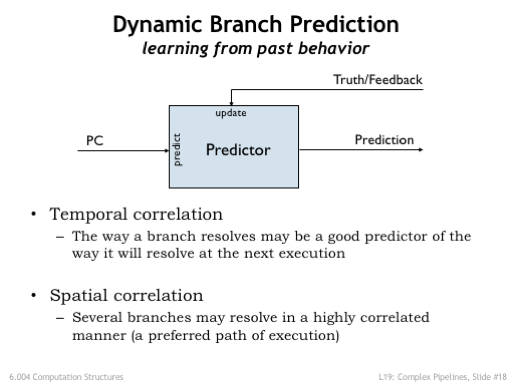
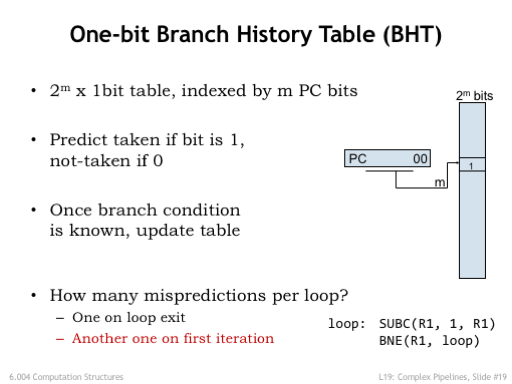
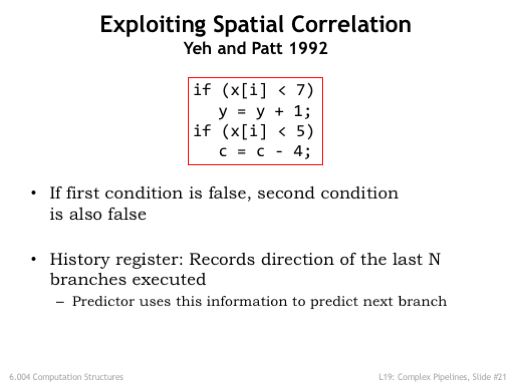
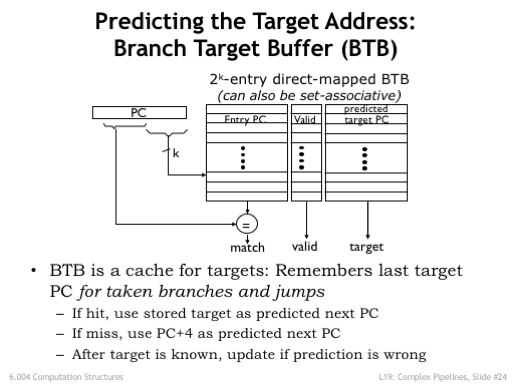
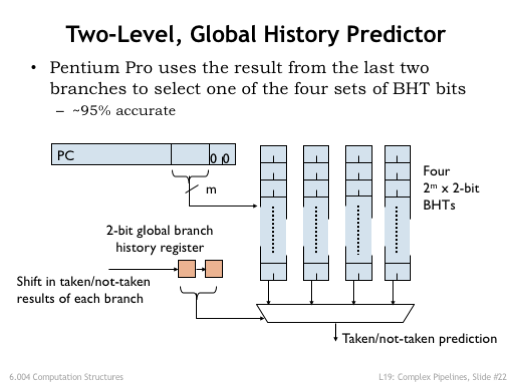
****

ANSWERS

**6.004 Worksheet**

**L19 – Complex Pipelines**

****

****

**Problem 1.**

In this problem, we will investigate the effects of adding global history bits to a standard branch prediction mechanism. Throughout this problem we will be working with the program fragment shown in Figure 1. Assume the initial value of R1 is N (N>0), the initial value of R2 is 0 (R2 holds the result of the program) and the initial value of R3 is a pointer to the beginning of an array of 32-bit integers with N elements.

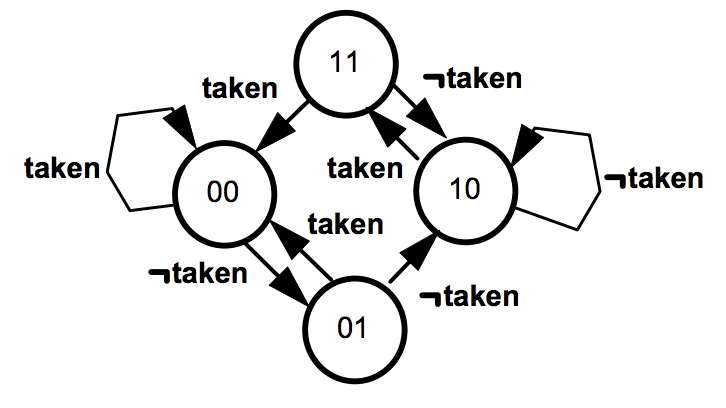


Figure 2: BP bits state diagram

loop: LD(R3, 0, R4)  
 ADDC(R3, 4, R3)  
 SUBC(R1, 1, R1)  
b1: BEQ(R4, b2)  
 ADDC(R2, 1, R2)  
b2: BNE(R1, loop)

Figure 1: Sample program

We’ll start by assuming that branch prediction uses a branch history table (BHT) with 2m entries indexed by bits [m+1:2] of the current value of the program counter. Whenever there’s a branch, the address of the branch instruction is used to look up a 2-bit state value from the BHT. We’ll predict the branch is taken if the state is 00 or 01 and we’ll predict the branch is not taken if the state is 10 or 11. When we know whether the actual branch was taken or not, the 2-bit state stored in the BHT is updated using the state transition diagram shown in Figure 2. This FSM will change the prediction only after two incorrect predictions.

(A) What does the program compute? That is, describe the value of R2 when we exit the loop.

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

Now we will investigate how well our standard 2-bit branch predictor performs. Assume the following:

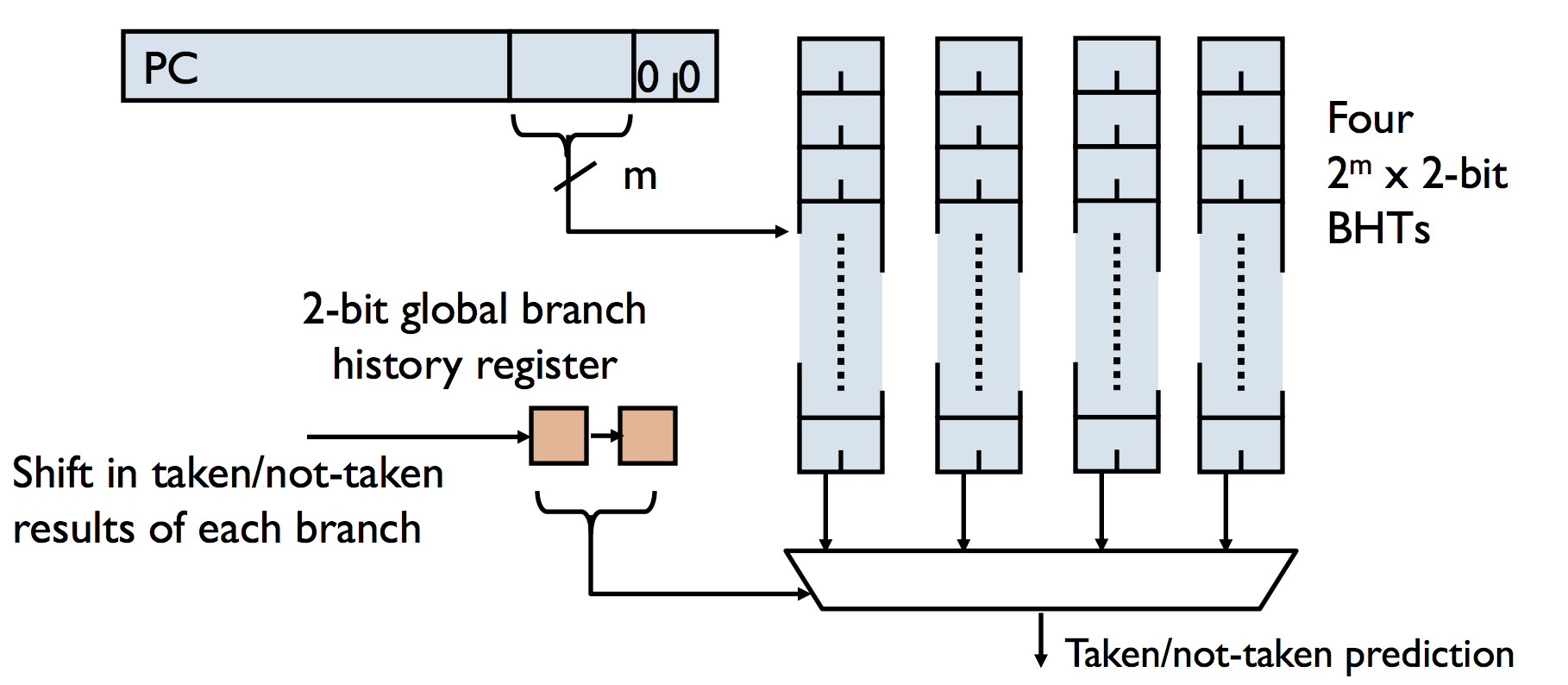
* Addresses b1 and b2 map to different entries in the BHT
* Initially R1 (=N) is 8 and R3 (= address of p[0]) is 4.
* The array contains an alternating pattern of 1’s and 0’s, i.e., p[0] = 1, p[1] = 0, p[2] = 1, p[3] = 0,… etc.
* The initial state of each BHT entry is 10.

(B) Fill out the table below, which contains an entry for every branch (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). The first few lines of the table have already been filled out. Use “\*” to mark predictions that are mispredictions.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| System state | | | BHT entries | | Branch behavior | |
| PC | R3 | R4 | BHT[b1] | BHT[b2] | 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 |

(C) What is the total number of mispredictions?

There are 7 mispredictions.

The performance of the predictor can be improved by adding an N-bit global branch history register that remembers the actual behavior (taken/not-taken) of the N most-recent branches. The global history register is used to select between multiple BHTs running in parallel as shown below.

(D) Assume we add a 1-bit global history register to the branch predictor. Fill out the table below when re-running the program using the same assumptions as in part (B). The initial value of the history bit is shown in the table.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| System state | | | | BHT entries | | | | Branch behavior | |
| PC | R3 | R4 | History bit | BHT[b1] | | BHT[b2] | | 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 |

(E) What is the total number of mispredictions using 1 global history bit?

There are 9 mispredictions.

(F) Now add a second global history bit. Fill out the table below when re-running the program using the same assumptions as in part (B). The initial values of the history bits are shown in the table.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| System state | | | | BHT entries | | | | | | | | Branch behavior | |
| PC | R3 | R4 | History bits | BHT[b1] | | | | BHT[b2] | | | | Predicted | Actual |
| Set 0 | Set 1 | Set 2 | Set 3 | Set 0 | Set 1 | Set 2 | Set 3 |
| 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 |

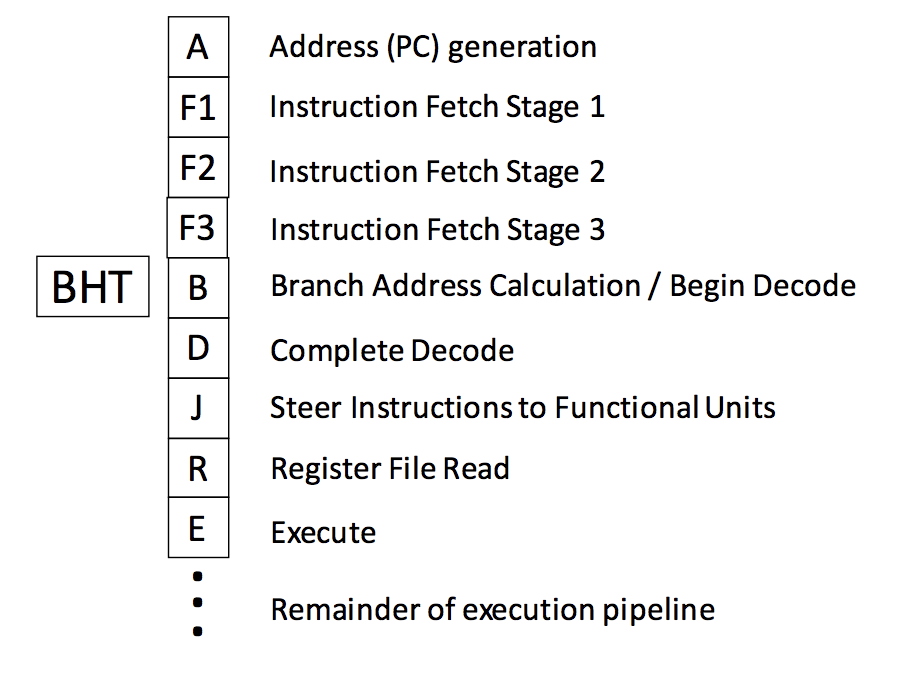
(G) What is the total number of mispredictions using 2 global history bits?

There are 7 mispredictions.

(H) Compare your results from parts (C), (E), (G). When do most of the mispredictions 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.

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 2.**

Ben Bitdiddle is designing a processor with a complex pipeline, shown to the right. 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 conditions (taken / not taken) are known at the end of the **R stage**.
* Branches go through the pipeline without any stalls or queueing delays.

For this question, assume there are no control flow instructions other than conditional branches, i.e., no JMP instructions.

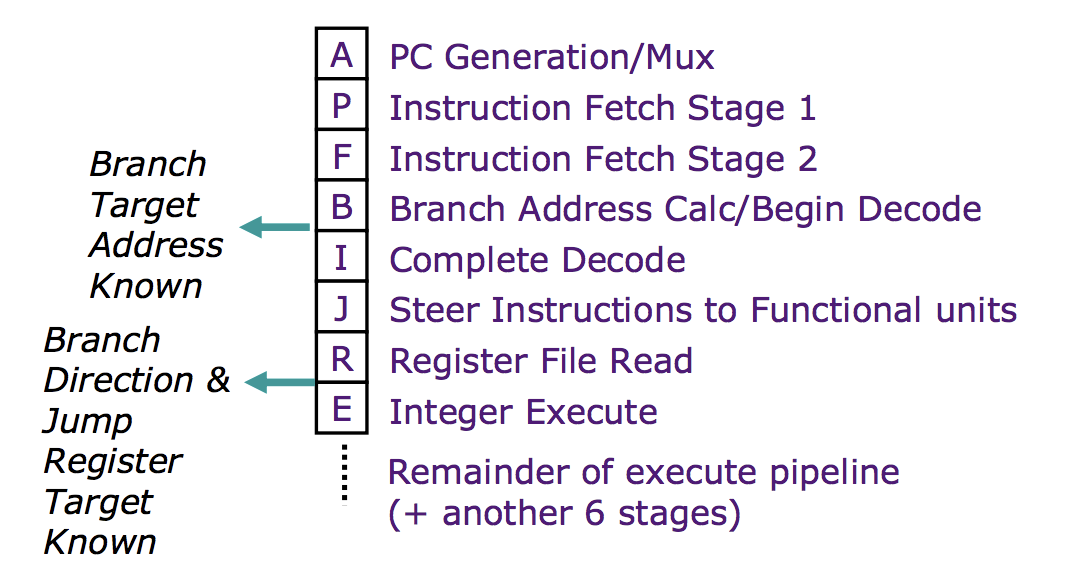
Ben adds a branch history table (BHT) to the pipeline as shown. With this addition, fetches work as follows. The A stage fetches the instruction at PC+4 by default. In the B stage (Branch Address Calc/Begin Decode), a conditional branch instruction (BEQ/BNE) accesses the BHT. If the branch is predicted to be taken, later instructions (stages A-F3) are flushed and the PC is redirected to the calculated branch target address. Mispredictions are discovered in the R stage and cause stages A-J to be flushed.

Fill in each cell with the branch penalty (in instruction fetches wasted) for each combination of prediction and actual branch outcome.

|  |  |  |
| --- | --- | --- |
| BHT Prediction | Penalty if branch is taken | Penalty if branch is not taken |
| TAKEN | 4 (flush A-F3) | 7 (flush A-J) |
| NOT TAKEN | 7 (flush A-J) | 0 (no flush needed!) |

**Problem 3.**

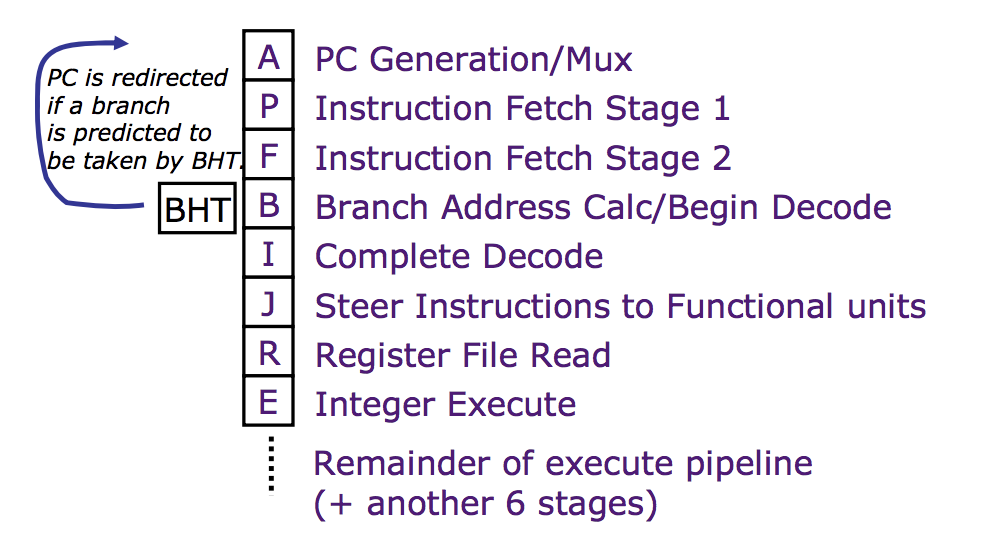
Consider the fetch pipeline of UltraSparc-III processor. In this part, we evaluate the impact of branch prediction on the processor’s performance.



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

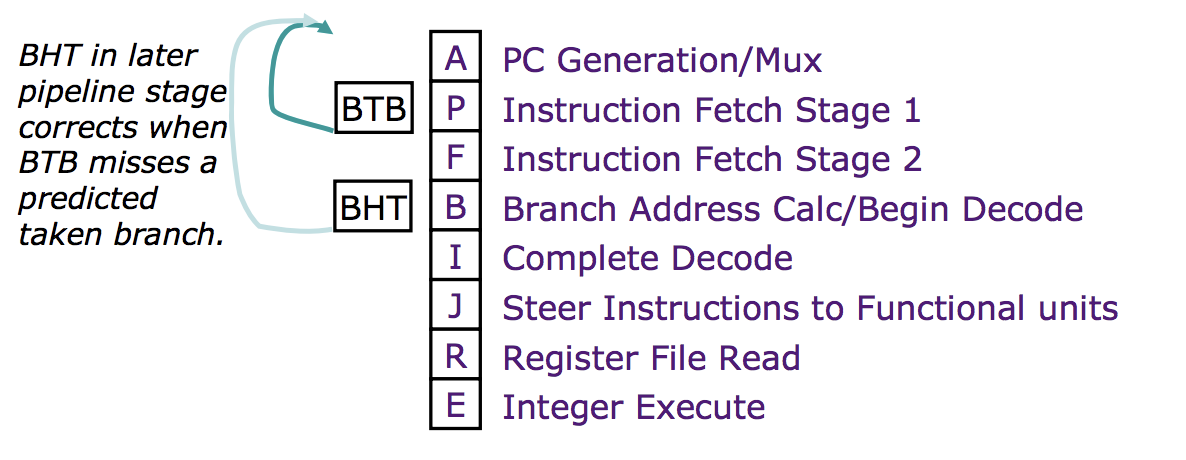
|  |  |  |
| --- | --- | --- |
| Instruction | Branch decision known? (at the end of) | Target known?  (at the end of) |
| BEQ/BNE | R | B |
| JMP | B (always taken) | R |

(A) As a first step, we add a branch history table (BHT) in the fetch pipeline as shown below. In the B stage (Branch Address Calc/Begin Decode), a conditional branch instruction (BEQ/BNE) looks up the BHT, but an unconditional JMP does not. If a branch is predicted to be taken, stages A-F 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.



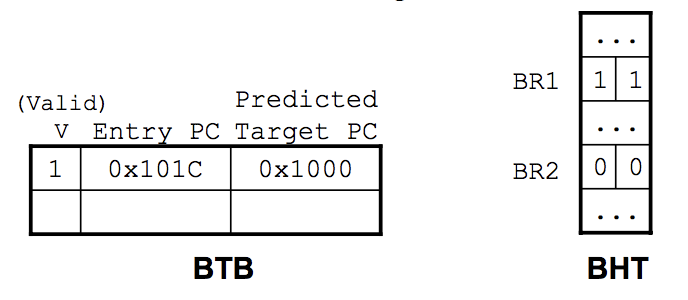
For each of the following cases, write down the number of pipeline bubbles caused by a branch or jump. If there is no bubble, you can simply put 0.

|  |  |  |  |
| --- | --- | --- | --- |
| *Instruction* | *Predicted taken?* | *Actually taken?* | *Pipeline Bubbles* |
| BEQ/BNE | Yes | Yes | 3 |
| Yes | No | 6 |
| No | Yes | 6 |
| No | No | 0 |
| JMP | Always taken  (no lookup) | Yes | 6 |

(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 entryPC, targetPC pairs for jumps and branches *predicted to be taken*. Assume that the targetPC predicted by the BTB is always correct for this question. (Yet the direction still might be wrong.)  
  
2. The BTB is accessed every cycle. If there is a match with the current PC, PC is redirected to the targetPC predicted by the BTB (unless PC is redirected by an older instruction); if not, it is set to PC+4.   
  
Fill out the following table of the number of pipeline bubbles (only for conditional branches).

|  |  |  |  |
| --- | --- | --- | --- |
| *BTB hit?* | *BHT predicted taken?* | *Actually taken?* | *Pipeline Bubbles* |
| Yes | Yes | Yes | 1 |
| Yes | Yes | No | 6 |
| Yes | No | Yes | Cannot occur |
| Yes | No | No | Cannot occur |
| No | Yes | Yes | 3 |
| No | Yes | No | 6 |
| No | No | Yes | 6 |
| No | No | No | 0 |

(C) We will be working the program shown below. The initial values of R1 and R5 are 0, so the branch at BR1 is always taken. Disregard any possible data hazards. There are no pipeline bubbles except for those created by branches. We fetch only one instruction per cycle.  
  
  
  
  
  
  
  
  
  
  
  
  
  
The BHT uses a 2-bit predictor with the same state diagram as shown in Figure 2 of Problem 1. Assume the BR1 and BR2 map to different entries in the BHT. We use a two-entry full-associative BTB with an LRU replacement policy. A snapshot of the initial state of the BTB and BHT is shown below.  
  
Please fill in the timing diagram to the right for one iteration plus the first two instructions of the next iteration. For each executed instruction indicate the cycle it enters each pipeline stage. Don’t worry about the stages beyond the E stage.

*Addr* *Instruction*

0x1000 BR1: BEQ(R5, NEXT) // always taken  
0x1004 ADDC(R4, 4, R4)

0x1008 MUL(R5, R3, R3)

0x100C ST(R3, 0, R4)

0x1010 SUBC(R5, 1, R5)

0x1014 NEXT: ADDC(R1, 1, R1)

0x1018 CMPLTC(R1, 100, R2) // repeat 100 times

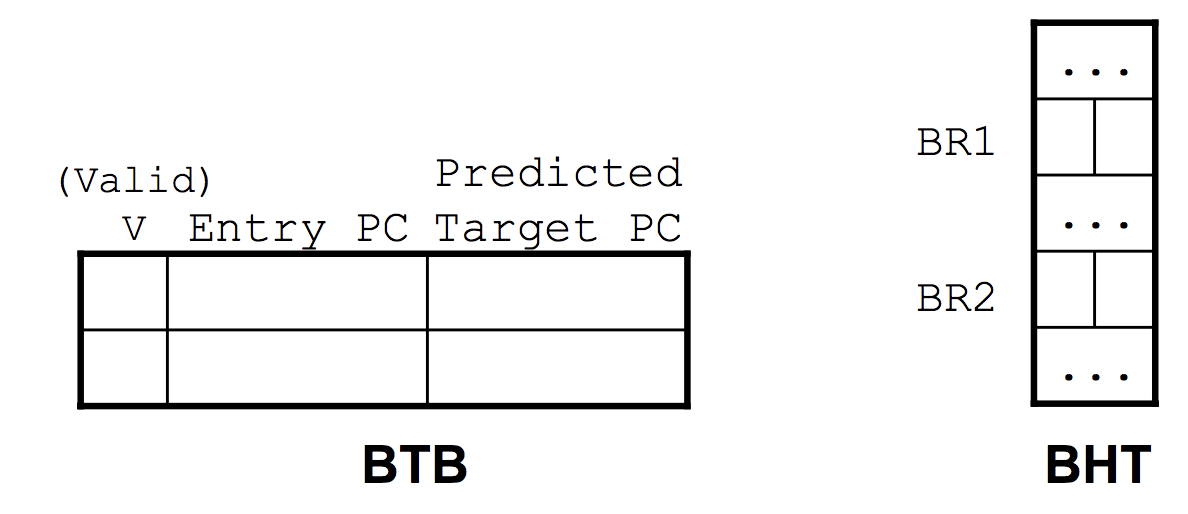
0x101C BR2: BT(R2, BR1)

0x1020 NOP

0x1024 NOP

0x1028 NOP

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Time | 0x1000 BEQ(R5,NEXT) | 0x1014 ADDC(R1,1,R1) | 0x1018 CMPLTC(R1,100,R2) | 0x101C BT(R2,BR1) | 0x1000 BEQ(R5,NEXT) | 0x1014 ADDC(R1,1,R1) |
| 1 | **A** |  |  |  |  |  |
| 2 | **P** |  |  |  |  |  |
| 3 | **F** |  |  |  |  |  |
| 4 | **B** |  |  |  |  |  |
| 5 | **I** |  |  |  |  |  |
| 6 | **J** |  |  |  |  |  |
| 7 | **R** |  |  |  |  |  |
| 8 | **E** | **A** |  |  |  |  |
| 9 |  | **P** | **A** |  |  |  |
| 10 |  | **F** | **P** | **A** |  |  |
| 11 |  | **B** | **F** | **P** |  |  |
| 12 |  | **I** | **B** | **F** | **A** |  |
| 13 |  | **J** | **I** | **B** | **P** |  |
| 14 |  | **R** | **J** | **I** | **F** | **A** |
| 15 |  | **E** | **R** | **J** | **B** | **P** |
| 16 |  |  | **E** | **R** | **I** | **F** |
| 17 |  |  |  | **E** | **J** | **B** |
| 18 |  |  |  |  | **R** | **I** |
| 19 |  |  |  |  | **E** | **J** |
| 20 |  |  |  |  |  | **R** |
| 21 |  |  |  |  |  | **E** |
| 22 |  |  |  |  |  |  |
| 23 |  |  |  |  |  |  |
| 24 |  |  |  |  |  |  |

(D) After the 6 instructions have executed and the predictor’s state has been updated, please show the contents of the BTB and BHT.

0 0

0 0

1 0x101C 0x1000

1 0x1000 0x1014