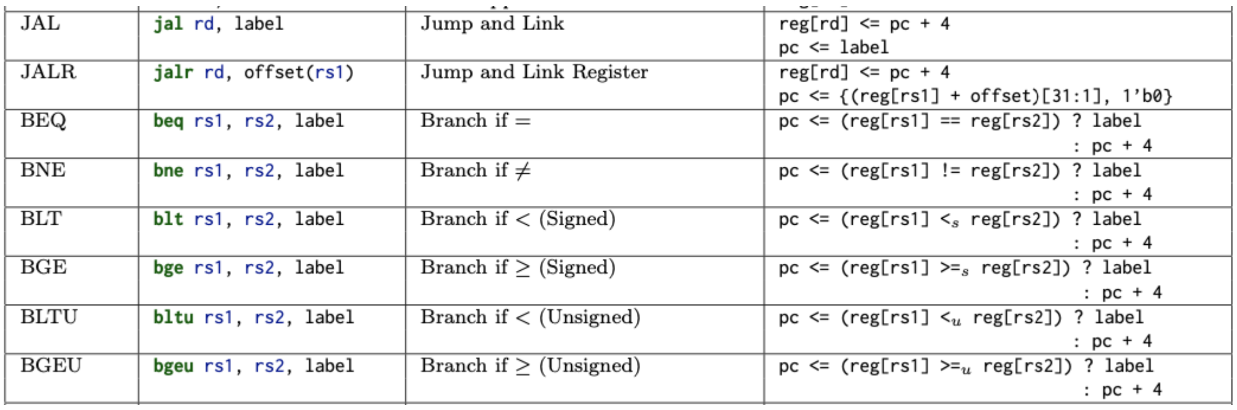
You can use the following information to answer the questions. You are NOT allowed to use any other resources (e.g., Google, notes, lectures, Campuswire, etc.). Using Calculator is OK.





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]` $\gg_u$ `constant`
SRAI	`srai rd, rs1, constant`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1]` $\gg_s$ `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]` $\gg_u$ `reg[rs2]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1]` $\gg_s$ `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