Last updated:
3/30/2021

Problem M7.1: Branch Prediction

```
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
```

Figure M3.6-A: BP bits state diagram

Problem M7.1.A — Program

R2 contains the number of non-zero entries in the first n elements of array.

Problem M7.1.B — 2-bit branch prediction

There are 7 mispredicts (shown in bold italics).

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*	N	N
b2	12/1	*10*	**00**	T	T
b1	16/0	**10**	*00*	N	***T***
b2	16/0	*11*	**00**	T	T
b1	20/1	**11**	*00*	N	N
b2	20/1	*10*	**00**	T	T
b1	24/0	**10**	*00*	N	***T***
b2	24/0	*11*	**00**	T	T
b1	28/1	**11**	*00*	N	N
b2	28/1	*10*	**00**	T	T
b1	32/0	**10**	*00*	N	***T***
b2	32/0	*11*	**00**	T	***N***

Table M7.1-1

Last updated:
3/30/2021

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

There are 9 mispredicts (shown in bold italics).

System State			Branch Predictor				Behavior	
PC	R3/R4	history bit	b1 bits		b2 bits		Predicted	Actual
			set 0	set 1	set 0	set 1		
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	N	***T***
b2	8/0	1	10	*11*	11	**10**	N	***T***
b1	12/1	1	10	**11**	11	*11*	N	N
b2	12/1	0	10	*10*	**11**	11	N	***T***
b1	16/0	1	10	**10**	*00*	11	N	***T***
b2	16/0	1	10	*11*	00	**11**	N	***T***
b1	20/1	1	10	**11**	00	*00*	N	N
b2	20/1	0	10	*10*	**00**	00	T	T
b1	24/0	1	10	**10**	*00*	00	N	***T***
b2	24/0	1	10	*11*	00	**00**	T	T
b1	28/1	1	10	**11**	00	*00*	N	N
b2	28/1	0	10	*10*	**00**	00	T	T
b1	32/0	1	10	**10**	*00*	00	N	***T***
b2	32/0	1	10	*11*	00	**00**	T	***N***

Table M7.1-2

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

There are 7 mispredicts (shown in bold italics).

System State			Branch Predictor								Behavior	
PC	R3/R4	history bits	b1 bits				b2 bits				Predicted	Actual
			set 00	set 01	set 10	set 11	set 00	set 01	set 10	set 11		
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	N	***T***
b2	8/0	11	10	10	*11*	10	10	11	10	**10**	N	***T***
b1	12/1	11	10	10	11	**10**	10	11	10	*11*	N	N
b2	12/1	01	10	10	11	*10*	10	**11**	10	11	N	***T***
b1	16/0	10	10	10	**11**	10	10	*00*	10	11	N	***T***
b2	16/0	11	10	10	*00*	10	10	00	10	**11**	N	***T***
b1	20/1	11	10	10	00	**10**	10	00	10	*00*	N	N
b2	20/1	01	10	10	00	*10*	10	**00**	10	00	T	T
b1	24/0	10	10	10	**00**	10	10	*00*	10	00	T	T
b2	24/0	11	10	10	*00*	10	10	00	10	**00**	T	T
b1	28/1	11	10	10	00	**10**	10	00	10	*00*	N	N
b2	28/1	01	10	10	00	*10*	10	**00**	10	00	T	T
b1	32/0	10	10	10	**00**	10	10	*00*	10	00	T	T
b2	32/0	**11**	**10**	10	*00*	10	10	00	10	**00**	T	***N***

Table M7.1-3

Problem 7.1.E **Analysis I**

The first thing to notice is that the more history bits we have, the longer it takes to get any correct prediction since we have to "train" the predictor. These start-up costs go up as the number of history bits increase.

Another thing to notice is that the single history bit does not help at all (even after we get into a steady-state phase). In both the single history bit and no history cases, the b2 branch is predicted correctly once we get past the start-up phase (since b2 is always taken). The single bit of history does not help since this history is too "nearsighted". The second history bit captures the alternating pattern of the b1 branch, and hence does not mispredict once it gets past the start-up phase. For a large n then, the 2-bit history predictor is the best.

The final point of observation is that all the predictors mispredict the fall-through case (the last b2 branch).

Problem 7.1.E **Analysis II**

When the input is random, no prediction scheme will help predict whether b1 is taken or not. All three schemes will eventually predict b2 as always taken. However, the more history bits are used, the more sets need to be trained to predict the always taken for b2. Thus, the more history bits used, the more mispredicts of branch b2 will occur initially. The answer does not depend on the size of n. However, as n gets large, the start-up costs become insignificant among the three schemes.

The moral of the problem is that history bits are useful if there is a pattern among a sequence of branches. The longer this pattern is, the more history bits are needed to be able to recognize this pattern. If the pattern is not recognized, then global history bits can hurt because it take longer to train the branches that can be predicted correctly.

Last updated:
3/10/2020

Problem M7.2:

Problem M7.2.A

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	4					
9	5	T	Y			
10	6					
11	1	T	N			
12	2					
13	4			T		TR
14	5	NT	N			
15	6			NT		TW
16	2					
17	3					
18	4					
19	5	T	Y			
20	6					
21	1	T	N			
22						
23				T		TR
24						
25				NT		TW
26						

Problem M7.2.B

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	4					
9	5	T	Y			
10	6			T		
11	1	NT	Y			
12	2			NT		
13	3					TR
14	4					
15	5	T	Y		NTR	
16	6			T		
17	1	NT	Y			
18				NT		
19						TR
20						
21					NTR	
22						
23						
24						
25						
26						

Problem M7.3: Branch Prediction

Problem 7.3.A

	Predicted Taken?	Actually Taken?	Pipeline bubbles
BEQZ/ BNEZ	Y	Y	3
	Y	N	6
	N	Y	6
	N	N	0
J	Always taken (No lookup)	Y	3
JR	Always taken (No lookup)	Y	6

Problem 7.3.B

	BTB Hit?	(BHT) Predicted Taken?	Actually Taken?	Pipeline bubbles
Conditional Branches	Y	Y	Y	1
	Y	Y	N	6
	Y	N	Y	Cannot occur
	Y	N	N	Cannot occur
	N	Y	Y	3
	N	Y	N	6
	N	N	Y	6
	N	N	N	0

Last updated:
3/10/2020

Problem M7.3.C

TIME →

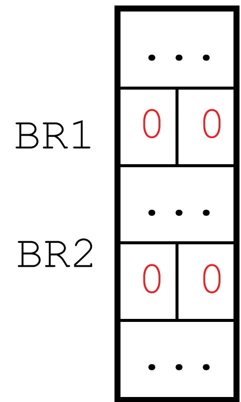
Address	Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0x1000	BEQZ R5, NEXT	A	P	F	B	I	J	R	E															
0x1014	ADDI R1, R1, #1								A	P	F	B	I	J	R	E								
0x1018	SLTI R2, R1, 100									A	P	F	B	I	J	R	E							
0x101C	BNEZ R2, LOOP										A	P	F	B	I	J	R	E						
0x1000	BEQZ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

Last updated:
3/10/2020

Problem 7.3.D

V (Valid)	Entry PC	Predicted Target PC
1	0x101c	0x1000
1	0x1000	0x1014

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 cycles) on a branch misprediction in this pipeline?

6 cycles are lost when stalls are inserted into pipeline stages A, P, F, B, I and J.

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)?

This answer is asking how much CPI is spent on branches in the machine, increase relative to a machine that never stalls on branches (e.g. has "magic fetch").

Branch CPI = misprediction rate x misprediction penalty

From the problem description, we always predict PC+4 or "not taken". So the misprediction rate is just the rate of taken branches.

Branch CPI = fraction branches x fraction taken x misprediction penalty

From the question:

Branch CPI = $\frac{1}{4} \times \frac{1}{2} \times 6 = \frac{3}{4}$

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?

Branch predictions earlier than B are unhelpful since we don't have an address to jump to even if the branch is predicted taken. So although the BTB is available later in the pipeline, it is better to have the BTB since it gives us an address we can use to redirect fetch.

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?

We inject two stalls into stages A and P and redirect fetch to the BTB address. So we lose 2 cycles.

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

The BTB has exactly the same misprediction penalty as the baseline machine—6 cycles. This is true regardless of whether the BTB predicted PC+4 or a different address, since no matter what after an incorrect prediction the branch will be followed by six stalls in the pipeline. (The penalties are not additive.)

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.)*

This is simply a matter of computing the probabilities of all combinations of prediction and accuracy and their associated penalties.

Denote each case as "prediction/actual". So "T/NT" means the BTB predicted a PC other than PC+4, but it turned out that PC+4 was the actual branch resolution.

Branch CPI = T/T CPI + T/NT CPI + NT/T CPI + NT/NT CPI

NT/NT CPI is zero since this just means the BTB predicted PC+4 and no stalls happened.

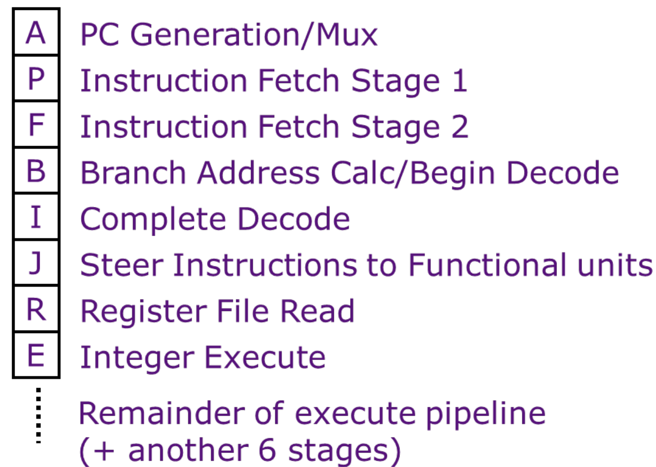
T/T CPI incurs a penalty of 2 cycles (see above), and this happens when the BTB predicts a PC other than PC+4 (10%) *and* it is correct (80%). So T/T CPI = 2 * 0.1 * 0.8

T/NT CPI and NT/T CPI both incur a penalty of 6 cycles (see above). These occur when the BTB is incorrect about its prediction: T/NT rate is 0.1 * 0.2, NT/T rate is 0.9 * 0.1. So T/NT CPI = 0.1 * 0.2 * 6 and NT/T CPI = 0.9 * 0.1 * 6.

Branch CPI = (0.1 * 0.2 + 0.9 * 0.1) * 6 + 0.1 * 0.8 * 2 = 0.82

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

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



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	0.5	0.5
B2	0.75	0.25
LP	~1	~0

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)?

Penalty for miss prediction = 6 cycles

6 * 0.5 + 6 * 0.75 + 6 * 1 = 13.5

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?

Penalty for miss prediction = 6 cycles

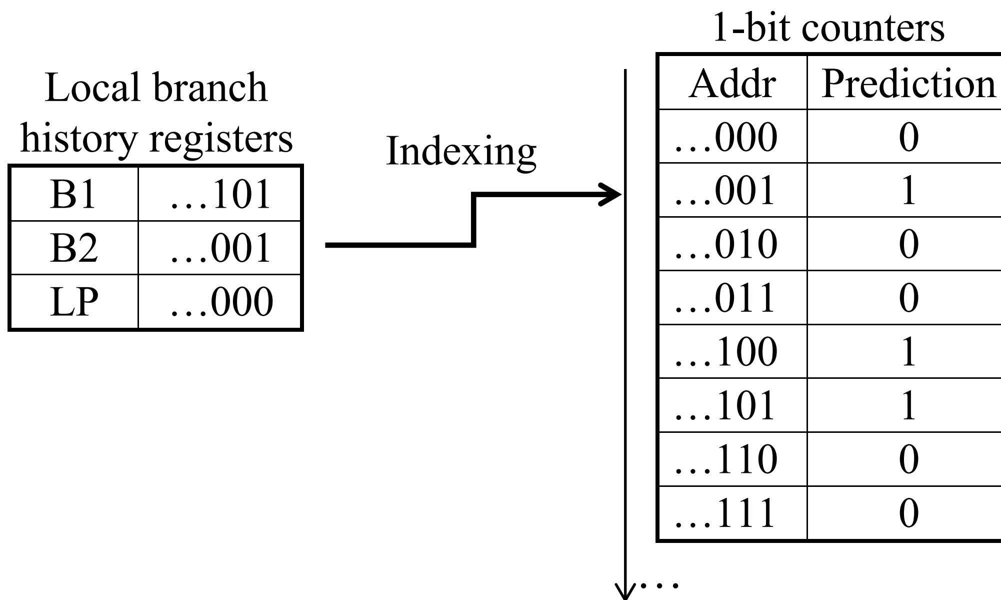
Penalty for correct prediction for taken = 3 cycles

6 * 0.5 + 6 * 0.75 + 3 * 1 = 10.5

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.



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

4 bits

B1: 01 => 0
10 => 1
B2: 0001 => 0
0010 => 0
0100 => 0
1000 => 1
LP: (all pattern) => 1

(Using 3 bits will have collision for pattern 010 of B1 and B2)

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.

Guess the prediction is correct and use the prediction to update history register

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?

   Greedy update for history registers and lazy update for 1-bit predictors

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?

   Recovery mechanism for history registers will be hard. We need to record all the information (PC, execution order) about branches that speculatively update the history registers and roll back the history register with the information sequentially.