**EECS 470 Homework 3, [redacted]**

1.
   a. The BTB target address could change if the branch's target address were to change. This could happen if we had an indirect branch.
   b. As the branch is a "hit" (in the BTB) the tag could not change.
   c. If the branch was predicted weakly taken it could change to strongly taken. In no other case would it change.
   d. You'd expect it to change unless it currently has all 1s (taken) in the history.
   e. If the branch was predicted weakly taken it could change to strongly taken. In no other case would it change.
   f. Both local and global have the same prediction, so no change.
   g. You'd expect it to change unless it was currently all 1s (taken) in the history.

2. Having a larger set of architected can help if you have a large number of "live values". That is values we have and will need in later computations. If you have enough architected registers, those values can be left in the architected registers. Otherwise you need to "spill" those values to memory (the stack) and "fill" them later when needed.

   Put another way, architected registers are a way for the programmer/complier to name things and keep them around without using loads and stores. Physical registers are *not* visible to the programmer/complier and cannot be so used.

3. Since the load stalls for a long time, we have to see if we first encounter a structural hazard due to the RS being full or the ROB being full. The ROB will be full once we hit the $6^{th}$ instruction no matter what (until the load finishes). The RS will become full if there are 3 instructions that cannot execute due to data dependencies on the load.

   In program A, there is only one instruction within the first 6 of the program that has a dependency on the load (R4=R1+6). So, the ROB will fill up before the RS and the last instruction to enter the ROB is the $6^{th}$ instruction: R6=R2+R6.

   For program B, the second instruction (that writes to R2) is dependent on the load, and the $4^{th}$ and $5^{th}$ instructions are dependent on R2 written by the second instruction. Therefore, the RS fills up at the $5^{th}$ instruction, and the last instruction that gets placed in the ROB before the load finishes is the $5^{th}$ instruction: R1=R2+R3.

4.

Consider the following tables that represent the state of a processor that implements what we have called the P6 scheme.

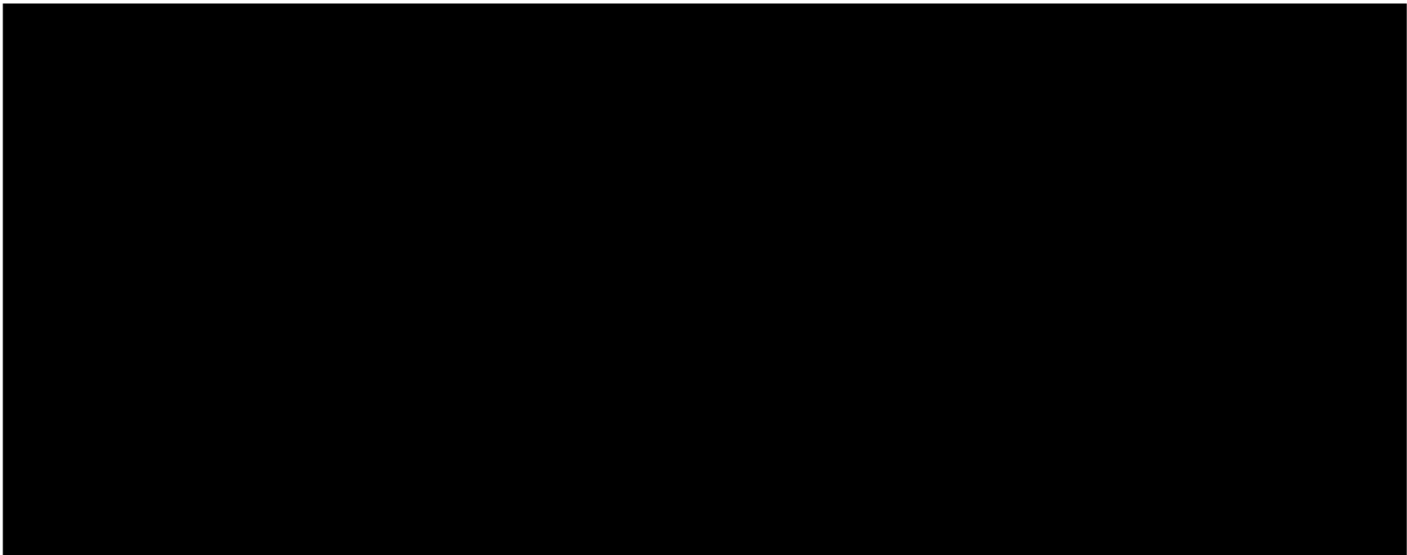
| RAT | |
|---|---|
| **Arch Reg. #** | **ROB# (-- if in ARF)** |
| **0** | -- |
| **1** | 1 |
| **2** | 3 |
| **3** | 2 |
| **4** | -- |
| **5** | 4 |

| ROB | | | | |
|---|---|---|---|---|
| **Buffer Number** | **PC** | **Done with EX?** | **Dest. Arch Reg #** | **Value** |
| 0 | | | | |
| 1 | 44 | N | 1 | |
| 2 | 48 | N | 3 | |
| 3 | 4C | N | 2 | |
| 4 | 50 | N | 5 | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |

| RS | | | | | | |
|---|---|---|---|---|---|---|
| **,** | **Op type** | **Op1 ready?** | **Op1 RoB/value** | **Op2 ready?** | **Op2 RoB/value** | **Dest ROB** |
| 0 | + | N | 1 / -- | Y | -- / 9 | 2 |
| 1 | + | N | 2 / -- | Y | -- / 1 | 4 |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | [Optional] * | Y | -- / 2 | Y | -- / 4 | 3 |

| **ARF** | **Reg#** | **0** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|---|
| | **Value** | 6 | 5 | 4 | 9 | 2 | 1 |

**Head= ROB1, Tail =ROB5 (or 4, it's not clear what to use, but normally)**



Tables below shows full execution time
Assuming 1 cycle latency for multiplies:

| Inst | Dispatch | Issue | Execute | Complete | Retire |
|---|---|---|---|---|---|
| A: R3=R2+R1 | 1 | 2 | 3 | 4 | 5 |
| B: R1=R1*R3 | 2 | 4 | 5 | 6 | 7 |
| C: R3=R1+R3 | 3 | 6 | 7 | 8 | 9 |
| D: R2=R4*R2 | 4 | 5 | 6 | 9* | 10 |
| E: R5=R3+R4 | 5 | 8 | 9 | 10 | 11 |

*stalls due to structural hazard with CDB in cycle 8

Assuming 2 cycle latency for multiplies.

| Inst | Dispatch | Issue | Execute | Complete | Retire |
|---|---|---|---|---|---|
| A: R3=R2+R1 | 1 | 2 | 3 | 4 | 5 |
| B: R1=R1*R3 | 2 | 4 | 5 | 7 | 8 |
| C: R3=R1+R3 | 3 | 7 | 8 | 9 | 10 |
| D: R2=R4*R2 | 4 | 5 | 6 | 7 | 8 |
| E: R5=R3+R4 | 5 | 9 | 10 | 11 | 12 |

5.

| Index | Predictor | Index | Predictor |
|---|---|---|---|
| 0 | 11 | 8 | ~~00~~ **01** |
| 1 | 11 | 9 | 10 |
| 2 | 00 | 10 | 11 |
| 3 | 11 | 11 | ~~10~~ 11 |
| 4 | 01 | 12 | 11 |
| 5 | 00 | 13 | 00 |
| 6 | 10 | 14 | ~~01~~ **00** |
| 7 | ~~01~~ **10** | 15 | 01 |

.

6.

a. In the algorithm we are calling R10K, when using an RRAT we will free ***all PRF entries / no PRF entries / those PRF entries which aren't pointed to by the RRAT / those PRF entries in the RAT that are overwritten by the RRAT*** when a branch mispredict occurs.

b. If, in the algorithm we are calling P6, the RAT had only one port for writing values, you could only have one instruction that writes a register ***dispatch / issue / complete execution / retire*** per cycle.

c. If, in the algorithm we are calling R10K, the PRF had only one write port, you could only have one instruction that writes a register ***dispatch / issue / complete execution / retire*** per cycle.

d. In the original Tomaulo's algorithm, the RAT points to ***a reorder buffer entry / a reservation station / a physical register / an execution unit***.

e. The branch target buffer is a(n) ***address / confidence / direction*** predictor that has particular problems with ***function calls / conditional branches / returns from functions / indirect branches***. To help with that case we might add a(n) ***Translation Lookaside Buffer (TLB) / Reorder Buffer (RoB) / Alias Table List Address Selector (ATLAS) / Return Address Stack (RAS)***.

f. In the algorithm we are calling R10K, if you have 32 RoB entries, 16 architected registers, 8 RS entries and a 4KB instruction-cache, you will not have a use for more than ***8 / 16 / 32 / 48 / 64*** PRF entries.

g. Given a 4-KB four-way associative cache with 16-byte cache lines and a 32-bit address space there will be __**6**_____ bits used for the index. If that same cache were direct-mapped you'd need ___20____ bits to be used for the tag.

h. In the P6 algorithm, a processor with 2 RSes, 16 RoB entries, and 8 architected registers would have a RAT whose size in bits is ***10 / 16 / 20 / 32 / 40 / 96 / 128*** ignoring state bits (like valid etc.). If that same design were using the algorithm we've called R10K, the RAT would have ***10 / 16 / 20 / 32 / 40 / 96 / 128*** bits (again ignoring state bits).