Last updated:
10/6/2021

Problem M7.1: Branch Prediction

This problem will investigate the effects of adding global history bits to a standard branch prediction mechanism. **In this problem assume that the MIPS ISA has no delay slots.**

Throughout this problem we will be working with the following program.

```
loop:
     LW   R4, 0(R3)
     ADDI R3, R3, 4
     SUBI R1, R1, 1
b1:
     BEQZ R4, b2
     ADDI R2, R2, 1
b2:
     BNEZ R1, loop
```

Assume the initial value of R1 is n (n>0).
Assume the initial value of R2 is 0 (R2 holds the result of the program).
Assume the initial value of R3 is p (a pointer to the beginning of an array of 32-bit integers).

All branch prediction schemes in this problem will be based on those covered in the lecture. We will be using a 2-bit predictor state machine, as shown below.

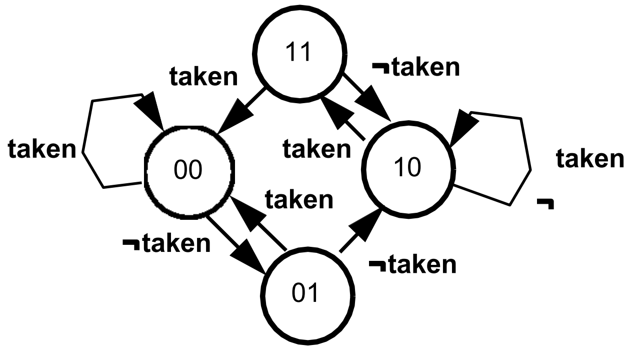


Figure M7.1-A: BP bits state diagram

In state 1X we will guess not taken. In state 0X we will guess taken.

Assume that b1 and b2 do not conflict in the BHT.

Problem M7.1.A Program

What does the program compute? That is, what does R2 contain when we exit the loop?

Problem M7.1.B 2-bit branch prediction

Now we will investigate how well our standard 2-bit branch predictor performs. Assume the inputs to the program are n=8 and p[0] = 1, p[1] = 0, p[2] = 1, p[3] = 0,… etc. That is the array elements exhibit an alternating pattern of 1's and 0's. Fill out Table M7.1-1 (note that the first few lines are filled out for you). What is the number of mispredicts?

Table M7.1-1 contains an entry for every branch (either b1 or b2) that is executed. The Branch Predictor (BP) bits in the table are the bits from the BHT. For each branch, check the corresponding BP bits (indicated by the bold entries in the examples) to make a prediction, then update the BP bits in the following entry (indicated by the italic entries in the examples).

Problem M7.1.C Branch prediction with one global history bit

Now we add a global history bit to the branch predictor, as described in the lecture. Fill out Table M7.1-2, and again give the total number of mispredicts you get when running the program with the same inputs.

Problem M7.1.D Branch prediction with two global history bits

Now we add a second global history bit. Fill out Table M7.1-3. Again, compute the number of mispredicts you get for the same input.

Problem M7.1.E Analysis

Compare your results from problems M7.1.B, M7.1.C and M7.1.D. When do most of the mispredicts occur in each case (at the beginning, periodically, at the end, etc.)? What does this tell you about global history bits in general? For a large n, what prediction scheme will work best? Explain briefly.

Problem M7.1.F Analysis II

The input we worked with in this problem is quite regular. How would you expect things to change if the inputs were random (each array element were equally probable to be 0 or 1). Of the three branch predictors we looked at in this problem, which one will perform best for this type of input? Is your answer the same for large and small n?

What does this tell you about additional history bits: when are they useful and when do they hurt you?

System State		Branch Predictor		Branch Behavior	
PC	R3/R4	b1 bits	b2 bits	Predicted	Actual
b1	4/1	**10**	10	**N**	**N**
b2	4/1	*10*	**10**	**N**	**T**
b1	8/0	**10**	*11*	**N**	**T**
b2	8/0	*11*	**11**	**N**	**T**
b1	12/1	**11**	*00*		
b2	12/1				
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					
b1					
b2					

Table M7.1-1

Last updated:
10/6/2021

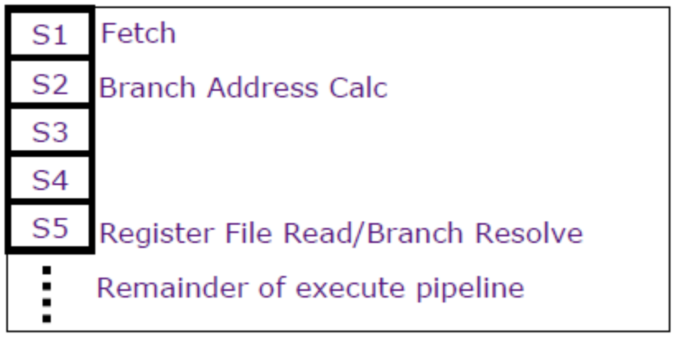
System State			Branch Predictor				Behavior	
PC	R3/R4	history bit	b1 bits		b2 bits			
			set 0	set 1	set 0	set 1	Predicted	Actual
b1	4/1	**1**	10	**10**	10	10	**N**	**N**
b2	4/1	**0**	10	*10*	**10**	10	**N**	**T**
b1	8/0	**1**	10	**10**	*11*	10		
b2	8/0							
b1	12/1							
b2	12/1							
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								
b1								
b2								

Table M7.1-2

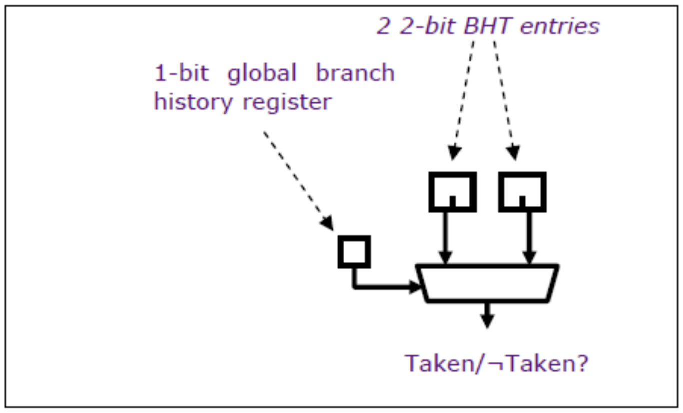
System State			Branch Predictor								Behavior	
PC	R3/R4	history	b1 bits				b2 bits					
		bits	set 00	set 01	set 10	set 11	set 00	set 01	set 10	set 11	Predicted	Actual
b1	4/1	**11**	10	10	10	**10**	10	10	10	10	**N**	**N**
b2	4/1	**01**	10	10	10	*10*	10	**10**	10	10	**N**	**T**
b1	8/0	**10**	10	10	**10**	10	10	*11*	10	10		
b2	8/0											
b1	12/1											
b2	12/1											
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												
b1												
b2												

Table M7.1-3

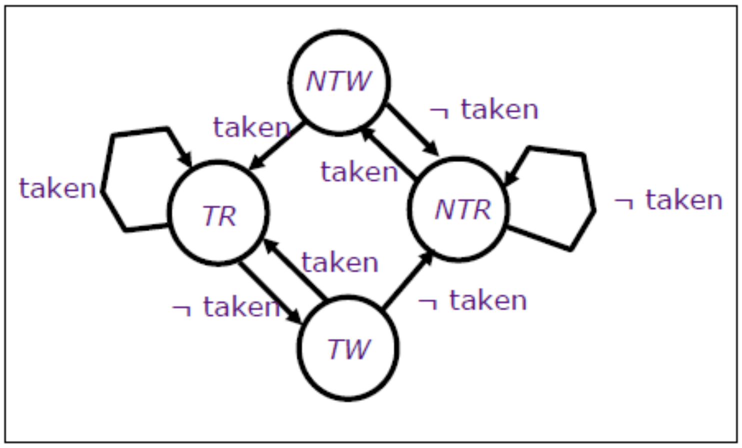
Problem M7.2: Branch Prediction


S1
Fetch
S2
Branch Address Calc
S3
S4
S5
Register File Read/Branch Resolve
Remainder of execute pipeline

Consider a CPU with a pipeline pictured on the right. The first stage of the pipeline fetches the instruction. The **second stage** of the pipeline recognizes branch instructions and **performs branch prediction** using a BHT. If the branch is predicted to be taken, it forwards the decoded target of the branch to the first stage, and kills the instruction in the first stage. The **fifth stage** of the pipeline reads the registers and **resolves the correct target of the branch**. If the branch target was mispredicted, the correct target is forwarded to the first stage, and all instructions in between are killed. The remaining stages finish the computation of the instruction.



The processor uses a **single global history bit** to remember whether the last branch was taken or not. There is only **one line in the BHT**, so the address of the branch instruction is not used for selecting the proper table entry. Each entry in the table is labeled as TW for Take Wrong, TR for Take Right, NTW for do Not Take Wrong and NTR for do Not Take Right, as pictured below. The setup of the BHT predictor is illustrated on the right.



In this question we will study execution of the following loop. This processor has **no** branch delay slots. You should assume that branch at address 1 is **never** taken, and that the branch at address 5 is **always** taken.

Instruction Label	Address	Instruction
LOOP	1	BEQ R2, R5, NEXT
	2	ADD R4, R4, 1
	3	MULT R3, R3, R4
NEXT	4	MULT R2, R2, 3847
	5	BNEZ R4, LOOP
	6	NOP
	7	NOP
	8	NOP
	9	NOP
	10	NOP

You should also **disregard any possible structural hazards**. The processor always runs at full speed, and there are **no pipeline bubbles** (except for those created by the branches).

Last updated:
10/6/2021

Problem M7.2.A

Now we study how well the history bit works, when it is being **updated by the fifth stage** of the processor. The fifth stage also updates the BHT based on the result of a branch. The same BHT entry that was used to make the original prediction is updated.

Please fill in the table below.

You should fetch a new instruction every cycle. You should fill in the *Branch Prediction* and the *Prediction Correct?* columns for branch instructions only (note that the branch prediction actually happens one cycle **after** the instruction is fetched). You should fill in the *Branch Predictor State* columns whenever they are updated. Please **circle the instructions which will be committed**.

The first three committing instructions fetched have been filled in for you. You should enter enough instructions to **add 8 more committing instructions**. You may not need all the rows in the table.

Cycle	Instruction Fetched	Branch Prediction	Prediction Correct?	Branch Predictor State		
				Branch History	Last Branch Taken Predictor	Last Branch Not Taken Predictor
0	-	-		T	TW	TW
1	[✓ 1]	T	N			
2	2					
3	4					
4	5	T	Y			
5	6			NT	NTR	
6	[✓ 2]					
7	[✓ 3]					
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						

Problem M7.2.B

Now we study how well the branch **history bit** works, when it **is being updated speculatively** by the second stage of the processor. If the branch is mispredicted, the fifth stage sets the branch history bit to the correct value. Finally, the fifth stage also updates the BHT based on the result of a branch. The same BHT entry that was used to make the original prediction is updated.

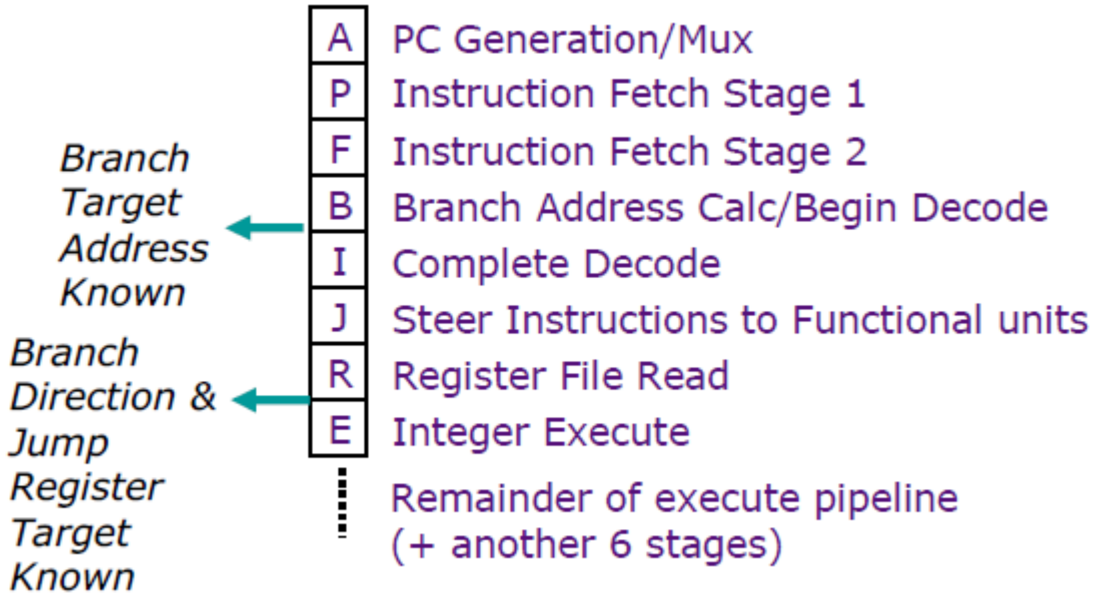
Please fill in the table below. The notation in the table should be same as in ***M7.2.A***.

The first three committing instructions fetched have been filled in for you. You should enter enough instructions to **add 8 more committing instructions**. You may not need all the rows in the table.

Cycle	Instruction Fetched	Branch Prediction	Prediction Correct?	Branch Predictor State		
				Branch History	Last Branch Taken Predictor	Last Branch Not Taken Predictor
0	-	-		T	TW	TW
1	1	T	N			
2	2			T		
3	4					
4	5	T	Y			
5	6			NT	NTR	
6	2					
7	3					
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						

Problem M7.3: Branch Prediction

Consider the fetch pipeline of UltraSparc-III processor. In this part, we evaluate the impact of branch prediction on the processor's performance. There are no branch delay slots.


A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute
Remainder of execute pipeline (+ another 6 stages)
Branch Target Address Known
Branch Direction & Jump Register Target Known

Here is a table to clarify when the direction and the target of a branch/jump is known.

Instruction	Taken known? (At the end of)	Target known? (At the end of)
BEQZ/BNEZ	R	B
J	B (always taken)	B
JR	B (always taken)	R

Problem M7.3.A

As a first step, we add a branch history table (BHT) in the fetch pipeline as shown on the next page. In the B stage (Branch Address Calc/Begin Decode), a conditional branch instruction (BEQZ/BNEZ) looks up the BHT, but an unconditional jump does not. If a branch is predicted to be taken, some of the instructions are flushed and the PC is redirected to the calculated branch target address. The instruction at PC+4 is fetched by default unless PC is redirected by an older instruction.

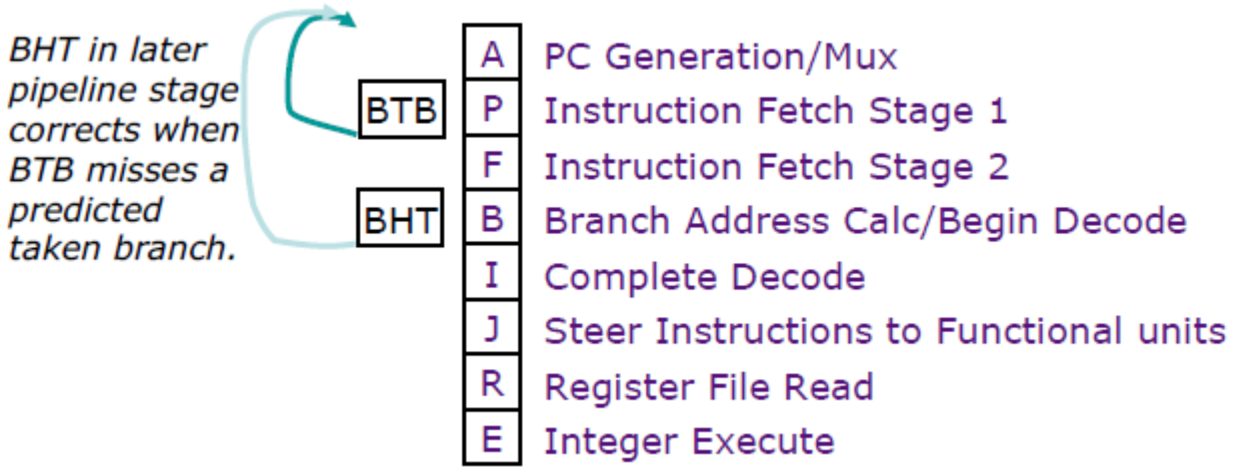
For each of the following cases, write down the number of pipeline bubbles caused by a branch or jump. If there is no bubble, you can simply put 0. (Y = yes, N= no)

	Predicted Taken?	**Actually Taken?**	**Pipeline bubbles**
BEQZ/ BNEZ	Y	Y	
	Y	N	
	N	Y	
	N	N	
J	Always taken (No lookup)	Y	
JR	Always taken (No lookup)	Y	

Problem M7.3.B

To improve the branch performance further, we decide to add a branch target buffer (BTB) as well. Here is a description for the operation of the BTB.

1. The BTB holds entry_PC, target_PC pairs for jumps and branches predicted to be taken. Assume that the target_PC predicted by the BTB is always correct for this question. (Yet the direction still might be wrong.)
2. The BTB is looked up every cycle. If there is a match with the current PC, PC is redirected to the target_PC predicted by the BTB (unless PC is redirected by an older instruction); if not, it is set to PC+4.


BHT in later pipeline stage corrects when BTB misses a predicted taken branch.
BTB
BHT
A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute

Fill out the following table of the number of pipeline bubbles (only for conditional branches).

	BTB Hit?	(BHT) Predicted Taken?	Actually Taken?	Pipeline bubbles
Conditional Branches	Y	Y	Y	
	Y	Y	N	
	Y	N	Y	Cannot occur
	Y	N	N	Cannot occur
	N	Y	Y	
	N	Y	N	
	N	N	Y	
	N	N	N	

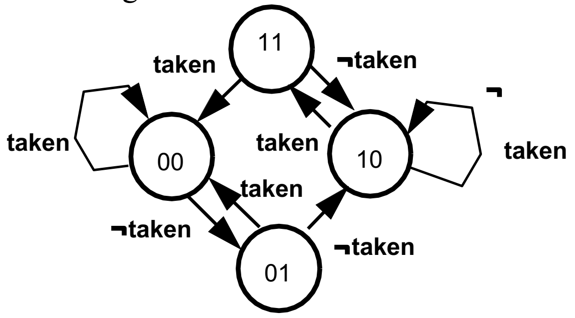
Problem M7.3.C

We will be working on the following program:

```
ADDRESS                INSTRUCTION
 0x1000          BR1:  BEQZ R5, NEXT      ; always taken
 0x1004                ADDI R4, R4, #4
 0x1008                MULT R3, R5, R3
 0x100C                ST   R3, 0(R4)
 0x1010                SUBI R5, R5, #1
 0x1014         NEXT:  ADDI R1, R1, #1
 0x1018                SLTI R2, R1, 100   ; repeat 100 times
 0x101C          BR2:  BNEZ R2, BR1
 0x1020                NOP
 0x1024                NOP
 0x1028                NOP
```

Given a snapshot of the BTB and the BHT states on entry to the loop, fill in the timing diagram for one iteration (plus two instructions) on the next page. (Don't worry about the stages beyond the E stage.) We assume the following for this question.

1. The initial values of R5 and R1 are zero, so `BR1` is always taken.
2. We disregard any possible structural hazards. There are no pipeline bubbles (except for those created by branches.)
3. We fetch only one instruction per cycle.
4. We use a two-bit predictor whose state diagram is shown below. In state 1X we will guess not taken; in state 0X we will guess taken. `BR1` and `BR2` do not conflict in the BHT.



5. We use a two-entry fully-associative BTB with the LRU replacement policy.

Initial Snapshot

(Valid) V	Entry PC	Predicted Target PC
1	0x101C	0x1000

BTB

	...	
BR1	1	1
	...	
BR2	0	0
	...	

BHT

TIME

Address	Instruction	1→	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0x1000	BEQZ R5, NEXT	A	P	F	B	I	J	R	E															
0x1014	ADDI R1, R1, #1																							
0x1018	SLTI R2, R1, 100																							
0x101C	BNEZ R2, LOOP																							
0x1000	BEQZ R5, NEXT																							
0x1014	ADDI R1, R1, #1																							

Timing diagram for M3.4.C

Last updated:
10/6/2021

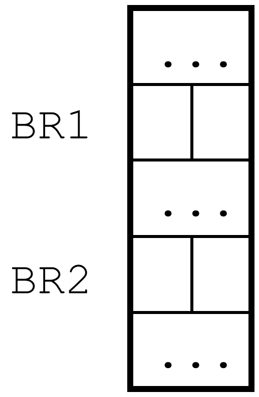
Last updated:
10/6/2021

Problem M7.3.D

What will be the BTB and BHT states right after the 6 instructions in Question 9 have updated the branch predictors' states? Fill in (1) the BTB and (2) the entries corresponding to BR1 and BR2 in the BHT.

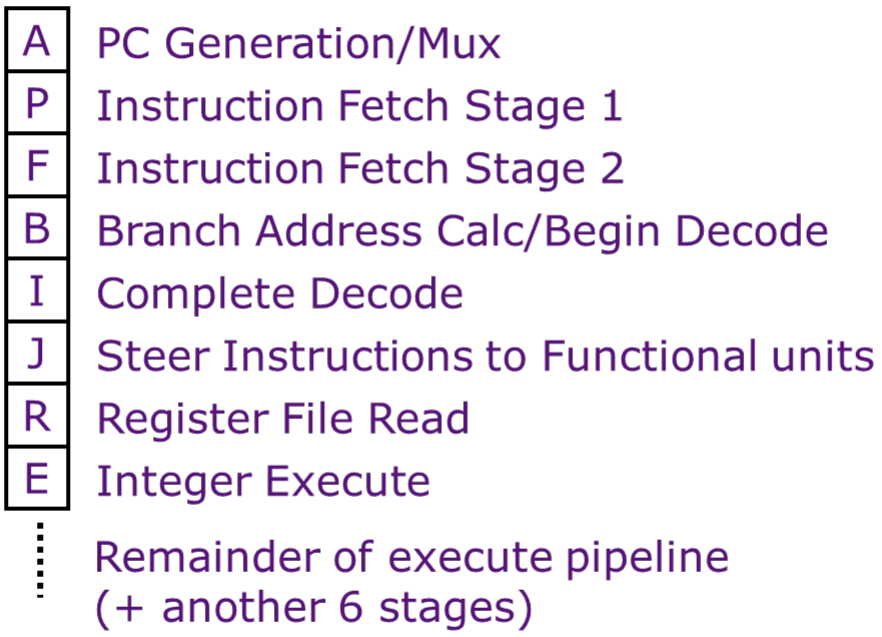
(Valid) V	Entry PC	Predicted Target PC

BTB



BHT

Problem M7.4: Complex Pipelining (Spring 2014 Quiz 2, Part B)


A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute
Remainder of execute pipeline
(+ another 6 stages)

You are designing a processor with the complex pipeline illustrated above. For this problem assume there are no unconditional jumps or jump register—*only* conditional branches.

Suppose the following:

- Each stage takes a single cycle.
- Branch addresses are known after stage Branch Address Calc/Begin Decode.
- Branch conditions (taken/not taken) are known after Register File Read.
- Initially, the processor *always* speculates that the next instruction is at PC+4, without any specialized branch prediction hardware.
- Branches always go through the pipeline without any stalls or queuing delays.

Problem M7.4.A

How much work is lost (in <u>cycles</u>) on a branch misprediction in this pipeline?

Problem M7.4.B

If one quarter of instructions are branches, and half of these are taken, then how much should we expect branches to increase the processor's CPI (cycles per instruction)?

Problem M7.4.C

You are unsatisfied with this performance and want to reduce the work lost on branches. Given your hardware budget, you can add only one of the following:

- A branch predictor to your pipeline that resolves after Instruction Fetch Stage 1.
- Or a branch target buffer (BTB) that resolves after Instruction Fetch Stage 2.

If each make the same predictions, which do you prefer? In one or two sentences, why?

Problem M7.4.D

You decide to add the BTB (not the branch predictor). Your BTB is a fully tagged structure, so if it predicts an address other than PC+4 then it always predicts the branch address of a conditional branch (but not the condition!) correctly. **For partial credit, show your work.**

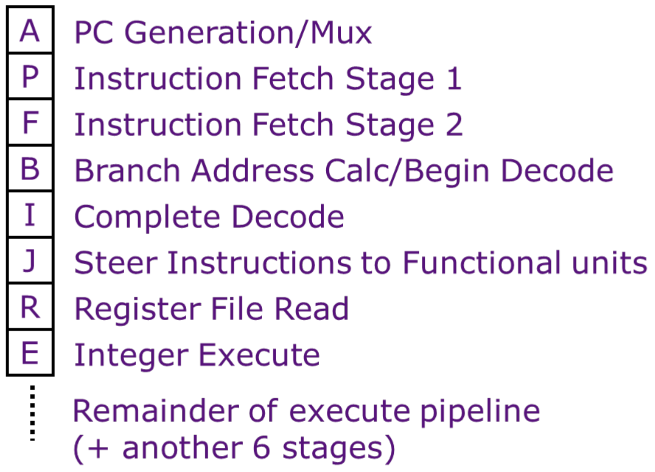
If the BTB correctly predicts a next PC other than PC+4, what is the effect on the pipeline?

If the BTB predicts the next PC incorrectly, what is the effect on the pipeline?

Assume the BTB predicts PC+4 90% of the time. When the BTB predicts PC+4 it is accurate 90% of the time. Otherwise it is accurate 80% of the time. How much should we expect branches to increase the CPI of the BTB design? *(Don't bother trying to compute exact decimal values.)*

Problem M7.5: Branch Prediction (Spring 2015 Quiz 2, Part A)

Ben Bitdiddle is designing a processor with the complex pipeline illustrated below:


A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute
Remainder of execute pipeline
(+ another 6 stages)

The processor has the following characteristics:

- Issues at most one instruction per cycle.
- Branch addresses are known at the end of the B stage (Branch Address Calc/Begin Decode).
- Branch conditions (taken/not taken) are known at the end of the R stage (Register File Read).
- Branches always go through the pipeline without any stalls or queuing delays.

Ben's target program is shown below:

```
for(int i = 0; i <= 1000000; i++)
{
    if(i % 2 == 0) //Branch B1
    { //Not taken
       (Do something A)
    }
    if(i % 4 == 0) //Branch B2
    { //Not taken
       (Do something B)
    }
}     //Branch LP
```

```
     ANDi R1  0
LOOP:MODi R2  R1  2
     BNE  R2  M4      // B1
     (Do something A)
     … …
M4:  MODi R3  R1  4
     BNE  R3  END     // B2
     (Do something B)
     … …
END: SUBi R4  R1  1000000
     BNE  R4  LOOP    // LP
     … …
```

The MODi (modulo-immediate) instruction is defined as follows:
`MODi Rd Rs imm: Rd <- Rs Mod imm`

Problem M7.5.A

In steady state, what is the probability for each branch in the code to be taken/not taken on average? Fill in the table below.

Branch	Probability to be **taken**	Probability to be **not taken**
B1		
B2		
LP		

Problem M7.5.B

In steady state, how many cycles per iteration are lost on average if the processor always speculates that every branch is not taken (i.e., next PC is PC+4)?

Problem M7.5.C

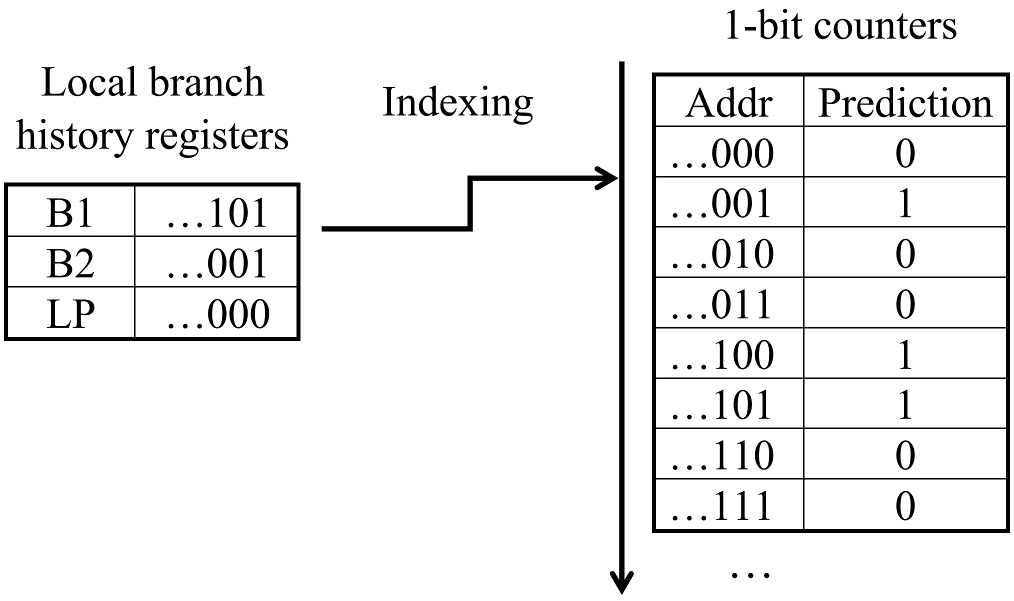
Ben designs a **static branch predictor** to improve performance. This predictor always predicts **not taken for forward jumps** and **taken for backward jumps**. The prediction is available at the end of the **B** stage. In steady state, how many cycles per iteration are lost on average?

Last updated:
10/6/2021

Problem M7.5.D

To improve performance further, Ben designs a **dynamic branch predictor with local branch history registers and 1-bit counters**.

Each local branch history registers store the last several outcomes of a single branch (branches B1, B2 and LP in our case). By convention, the most recent branch outcome is the least significant bit, and so on. The predictor uses the local history of the branch to index a table of 1-bit counters. It predicts not taken if the corresponding 1-bit counter is 0, and taken if it is 1. Assume local branch history registers are always correct.



B1	…101
B2	…001
LP	…000

Addr	Prediction
…000	0
…001	1
…010	0
…011	0
…100	1
…101	1
…110	0
…111	0

How many bits per branch history register do we need to perform perfect prediction in steady state?

Problem M7.5.E

The local-history predictor itself is a speculative structure. That is, for subsequent predictions to be accurate, the predictor has to be updated speculatively.

Explain what guess the local history update function should use.

Problem M7.5.F

Ben wants to design the data management policy (i.e., how to manage the speculative data in different structures of the predictor) for the local-history branch predictor to work well. Use a couple of sentences to answer the following questions.

1) What data management policies should be applied to each structure?

2) For your selected data management policies, is there any challenge for the recovery mechanism when there is misspeculation? If so, what are the challenges?