1 Branch Prediction [15 points]

(a) What is the prediction accuracy for each of the three branches using a per-branch last-time predictor (assume that every per-branch counter starts at “not-taken”)? Please show all of your work.

Solution:

BRANCH 1:
998/1000 = 99.8%. The branch is mispredicted the first time it’s executed.

BRANCH 2:
500/1000 × 100 = 50%.

BRANCH 3:
0%. The branch changes direction every time it’s executed.

(b) What is the prediction accuracy for each of the three branches when a per-branch 2-bit saturating counter-based predictor is used (assume that every per-branch counter starts at “strongly not-taken”)? Please show all of your work.

Solution:

BRANCH 1:
997/1000 × 100 = 99.7%. The branch is mispredicted the first two times it’s executed and the last time (when the loop exits).

BRANCH 2:
750/1000 × 100 = 75%. The branch repeats the pattern T N N N T N N N ... The saturating counter moves between “strongly not-taken” and “weakly not-taken” (once out of every four predictions, after the branch is actually taken), and the prediction is always not-taken.

BRANCH 3:
500/1000 × 100 = 50%. The branch repeats the pattern T N T N ... The saturating counter moves between “strongly not-taken” and “weakly not-taken” every prediction, and every prediction is not-taken, which is correct half the time.

(c) What is the prediction accuracy for both Branch 2 and Branch 3, when the counter starts at (i) “weakly not-taken” and (ii) “weakly taken”?

Solution:

(i) “weakly not-taken”?

BRANCH 2:
749/1000 × 100 = 74.9%.

BRANCH 3:
0%. The predictor oscillates between “weakly not-taken” and “weakly taken”

(ii) “weakly taken”?

BRANCH 2:
749/1000 × 100 = 74.9%. The branch’s pattern is T N N N T N N N ... The first four iterations, the predictor’s counter starts at weakly taken, moves to strongly taken, then weakly taken, then weakly not-taken. It thus predicts T T T N. Starting for the next group of four branches, the counter is strongly
not-taken, weakly not-taken, strongly not-taken, strongly not-taken, yielding N N N N predictions. Thus
249 * 3 + 2 = 749 correct predictions are made.

BRANCH 3:
500/1000 * 100 = 50%. The branch pattern is T N T N ... The counter is at weakly-taken for the first
branch, moves to strongly-taken, then weakly taken, etc. Thus the predictions are T T T T ... so the
predictor is 50% accurate.

Note that for BRANCH 3, the prediction accuracy is strongly dependent on the initial
state of the branch predictor.

(d) What is the prediction accuracy for each of the three branches when a two-level global branch predictor
with a two-bit global history register and a separate pattern history table per branch, consisting of 2-bit
saturating counters for every entry, is used? Assume that both bits of the global history register are
initialized to “not-taken” and the 2-bit saturating counters in the pattern history tables are initialized
to “strongly not-taken”. When calculating prediction accuracy, ignore the first 500 loop iterations.

Solution:
BRANCH 1:
499/500 * 100 = 99.8%. The predictor has already entered a “strongly taken” state for all global histories
for this branch after 500 loop iterations. It always predicts taken. Only the last loop iteration (for which
the loop branch is not taken) results in a misprediction.

BRANCH 2:
75%. Correlation between BRANCH 3 (of the previous loop iteration) and BRANCH 2 helps out here.
The global branch history will include BRANCH 3’s last result, as well as the loop branch, which will
always be taken. There is thus effectively a separate saturating counter for even loop iterations and odd
loop iterations. When the last BRANCH 3 branch was not taken, BRANCH 2’s pattern is N T N T ...
There are never two consecutive T’s in this subsequence so the saturating counter oscillates between
strongly not-taken and weakly-not taken, resulting in all not-taken predictions and 50% accuracy. When
the last BRANCH 3 branch was taken, BRANCH 2’s pattern is N N N N ... which will result in 100%
accuracy. Hence the overall accuracy is 75%.

BRANCH 3:
75%. Similar to the above, BRANCH 2’s branch history correlates with BRANCH 3’s, and BRANCH
3 effectively uses two saturating counters, conditional on IF BRANCH 2’s outcome. When BRANCH 2
is not taken, BRANCH 3’s pattern is N T N N T N T N ... (these branches result from iterations 1,
2, 3, 5, 6, 7, 9, 10, 11, ...). The predictor will oscillate between weakly and strongly not-taken, but will
always predict N. When BRANCH 2 is taken, BRANCH 3 is also always taken, so the predictor will
have 100% accuracy in this case. Altogether, only 1 out of every 4 iterations is mispredicted, so overall
accuracy is 75%.

2 Branch Prediction [20 points]
Suppose we have the following loop executing on a pipelined LC-3b machine.

\[
\begin{align*}
\text{DOIT} & \quad \text{STW} & \quad R1 \leftarrow R6, \#0 \\
& \quad \text{ADD} & \quad R6 \leftarrow R6, \#2 \\
& \quad \text{AND} & \quad R3 \leftarrow R1, R2 \\
& \quad \text{BRz} & \quad \text{EVEN} \\
& \quad \text{ADD} & \quad R1 \leftarrow R1, \#3 \\
& \quad \text{ADD} & \quad R5 \leftarrow R5, \#-1 \\
& \quad \text{BRp} & \quad \text{DOIT} \\
\text{EVEN} & \quad \text{ADD} & \quad R1 \leftarrow R1, \#1 \\
& \quad \text{ADD} & \quad R7 \leftarrow R7, \#-1 \\
& \quad \text{BRp} & \quad \text{DOIT}
\end{align*}
\]
Assume that before the loop starts, the registers have the following decimal values stored in them:

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0</td>
</tr>
<tr>
<td>R1</td>
<td>0</td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td>0</td>
</tr>
<tr>
<td>R4</td>
<td>0</td>
</tr>
<tr>
<td>R5</td>
<td>5</td>
</tr>
<tr>
<td>R6</td>
<td>4000</td>
</tr>
<tr>
<td>R7</td>
<td>5</td>
</tr>
</tbody>
</table>

The fetch stage takes one cycle, the decode stage also takes one cycle, the execute stage takes a variable number of cycles depending on the type of instruction (see below), and the store stage takes one cycle.

All execution units (including the load/store unit) are fully pipelined and the following instructions that use these units take the indicated number of cycles:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Number of Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>STW</td>
<td>3</td>
</tr>
<tr>
<td>ADD</td>
<td>2</td>
</tr>
<tr>
<td>AND</td>
<td>3</td>
</tr>
<tr>
<td>BR</td>
<td>1</td>
</tr>
</tbody>
</table>

Data forwarding is used wherever possible. Instructions that are dependent on the previous instructions can make use of the results produced right after the previous instruction finishes the execute stage.

The target instruction after a branch can be fetched when the BR instruction is in ST stage. For example, the execution of an AND instruction followed by a BR would look like:

```
AND       F | D | E1 | E2 | E3 | ST
BR         F | D | -  | -  | E1 | ST
TARGET     F | D
```

A scoreboarding mechanism is used.

Answer the following questions:

1. How many cycles does the above loop take to execute if no branch prediction is used (the pipeline stalls on fetching a branch instruction, until it is resolved)?

   **Solution:**
   The first iteration of the DOIT loop takes 15 cycles as shown below:

   F | D | E1 | E2 | E3 | ST |
   F | D | E1 | E2 | ST |
   F | D | E1 | E2 | E3 | ST |
   F | D | -  | -  | E1 | ST |
   F | D | E1 | E2 | ST |
   F | D | E1 | E2 | ST |
   F | D | -  | E1 | ST |

   The rest of the iterations each take 14 cycles, as the fetch cycle of the STW instruction can be overlapped with the ST stage of the BRp DOIT branch.
There are 9 iterations in all as the loop execution ends when R7 is zero and R5 is one.

Total number of cycles = $15 + (14 \times 8) = 127$ cycles

2. How many cycles does the above loop take to execute if all branches are predicted with 100% accuracy?

**Solution:**
The first iteration of the DOIT loop takes 13 cycles as shown below:

```
  F | D | E1 | E2 | E3 | ST |
  F | D | E1 | E2 | ST |
  F | D | E1 | E2 | E3 | ST |
  F | D | - | - | E1 | ST |
  F | - | - | D | E1 | E2 | ST |
  F | D | E1 | E2 | ST |
  F | D | - | E1 | ST |
```

The rest of the iterations each take 10 cycles, as the first three stages of the STW instruction can be overlapped with the execution of the BRp DOIT branch instruction.

Total number of cycles = $13 + (10 \times 8) = 93$ cycles

3. How many cycles does the above loop take to execute if a static BTFN (backward taken-forward not taken) branch prediction scheme is used to predict branch directions? What is the overall branch prediction accuracy? What is the prediction accuracy for each branch?

**Solution:**
The first iteration of the DOIT loop takes 15 cycles as the BRz EVEN branch is predicted wrong the first time.

```
  F | D | E1 | E2 | E3 | ST |
  F | D | E1 | E2 | ST |
  F | D | E1 | E2 | E3 | ST |
  F | D | - | - | E1 | ST |
  F | D | E1 | E2 | ST |
  F | D | E1 | E2 | ST |
  F | D | - | E1 | ST |
```

Of the remaining iterations, the BRz EVEN branch is predicted right 4 times, while it is mispredicted the remaining four times.

The DOIT branch is predicted right all times.

Number of cycles taken by an iteration when the BRz EVEN branch is predicted right = 10 cycles
Number of cycles taken by an iteration when the BRz EVEN branch is not predicted right = 12 cycles

Total number of cycles = $15 + (10 \times 4) + (12 \times 4) = 103$ cycles

The BRz EVEN branch is mispredicted 5 times out of 9. So, the prediction accuracy is $4/9$.
The first BRp DOIT branch is predicted right 4 times out of 4. So, the prediction accuracy is $4/4$.
The second BRp DOIT branch is predicted right 4 times out of 5. So, the prediction accuracy is $4/5$.

Therefore the overall prediction accuracy is $12/18$.  

4/12
3 Scoreboarding [10 points]

Conventional scoreboarding as discussed in class sometimes introduces stalls when they might not actually be necessary. Consider the following code sequence:

\[
\text{ADD R2} \leftarrow \text{R1, R3} \\
\text{ADD R2} \leftarrow \text{R4, R5}
\]

Answer the following questions:

1. Why would the second instruction in this sequence stall in a conventional scoreboarding processor? (Hint: what can’t the scoreboard table track?)

**Solution:**
The first ADD instruction sets its destination register (R2) to invalid in the scoreboard. The second ADD checks the scoreboard and stalls when it finds that its destination register is invalid.

Why should the second ADD stall?

The second ADD stalls because the scoreboard has only one valid bit per register and can track only one instruction writing to a register. If the second ADD were allowed to proceed without stalling, the register R2 would become valid after the first ADD completes execution. A subsequent instruction that sources R2 could potentially get uninstalled after this happens and -incorrectly- source the value of R2 from the first ADD, when it should actually use the new value of R2 from the second ADD.

2. Propose a scoreboard-based dependency check mechanism that addresses this issue. Show the structure of the new scoreboard table clearly, show the conditions for stalling an instruction (and in which stage this stall occurs) due to dependences, and show how the table is updated by instructions as they pass through the pipeline stages.

**Solution 1: Valid Counter**

<table>
<thead>
<tr>
<th>Valid Counter</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R0</td>
</tr>
<tr>
<td></td>
<td>R1</td>
</tr>
<tr>
<td></td>
<td>R2</td>
</tr>
</tbody>
</table>

The proposed scoreboard has a valid counter of size \(\log_2\) (Number of pipeline stages - Number of fetch stages), for each register, instead of a single valid bit. This is because there can be at most one instruction in every post decode stage, writing to the same register.

An instruction updates the scoreboard and executes as follows, in its different stages:

- **Decode:**
  Check the scoreboard to determine if the valid counter of the source registers is zero. If yes, increment the valid counter of the destination register and proceed to execute. If no, stall.
- **Writeback:**
  Decrement the valid counter of the destination register.
Note that this solution would work only when write backs happen in-order.

**Solution 2: Tagging**

Extend the register file entry with a tag. Still keep the valid bit.

Each instruction is assigned a tag.

On **decode**, the instruction writes its tag to the destination register’s entry and sets the valid bit to 0.

On **execute**, the instruction writes to the register and sets the valid bit only if its tag matches the tag in the register file entry.

This solution works when write backs happen out-of-order too.

4 **Interference in Two-Level Branch Predictors** [20 points]

Assume a two-level global predictor with a global history register and a single pattern history table shared by all branches (call this “predictor A”).

1. We call the notion of different branches mapping to the same locations in a branch predictor "branch interference”. Where do different branches interfere with each other in these structures?

   **Solution:**
   
   Global history register (GHR), Pattern history table (PHT)

2. Compared to a two-level global predictor with a global history register and a separate pattern history table for each branch (call this “predictor B”),

   (a) When does predictor A yield lower prediction accuracy than predictor B? Explain. Give a concrete example. If you wish, you can write source code to demonstrate a case where predictor A has lower accuracy than predictor B.

   **Solution:**
   
   Predictor A yields lower prediction accuracy when two branches going in opposite directions are mapped to the same PHT entry. Consider the case of a branch B1 which is always-taken for a given global history. If branch B1 had its own PHT, it would always be predicted correctly. Now, consider a branch B2 which is always-not-taken for the same history. If branch B2 had its own PHT, it would also be predicted right always. However, if branches B1 and B2 shared a PHT, they would map to the same PHT entry and hence, interfere with each other and degrade each other’s prediction accuracy.

   Consider a case when the global history register is 3 bits wide and indexes into a 8-entry pattern history table and the following code segment:

   ```
   for (i = 0; i < 1000; i++)
   {
     if (i % 2 == 0) //IF CONDITION 1
     {
       ........
     }

     if (i % 3 == 0) // IF CONDITION 2
     {
       ........
     }
   }
   ```

   For a global history of “NTN”, IF CONDITION 1 is taken, while IF CONDITION 2 is not-taken. This causes destructive interference in the PHT.
(b) Could predictor A yield higher prediction accuracy than predictor B? Explain how. Give a concrete example. If you wish, you can write source code to demonstrate this case.

**Solution:**
This can happen if the predictions for a branch B1 for a given history become more accurate when another branch B2 maps to the same PHT entry whereas the predictions would not have been accurate had the branch had its own PHT. Consider the case in which branch B1 is always mispredicted for a given global history (when it has its own PHT) because it happens to oscillate between taken and not taken for that history. Now consider an always-taken branch B2 mapping to the same PHT entry. This could improve the prediction accuracy of branch B1 because now B1 could always be predicted taken since B2 is always taken. This may not degrade the prediction accuracy of B2 if B2 is more frequently executed than B1 for the same history. Hence, overall prediction accuracy would improve.

Consider a 2-bit global history register and the following code segment.

```c
if (cond1) { }
if (cond2) { }
if ((a % 4) == 0) {} //BRANCH 1
if (cond1) { }
if (cond2) { }
if ((a % 2) == 0) {} //BRANCH 2
```

BRANCH 2 is strongly correlated with BRANCH 1, because when BRANCH 1 is taken, BRANCH 2 is always taken. Furthermore, the two branches have the same history leading up to them. Therefore, BRANCH 2 can be predicted accurately based on the outcome of BRANCH 1, even if BRANCH 2 has not been seen before.

(c) Is there a case where branch interference in predictor structures does not impact prediction accuracy? Explain. Give a concrete example. If you wish, you can write source code to demonstrate this case as well.

**Solution:**
Predictor A and B yield the same prediction accuracy when two branches going in the same direction are mapped to the same PHT entry. In this case, the interference between the branches does not impact prediction accuracy. Consider two branches B1 and B2 which are always-taken for a certain global history. The prediction accuracy would be the same regardless of whether B1 and B2 have their own PHTs or share a PHT.

Consider a case when the global history register is 3 bits wide and indexes into a 8 entry pattern history table and the following code segment:

```c
for (i = 0; i < 1000; i += 2) //LOOP BRANCH
{
    if (i % 2 == 0) //IF CONDITION
    {
        ........
    }
}
```

LOOP BRANCH and IF CONDITION are both taken for a history of “TTT”. Therefore, although these two branches map to the same location in the pattern history table, the interference between them does not impact prediction accuracy.
5 Branch Prediction vs Predication [30 points]

Consider two machines A and B with 15-stage pipelines with the following stages.

- Fetch (one stage)
- Decode (eight stages)
- Execute (five stages).
- Write-back (one stage).

Both machines do full data forwarding on flow dependences. Flow dependences are detected in the last stage of decode and instructions are stalled in the last stage of decode on detection of a flow dependence.

Machine A has a branch predictor that has a prediction accuracy of P%. The branch direction/target is resolved in the last stage of execute.

Machine B employs predicated execution, similar to what we saw in lecture.

1. Consider the following code segment executing on Machine A:

```
add r3 <- r1, r2
sub r5 <- r6, r7
beq r3, r5, X
addi r10 <- r1, 5
add r12 <- r7, r2
add r14 <- r11, r9
X: addi r15 <- r2, 10
.....
```

When converted to predicated code on machine B, it looks like this:

```
add r3 <- r1, r2
sub r5 <- r6, r7
cmp r3, r5
addi.ne r10 <- r1, 5
add.ne r12 <- r7, r2
add.ne r14 <- r11, r9
addi r15 <- r2, 10
.....
```

(Assume that the condition codes are set by the “cmp” instruction and used by each predicated “.ne” instruction. Condition codes are evaluated in the last stage of execute and can be forwarded like any other data value.)

This segment is repeated several hundreds of times in the code. The branch is taken 40% of the time and not taken 60% of the time. On average, for what range of P would you expect machine A to have a higher instruction throughput than machine B?

**Solution:**
This question illustrates the trade-off between misprediction penalty on a machine with branch prediction and the wasted cycles from executing useless instructions on a machine with predication.

This is one solution with the following assumptions:

- Machines A and B have separate (pipelined) branch/compare and add execution units. So, an add instruction can execute when a branch/compare instruction is stalled.
- Writebacks happen in-order.
When a predicated instruction is discovered to be useless (following the evaluation of the cmp instruction), it still goes through the remaining pipeline stages as nops.

There are several possible right answers for this question, based on the assumptions you make.

On machine A, when the beq r3, r5, X branch is not-taken and predicted correctly, the execution timeline is as follows:

When the branch is taken and predicted correctly, the execution timeline is as follows:

Machine A encounters a misprediction penalty of 17 cycles (8 decode stages + 5 execution stages + 4 stall cycles) on a branch misprediction (regardless of whether the branch is taken or not-taken).

Machine B’s execution timeline is exactly the same as machine A’s timeline with correct prediction, when the branch is not-taken. However, when the branch is taken (cmp evaluates to equal) machine B wastes three cycles as shown below.

Therefore, machine A has higher instruction throughput than machine B if the cost of misprediction is lower than the wasted cycles from executing useless instructions.

Therefore, for \( P > 0.9294 \), machine A has higher instruction throughput than machine B.

2. Consider another code segment executing on Machine A:

\[
\begin{align*}
\text{add r3 <- r1, r2} & \quad \text{F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|} \\
\text{sub r5 <- r6, r7} & \quad \text{F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|} \\
\text{beq r3, r5, X} & \quad \text{F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|} \\
\text{addi r10 <- r1, 5} & \quad \text{F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|} \\
\text{add r12 <- r7, r2} & \quad \text{F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|} \\
\text{add r14 <- r11, r9} & \quad \text{F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|} \\
\text{addi r15 <- r2, 10} & \quad \text{F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|} \\
\text{X: add i r15 <- r14, 10} & \quad \text{F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|} \\
\end{align*}
\]
When converted to predicated code on machine B, it looks like this:

```
add r3 <- r1, r2
sub r5 <- r6, r7
cmp r3, r5
addi.ne r10 <- r1, 5
add.ne r12 <- r10, r2
add.ne r14 <- r12, r9
addi r15 <- r14, 10
```

(Assume that the condition codes are set by the “cmp” instruction and used by each predicated “.ne” instruction. Condition codes are evaluated in the last stage of execute and can be forwarded like any other data value.)

This segment is repeated several hundreds of times in the code. The branch is taken 40% of the time and not taken 60% of the time. On average, for what range of P would you expect machine A to have a higher instruction throughput than machine B?

**Solution:**

On machine A, when the beq r3, r5, X branch is not-taken and predicted correctly, the execution timeline is as follows:

```
add r3 <- r1, r2 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
sub r5 <- r6, r7 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
beq r3, r5, X F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
addi r10 <- r1, 5 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
add r12 <- r10, r2 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
add r14 <- r12, r9 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
addi r15 <- r14, 10 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
```

When the branch is taken and predicted correctly, the execution timeline is as follows:

```
add r3 <- r1, r2 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
sub r5 <- r6, r7 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
cmp r3, r5 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
addi.ne r10 <- r1, 5 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
add.ne r12 <- r10, r2 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
add.ne r14 <- r12, r9 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
addi r15 <- r14, 10 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
```

Machine A encounters a misprediction penalty of 17 cycles (8 decode stages + 5 execution stages + 4 stall cycles) on a branch misprediction (regardless of whether the branch is taken or not-taken).

Machine B’s execution timeline is exactly the same as machine A’s timeline with correct prediction, when the branch is not-taken. However, when the branch is taken (cmp evaluates to equal) machine B wastes eleven cycles as shown below.

```
add r3 <- r1, r2 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
sub r5 <- r6, r7 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
cmp r3, r5 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
addi.ne r10 <- r1, 5 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
add.ne r12 <- r10, r2 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
addi r15 <- r14, 10 F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
```
Machine A has higher instruction throughput than machine B if the cost of misprediction is lower than the wasted cycles from executing useless instructions.

\[(1 - P) \times 17 < 11 \times 0.4\]

Therefore, for \(P > 0.7411\), machine A has higher instruction throughput than machine B.

### 6 Out-of-order Execution [30 points]

A five instruction sequence executes according to Tomasulo’s algorithm. Each instruction is of the form ADD DR,SR1,SR2 or MUL DR,SR1,SR2. ADDs are pipelined and take 9 cycles (F-D-E1-E2-E3-E4-E5-WB). MULs are also pipelined and take 11 cycles (two extra execute stages). An instruction must wait until a result is in a register before it sources it (reads it as a source operand). For instance, if instruction 2 has a read-after-write dependence on instruction 1, instruction 2 can start executing in the next cycle after instruction 1 writes back (shown below).

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>F</th>
<th>D</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
<th>...</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 2</td>
<td>F</td>
<td>D</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td>E1</td>
</tr>
</tbody>
</table>

The machine can fetch one instruction per cycle, and can decode one instruction per cycle.

The register file before and after the sequence are shown below.

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Value</th>
<th>Valid</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>1</td>
<td>4</td>
<td>R0</td>
<td>1</td>
<td>310</td>
</tr>
<tr>
<td>R1</td>
<td>1</td>
<td>5</td>
<td>R1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td>6</td>
<td>R2</td>
<td>1</td>
<td>410</td>
</tr>
<tr>
<td>R3</td>
<td>1</td>
<td>7</td>
<td>R3</td>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>R4</td>
<td>1</td>
<td>8</td>
<td>R4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>R5</td>
<td>1</td>
<td>9</td>
<td>R5</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>R6</td>
<td>1</td>
<td>10</td>
<td>R6</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>R7</td>
<td>1</td>
<td>11</td>
<td>R7</td>
<td>1</td>
<td>21</td>
</tr>
</tbody>
</table>

(a) Before (b) After

(a) Complete the five instruction sequence in program order in the space below. Note that we have helped you by giving you the opcode and two source operand addresses for the fourth instruction. (The program sequence is unique.)

Give instructions in the following format: “opcode destination \(\Leftarrow\) source1, source2.”

**Solution:**

- ADD R7 \(\Leftarrow\) R6, R7
- ADD R3 \(\Leftarrow\) R6, R7
- MUL R0 \(\Leftarrow\) R3, R6
- MUL R2 \(\Leftarrow\) R6, R6
- ADD R2 \(\Leftarrow\) R0, R2

(b) In each cycle, a single instruction is fetched and a single instruction is decoded.

Assume the reservation stations are all initially empty. Put each instruction into the next available reservation station. For example, the first ADD goes into “a”. The first MUL goes into “x”. Instructions
remain in the reservation stations until they are completed. Show the state of the reservation stations at the end of cycle 8.

Note: to make it easier for the grader, when allocating source registers to reservation stations, please always have the higher numbered register be assigned to source2

**Solution:**

(c) Show the state of the Register Alias Table (Valid, Tag, Value) at the end of cycle 8.

**Solution:**

<table>
<thead>
<tr>
<th></th>
<th>Valid</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0</td>
<td>x</td>
<td>4</td>
</tr>
<tr>
<td>R1</td>
<td>1</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>R2</td>
<td>0</td>
<td>c</td>
<td>6</td>
</tr>
<tr>
<td>R3</td>
<td>0</td>
<td>b</td>
<td>7</td>
</tr>
<tr>
<td>R4</td>
<td>1</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>R5</td>
<td>1</td>
<td>-</td>
<td>9</td>
</tr>
<tr>
<td>R6</td>
<td>1</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>R7</td>
<td>0</td>
<td>a</td>
<td>11</td>
</tr>
</tbody>
</table>

7 Dataflow [15 points]

**Solution:**

\( K^N \) when \( N \geq 0 \)