Control Hazards

Given a loop that has two inner branches, B1 and B2, the results of the branches are given.
Fill in the prediction made by different predictors.

Assume:
- T = Taken, NT = Not Taken.
- All counters, unless otherwise specified, are 2-bit saturating counters.
- Two bit counters are initialized as weakly taken.
- One bit counters are initialized as not taken.
- All history before the execution of the trace is not taken.
- Branches are executed in the order below (left to right):

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
</tr>
</tbody>
</table>

- The PC address of the branches are PC(B1) = 0x28, PC(B2) = 0x34.
- History[2:0] is a 3-bit shift register. 0 indicates a prior branch is NT, 1 indicates a prior branch is T. The most recently resolved branch action is stored in History[0].

The predictors in the problem below are:
- Static-NT : Static Not Taken
- Static-T : Static Taken
- Bimodal : 1-bit saturating counter table indexed by PC[4:2]
- Global : 2-bit saturating counter table indexed by History[2:0]
- Gshare : Global correlation predictor. 2-bit saturating counter table indexed by XOR(PC[4:2], History[2:0])
(a) Complete the table of predictions and branch prediction accuracy. On each row, write the state of the corresponding predictor before it is updated by the branch. For 2-bit predictors, write SNT, WNT, WT, or ST to distinguish between strong and weak prediction.

Note: Two additional columns are given to you for intermediate computation.

<table>
<thead>
<tr>
<th>Branch</th>
<th>Action</th>
<th>Static-NT</th>
<th>Static-T</th>
<th>Bimodal</th>
<th>Global</th>
<th>Gshare</th>
<th>History</th>
<th>xor</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1(010)</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>wT</td>
<td>wT</td>
<td>000</td>
<td>010</td>
</tr>
<tr>
<td>B2(101)</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>wT</td>
<td>wT</td>
<td>001</td>
<td>100</td>
</tr>
<tr>
<td>B1</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>wT</td>
<td>wT</td>
<td>010</td>
<td>000</td>
</tr>
<tr>
<td>B2</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>wT</td>
<td>sT</td>
<td>101</td>
<td>000</td>
</tr>
<tr>
<td>B2</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>wT</td>
<td>wT</td>
<td>011</td>
<td>110</td>
</tr>
<tr>
<td>B1</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>wT</td>
<td>wNT</td>
<td>110</td>
<td>100</td>
</tr>
<tr>
<td>B2</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>sT</td>
<td>sT</td>
<td>101</td>
<td>000</td>
</tr>
<tr>
<td>B2</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>wNT</td>
<td>wNT</td>
<td>011</td>
<td>110</td>
</tr>
<tr>
<td>B1</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>sT</td>
<td>wT</td>
<td>110</td>
<td>100</td>
</tr>
<tr>
<td>B1</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>sT</td>
<td>wT</td>
<td>101</td>
<td>111</td>
</tr>
</tbody>
</table>

| B1 Accuracy | 1/5 | 4/5 | 3/5 | 4/5 | 3/5 |
| B2 Accuracy | 3/5 | 2/5 | 1/5 | 3/5 | 3/5 |

(b) From the results above, which branch prediction technique would you choose for both B1 and B2. Circle only one answer.
There are five processor designs, each with a different branch predictor and misprediction penalty (# of cycles.) Rank the processor designs according to the number of cycles wasted due to misprediction for the branches B1 and B2 in the trace above. Indicate the ranking by writing an integer from 1 (fewest cycles) to 5 (greatest cycles) under each column. If two or more processors rank equally, write the same integer number for them.

<table>
<thead>
<tr>
<th>Processor</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predictor</td>
<td>Static-NT</td>
<td>Static-T</td>
<td>Bimodal</td>
<td>Global</td>
<td>Gshare</td>
</tr>
<tr>
<td>Penalty</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Rank</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Note: The following parts are independent of a, b, and c.

(d) Rewrite the following LC-3 program to make use of two branch delay slots.

```
LOOP:
    Add R0, R0, 3
    And R0, R0, -2
    Add R1, R1, -1
    Brp LOOP
    nop
    nop
```

```
LOOP:
    Add R1, R1, -1
    Brp LOOP
    Add R0, R0, 3
    And R0, R0, -2
```
(e) The following LC-3 program is run on a processor with static taken branch prediction and a BTB with unknown size. The initial register values are R2=4 and all others zero. The BTB is initially empty.

A:
   And R0, R1, 2
   Brz B
   Add R2, R2, -2
   Brnzp A

B:
   Add R2, R2, -1
   Brp A

(i) Assume the BTB is large enough for this program. Write the number of branches that will have a correct direction prediction. Also, write the number of branches that will have a correct target prediction.

7 taken out of /8 executed. 2 BTB misses, 6 BTB hits. Accuracy = 5/8

(ii) Now, you observe that the predicted branch target is always incorrect. What is the size of the BTB? Explain how this could happen.

<=4 entries, because 0 accuracy implies that “Brz B” and “Brp A” alias to the same location in the BTB.
3. Data Hazards (13 points)
Consider a 6 stage pipeline: IF, ID, EX, MEM1, MEM2, WB
Assume:
- The ISA is LC3-b.
- Always hit 1-cycle I-Cache.
- Always hit 1-cycle D-Cache.
- D-cache only has an 8-bit bus to the CPU. Therefore, it takes two cycles for a 16 bit read or a write to complete.
- EX calculates memory addresses.
- MEM1 retrieves the low byte of the word requested, MEM2 retrieves the high byte.
- As in your MP2 cache, the last bit of a memory address is ignored. e.g. 0x0000 and 0x0001 are treated as the same address by your cache.
- There is only 1 set of forwarding paths available: from MEM1 -> EX
- Transparent register file (Writes first half of cycle, reads second half).
- Only 1 instruction may be in MEM1/2 at a time.

(a) Identify all of the data dependencies (both true dependencies and anti-dependencies) in this program. Specify in the following format:

Type: \( I_1 \rightarrow I_2 \) (\( R_e \)) e.g. RAW \( I_8 \rightarrow I_7 \) (\( R_0 \))

\[
\begin{align*}
I_1 & - ADD R1, R2, R3 \\
I_2 & - ADD R2, R1, R4 \\
I_3 & - LDR R2, R2, R4 \\
I_4 & - LDB R6, R7, 25 \\
I_5 & - STR R7, R6, 12 \\
I_6 & - ADD R3, R5, R7 \\
\end{align*}
\]

\( R1 = R2 = R3 = R4 = R5 = R6 = R7 = 0 \) at beginning of program

(3 points)

RAW: \( I_2 \rightarrow I_1 \) (\( R_1 \)) \hspace{1cm} \text{WAR: } I_2 \rightarrow I_1 \) (\( R_2 \))

RAW: \( I_3 \rightarrow I_2 \) (\( R_2 \)) \hspace{1cm} I_3 \rightarrow I_1 \) (\( R_3 \))

RAW: \( I_5 \rightarrow I_4 \) (\( R_6 \)) \hspace{1cm} I_6 \rightarrow I_1 \) (\( R_3 \))
(b) Which of the above dependencies will cause stalls? Specify in the following format:

Type: $I_{in} \rightarrow I_{out} \ (Rn), \ # \ cycles.$

(2 points)

RAW $I_2 \rightarrow I_1 \ (R1), \ 1 \ cycle$
RAW $I_3 \rightarrow I_2 \ (R2), \ 1 \ cycle$
RAW $I_5 \rightarrow I_4 \ (R6), \ 3 \ cycles$

ADD $R2, R2, R3 \quad \text{IF ID} \quad \text{EX} \quad M1 \ M2 \ WB$
ADD $R2, R2, R4 \quad \text{IF ID} \quad \text{EX} \quad M1 \ M2 \ WB$
LDR $R2, R2, R4 \quad \text{IF ID} \quad \text{EX} \quad M1 \ M2 \ WB$
LDB $R2, R2, 25 \quad \text{STR} \quad R7, R2 \quad 12$

Note: part (c) on the next page.
(d) Consider instructions LBI, SBI added to the LC3 ISA.

- LBI is load byte indirect, which loads a single byte from a byte array in memory.
- SBI is store byte indirect, which stores a single byte to a byte array in memory.
- Full specification is given in Appendix A.

**Question:** in the figure on the next page, draw forwarding paths and stalling logic so that LBI and SBI cause minimal stalling but provide correct operation even when data dependencies exist for these instructions. Fully specify the control signals. Control signals and forwarded values should originate from stage latches and terminate at forwarding MUXes. Label the destination of each MUX that you use.

**Requirements:**
- You may use only basic logic gates and comparators. You must show all paths related to LBI/SBI. Do not rely on existing forwarding paths.
- You need not consider control hazards or structural hazards.
- You must show your stall logic for all data hazards involving LBI/SBI but you need not show your stall mechanism i.e., an arrow captioned “stall” is sufficient.
- To simplify your logic, you may assume that the execute stage of SBI uses the value stored in the stage register SR2 as DR.
- Enumerate each of the forwarding and stall conditions to receive partial credit in case of an incorrect implementation.

(8 points)

- Many designs possible
- Bug-free not expected
- Must identify the high byte/low byte forwarding cases

**half credit for listing forwarding / stalling cases**

Given solution is partially buggy, but would receive full credit.

*Show your answer on the next page.*
CASES:
1. LBI of mem word
2. LBI high word
3. LBS of mem word
4. LBS of mem word
5. LBI high word
6. Stall during first access

LD signals that third of instruction has been read.
LBS signals the dram-memory access.
### TOMASULO

<table>
<thead>
<tr>
<th>ROB 4 wait 1 cycle in RS</th>
<th>In Reservation Station</th>
<th>Executing</th>
<th>CDB</th>
<th>Committed</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA</td>
<td>1:4</td>
<td>2:2</td>
<td>3:3</td>
<td>4:4</td>
</tr>
<tr>
<td>LDR</td>
<td>2:16</td>
<td>4:14</td>
<td>15:15</td>
<td>16:16</td>
</tr>
<tr>
<td>LDR</td>
<td>3:18</td>
<td>15:16</td>
<td>17:17</td>
<td>18:18</td>
</tr>
<tr>
<td>ADDI</td>
<td>4:19</td>
<td>5:5</td>
<td>6:6</td>
<td>19:19</td>
</tr>
<tr>
<td>MUL</td>
<td>5:27</td>
<td>16:25</td>
<td>26:26</td>
<td>27:27</td>
</tr>
<tr>
<td>MUL</td>
<td>16:37</td>
<td>26:35</td>
<td>36:36</td>
<td>37:37</td>
</tr>
<tr>
<td>ADD</td>
<td>18:39</td>
<td>37:37</td>
<td>38:38</td>
<td>39:39</td>
</tr>
<tr>
<td>STR</td>
<td>19:43</td>
<td>40:41</td>
<td>42:42</td>
<td>43:43</td>
</tr>
<tr>
<td>LDR</td>
<td>37:45</td>
<td>38:39</td>
<td>40:40</td>
<td>45:45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ROB 8 wait 1 cycle in RS</th>
<th>In Reservation Station</th>
<th>Executing</th>
<th>CDB</th>
<th>Committed</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA</td>
<td>1:4</td>
<td>2:2</td>
<td>3:3</td>
<td>4:4</td>
</tr>
<tr>
<td>LDR</td>
<td>2:</td>
<td>4:14</td>
<td>15:15</td>
<td>16:16</td>
</tr>
<tr>
<td>LDR</td>
<td>3:</td>
<td>15:16</td>
<td>17:17</td>
<td>18:18</td>
</tr>
<tr>
<td>ADDI</td>
<td>4:</td>
<td>5:5</td>
<td>6:6</td>
<td>19:19</td>
</tr>
<tr>
<td>MUL</td>
<td>5:</td>
<td>16:25</td>
<td>26:26</td>
<td>27:27</td>
</tr>
<tr>
<td>MUL</td>
<td>6:</td>
<td>26:35</td>
<td>36:36</td>
<td>37:37</td>
</tr>
<tr>
<td>ADD</td>
<td>7:</td>
<td>37:37</td>
<td>38:38</td>
<td>39:39</td>
</tr>
<tr>
<td>STR</td>
<td>16:</td>
<td>39:40</td>
<td>41:41</td>
<td>42:42</td>
</tr>
<tr>
<td>ADDI</td>
<td>19:</td>
<td>20:20</td>
<td>21:21</td>
<td>43:43</td>
</tr>
<tr>
<td>LDR</td>
<td>20:</td>
<td>21:22</td>
<td>23:23</td>
<td>44:44</td>
</tr>
<tr>
<td>ROB 4 forwarding (ok)</td>
<td>In Reservation Station</td>
<td>Executing</td>
<td>CDB</td>
<td>Committed</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------------</td>
<td>-----------</td>
<td>-------</td>
<td>-----------</td>
</tr>
<tr>
<td>LEA</td>
<td>1:4</td>
<td>2:2</td>
<td>3:3</td>
<td>4:4</td>
</tr>
<tr>
<td>LDR</td>
<td>2:15</td>
<td>3:13</td>
<td>14:14</td>
<td>15:15</td>
</tr>
<tr>
<td>LDR</td>
<td>3:17</td>
<td>14:15</td>
<td>16:16</td>
<td>17:17</td>
</tr>
<tr>
<td>ADDI</td>
<td>4:18</td>
<td>5:5</td>
<td>6:6</td>
<td>18:18</td>
</tr>
<tr>
<td>MUL</td>
<td>5:25</td>
<td>14:23</td>
<td>24:24</td>
<td>25:25</td>
</tr>
<tr>
<td>MUL</td>
<td>15:35</td>
<td>24:33</td>
<td>34:34</td>
<td>35:35</td>
</tr>
<tr>
<td>ADD</td>
<td>17:36</td>
<td>34:34</td>
<td>35:35</td>
<td>36:36</td>
</tr>
<tr>
<td>STR</td>
<td>18:38</td>
<td>35:36</td>
<td>37:37</td>
<td>38:38</td>
</tr>
<tr>
<td>LDR</td>
<td>35:40</td>
<td>37:38</td>
<td>39:39</td>
<td>40:40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ROB 8 forwarding</th>
<th>In Reservation Station</th>
<th>Executing</th>
<th>CDB</th>
<th>Committed</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA</td>
<td>1:4</td>
<td>2:2</td>
<td>3:3</td>
<td>4:4</td>
</tr>
<tr>
<td>LDR</td>
<td>2:15</td>
<td>3:13</td>
<td>14:14</td>
<td>15:15</td>
</tr>
<tr>
<td>LDR</td>
<td>3:17</td>
<td>14:15</td>
<td>16:16</td>
<td>17:17</td>
</tr>
<tr>
<td>ADDI</td>
<td>4:18</td>
<td>5:5</td>
<td>6:6</td>
<td>18:18</td>
</tr>
<tr>
<td>MUL</td>
<td>5:25</td>
<td>14:23</td>
<td>24:24</td>
<td>25:25</td>
</tr>
<tr>
<td>MUL</td>
<td>6:35</td>
<td>24:33</td>
<td>34:34</td>
<td>35:35</td>
</tr>
<tr>
<td>ADD</td>
<td>7:36</td>
<td>34:34</td>
<td>35:35</td>
<td>36:36</td>
</tr>
<tr>
<td>STR</td>
<td>15:38</td>
<td>35:36</td>
<td>37:37</td>
<td>38:38</td>
</tr>
<tr>
<td>LDR</td>
<td>19:40</td>
<td>20:21</td>
<td>22:22</td>
<td>40:40</td>
</tr>
</tbody>
</table>
Software Pipelining

(a) How many cycles will this code take? As a simplification, consider only the dominating costs: the latency given above for data cache misses, floating point multiplication, and floating point add. Assume that due to loop iteration and pointer increment overhead, there is no pipelining across iterations.

\[(4 \times N \times \text{sizeof(double)}/64) \times 50 + N \times (10+3) = 19,456\]

(b) To make the code run faster, fuse the loops together. That is, rewrite the code with the same behavior, but only using one loop.

```c
for (int R3 = 0; R3 < N; R3++) {
    *F0 = *F0*(*F1);
    *F0 = *F0 + *F2;
    // or *F0 = (*F0*(*F1)) + *F2;
    F0++;
    F1++;
    F2++;
}
```

(c) Using the same assumptions as in (a), how many cycles will the modified code take?

\[(3 \times N \times \text{sizeof(double)}/64) \times 50 + N \times (10+3)\]

(d) Rewrite the code to take advantage of pipelined arithmetic units (assume no pipelining across iterations). Make sure to only update pointers once per iteration, and make sure the compiled code will not need to do additional pointer arithmetic. Within this constraint, maximize the amount of pipelining being done.

```c
for (int R3 = 4; R3 < any{N-4,N+3}; R3+=8) {
    F0[-4] = (F0[-4]*F1[-4]) + F2[-4];
    F0[-3] = (F0[-3]*F1[-3]) + F2[-3];
    F0[-2] = (F0[-2]*F1[-2]) + F2[-2];
    F0[-1] = (F0[-1]*F1[-1]) + F2[-1];
    F0[0] = (F0[0]*F1[0]) + F2[0];
    F0[1] = (F0[1]*F1[1]) + F2[1];
    F0[2] = (F0[2]*F1[2]) + F2[2];
    F0[3] = (F0[3]*F1[3]) + F2[3];
    F0 += 8;
    F1 += 8;
    F2 += 8;
}
```
(e) How many cycles does the above code take to execute? As in (a), consider only the
three dominating latencies. Assume a perfect out of order machine:

- Each execution unit is perfectly pipelined (one set of operands consumed each
cycle).
- There is only one of each type of execution unit (memory unit has its own adder).
- Infinite issue width
- No speculative execution across a branch
- Dependencies must still be obeyed
- Ignore commit latency including data bus contention
- Ignore reservation station limitations

\[
(3*\text{N*sizeof(double)/64})*50 + (\text{N/8})*(10 + 7 + 3)
\]