CMU 18-447 Introduction to Computer Architecture, Spring 2012
Handout 11/ Solutions to HW 4: Pipelining

Prof. Onur Mutlu, Instructor
Chris Fallin, Lavanya Subramanian, Abeer Agrawal, TAs

Given: Monday, Feb 27, 2012
Due: **Monday, Mar 19, 2012**

1 Delay Slots

A machine has a five-stage pipeline consisting of fetch, decode, execute, mem and write-back stages. The machine uses delay slots to handle control dependences. Jump targets, branch targets and destinations are resolved in the execute stage.

(a) What is the number of delay slots needed to ensure correct operation?

2

(b) Which instruction(s) in the assembly sequences below would you place in the delay slot(s), assuming the number of delay slots you answered for part(a)? Clearly rewrite the code with the appropriate instruction(s) in the delay slot(s).

(a)
```
ADD R5 <- R4, R3
OR  R3 <- R1, R2
SUB R7 <- R5, R6
J   X
```
Delay Slots
```
LW  R10 <- (R7)
ADD R6 <- R1, R2
X:
```

Solution:

```
ADD R5 <- R4, R3
J   X
OR  R3 <- R1, R2
SUB R7 <- R5, R6
LW  R10 <- (R7)
ADD R6 <- R1, R2
X:
```

(b)
```
ADD R5 <- R4, R3
OR  R3 <- R1, R2
SUB R7 <- R5, R6
BEQ R5 <- R7, X
```
Delay Slots
```
LW  R10 <- (R7)
ADD R6 <- R1, R2
X:
```

Solution:

```
ADD R5 <- R4, R3
SUB R7 <- R5, R6
BEQ R5 <- R7, X
OR  R3 <- R1, R2
NOP
LW  R10 <- (R7)
ADD R6 <- R1, R2
X:
```

(c)
```
ADD R2 <- R4, R3
OR  R5 <- R1, R2
SUB R7 <- R5, R6
BEQ R5 <- R7, X
```
$\boxed{\text{Delay Slots}}$
```
LW  R10 <- (R7)
ADD R6 <- R1, R2
X:
```

Solution:

```
ADD R2 <- R4, R3
OR  R5 <- R1, R2
SUB R7 <- R5, R6
BEQ R5 <- R7, X
NOP
NOP
LW  R10 <- (R7)
ADD R6 <- R1, R2
X:
```

(c) Can you modify the pipeline to reduce the number of delay slots (without introducing branch prediction)? Clearly state your solution and explain why.

Move the resolution of jump targets and branch targets and destinations to the decode stage.
Jumps and branches would get resolved one cycle earlier and hence one delay slot would be enough to ensure correct operation.

2 Hardware vs Software Interlocking

Consider two pipelined machines A and B.

Machine I implements interlocking in hardware. On detection of a flow dependence, it stalls the instruction in the decode stage of the pipeline (blocking fetch/decode of subsequent instructions) until all of the instruction's sources are available. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can access the same register in the next half of the cycle). No other data forwarding is implemented. However, there are two execute units with adders, and independent instructions can be executed in separate execution units and written back out-of-order. There is one write-back stage per execute unit, so an instruction can write-back as soon as it finishes execution.

Machine II does not implement interlocking in hardware. It assumes all instructions are independent and relies on the compiler to order instructions such that there is sufficient distance between dependent instructions. The compiler either moves other independent instructions between two dependent instructions, if it can find such instructions, or otherwise, inserts NOPs. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can access the same register in the next half of the cycle).

Both machines have the following four pipeline stages and two adders.

- Fetch (one clock cycle)
- Decode (one clock cycle)
- Execute (ADD takes 2 clock cycles. Each ADD unit is not pipelined, but an instruction can be executed if an unused execute (ADD) unit is available.)
- Write-back (one clock cycle). There is one write-back stage per execute (ADD) unit.

Consider the following 2 code segments.

Code segment A

```
ADD R5 <- R6, R7
ADD R3 <- R5, R4
ADD R6 <- R3, R8
ADD R9 <- R6, R3
```

Code segment B

```
ADD R3 <- R1, R2
ADD R8 <- R9, R10
ADD R4 <- R5, R6
ADD R7 <- R1, R4
ADD R12 <- R8, R2
```

(a) Calculate the number of cycles it takes to execute each of these two code segments on machines I and II.

Solution:
Code Segment A:

Machine I:
The machine stalls for two cycles after each instruction, as each instruction is dependent on the previous one.

```
F D E E W
  F D     E E W
    F     D     E E W
          F     D     E E W
```

14 cycles

Machine II:
The compiler places nops between every two instuctions as they are dependent and it can't find other independent instructions.

```
ADD R5 <- R6, R7
NOP
NOP
ADD R3 <- R5, R4
NOP
```

```
NOP
ADD R6 <- R3, R8
NOP
NOP
ADD R9 <- R6, R3
```

The execution time line of this modified code segment is as shown below. N indicates a NOP in a pipeline stage.

```
F D E E W
  N N N N N
    N N N N N
      F D E E W
        N N N N N
          N N N N N
            F D E E W
              N N N N N
                N N N N N
                  F D E E W
```

14 cycles

Code Segment B:

Machine I:

The execution timeline is as below:

```
F D E E W
  F D E E W
    F D E E W
      F D     E E W
        F     D E E W
```

11 cycles

Machine II:

For code segment B, the compiler can find independent instructions to place between dependent instructions. The compiler reorders instructions such that each pair of dependent instructions are separated by two independent instructions.

```
ADD R4 <- R5, R6
ADD R8 <- R9, R10
ADD R3 <- R1, R2
ADD R7 <- R1, R4
ADD R12 <- R8, R2
```

The execution timeline for this reordered code is as below:

```
F D E E W
  F D E E W
    F D E E W
      F D E E W
        F D E E W
```

9 cycles

This is assuming that machine I does not implement any instruction reordering. Instructions could be reordered in hardware in machine I too producing similar results as machine II. However, this reodering logic could increase the critical path.

(b) Calculate the machine code size of each of these two code segments on machines I and II, assuming a fixed-length ISA, where each instruction is encoded as 4 bytes.

Solution:
Code Segment A:

Machine I - 16 bytes
Machine II - 40 bytes (because of the additional NOPs)

Code Segment B:

Machine I - 20 bytes
Machine II - 20 bytes (no additional NOPs as compiler was able to find independent instructions)

(c) Which machine takes a smaller number of cycles to execute each code segment A and B?

Solution:
Code Segment A: Both machines take the same number of cycles - 14 cycles
Code Segment B: Machine II takes a smaller number of cycles

(d) Does the machine that takes a smaller number of cycles for code segment A also take a smaller number of cycles than the other machine for code segment B? Why or why not?

Solution:
For code segment A, both machines take the same number of cycles as every instruction is dependent on the previous instruction. Therefore, the compiler cannot find independent instructions to place between dependent ones.

For code segment B, machine II takes a smaller number of cycles as the compiler can find independent instructions to place between dependent instructions.

(e) Would you say that the machine that provides a smaller number of cycles as compared to the other machine has higher performance (taking into account all components of the Iron Law of Performance)?

Solution:
Which machine that takes a smaller number of cycles depends on the code segment, as seen in the previous part of this question. For code segment A both machines take the same number of clock

cycles. However, machine I's cycle time could be longer due to interlocking hardware. Therefore, it could potentially incur a larger execution time.
For code segment B, machine II has higher performance than machine I, as machine II takes a smaller number of cycles to execute and also has simpler hardware than machine I (could potentially have a smaller cycle time).

(f) Which machine incurs lower code size for each code segment A and B?

Solution:
For code segment A, machine I has lower code size than machine II.
For code segment B, both machines have the same code size.

(g) Does the same machine incur lower code sizes for both code segments A and B? Why or why not?

Solution:
No.
For code segment A, machine I has lower code size as the compiler inserts NOPs in machine II.
For code segment B, both machines incur the same code size as the compiler does not need to add additional NOPs.

3 Branch Prediction

Suppose we have the following loop executing on a pipelined LC-3b machine.

```
DOIT     STW   R1 -> R6, #0
         ADD   R6 <- R6, #2
         AND   R3 <- R1, R2
         BRz   EVEN
         ADD   R1 <- R1, #3
         ADD   R5 <- R5, #-1
         BRp   DOIT
EVEN     ADD   R1 <- R1, #1
         ADD   R7 <- R7, #-1
         BRp   DOIT
```

Assume that before the loop starts, the registers have the following decimal values stored in them:

Register	Value
R0	0
R1	0
R2	1
R3	0
R4	0
R5	5
R6	4000
R7	5

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:

Instruction	Number of Cycles
STW	3
ADD	3
AND	2
BR	1

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 ADD instruction followed by a BR would look like:

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

A scoreboarding mechanism is used.

Answer the following questions:

(a) 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 | E3 | ST |
        F  | D  | E1 | E2 | ST |
             F  | D  | -  | E1 | ST |
                                 F  | D  | E1 | E2 | E3 | ST |
                                      F  | D  | E1 | E2 | E3 | 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

(b) 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 | E3 | ST |
        F  | D  | E1 | E2 | ST |
             F  | D  | -  | E1 | ST |
                  F  | -  | D  | E1 | E2 | E3 | ST |
                            F  | D  | E1 | E2 | E3 | 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

(c) 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 | E3 | ST |
        F  | D  | E1 | E2 | ST |
             F  | D  | -  | E1 | ST |
                                 F  | D | E1 | E2 | E3 | ST |
                                      F | D  | E1 | E2 | E3 | 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 Scoreboarding

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

```
ADD R2 <- R1, R3
ADD R2 <- R4, R5
```

Answer the following questions:

(a) Why would the second instruction in this sequence stall in a conventional scoreboarding processor? (Hint: what can't the scoreboarding 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 unstalled 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.

(b) 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

Valid Counter	Register
	R0
	R1
	R2

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.

5 Interference in Two-Level Branch Predictors

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").

(a) 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)

(b) 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"),

(i) 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.

(ii) 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.

```
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.

(iii) 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:

```
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.

6 Branch Prediction vs Predication

Consider two machines A and B with 17-stage pipelines with the following stages.

- Fetch (one stage)
- Decode (nine stages)
- Execute (six 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.

(a) 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 r1 <- 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 50% of the time and not taken 50% 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:

```
add r3 <- r1, r2    F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|WB|
sub r5 <- r6, r7       F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|WB|
beq r3, r5, X             F|D1|D2|D3|D4|D5|D6|D7|D8|D9|- |- |- |- |- |E1|E2|E3|E4|E5|E6|WB|
addi r10 <- r1, 5            F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|- |- |- |- |- |WB|
add r12 <- r7, r2               F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|- |- |- |- |- |WB|
add r1 <- r11, r9                  F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|- |- |- |- |- |WB|
X: addi r15 <- r2, 10                 F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|- |- |- |- |- |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|D9|E1|E2|E3|E4|E5|E6|WB|
sub r5 <- r6, r7       F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|WB|
beq r3, r5, X             F|D1|D2|D3|D4|D5|D6|D7|D8|D9|- |- |- |- |- |E1|E2|E3|E4|E5|E6|WB|
X: addi r15 <- r2, 10        F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|- |- |- |- |- |WB|
.....
```

Machine A encounters a misprediction penalty of 20 cycles (9 decode stages + 6 execution stages + 5 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.

```
add r3 <- r1, r2    F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|WB|
sub r5 <- r6, r7       F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|WB|
cmp r3, r5                F|D1|D2|D3|D4|D5|D6|D7|D8|D9|- |- |- |- |- |E1|E2|E3|E4|E5|E6|WB|
addi.ne r10 <- r1, 5         F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|- |- |- |- |- |WB|
add.ne r12 <- r7, r2            F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|- |- |- |- |- |WB|
add.ne r14 <- r11, r9              F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|- |- |- |- |- |WB|
addi r15 <- r2, 10                    F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|- |- |- |- |- |WB|
.....
```

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.

$(1 - P) \times 20 < 3 \times 0.5$

Therefore, for $P > 0.925$, machine A has higher instruction throughput than machine B.

(b) Consider another code segment executing on Machine A:

```
add r3 <- r1, r2
sub r5 <- r6, r7
beq r3, r5, X
addi r10 <- r1, 5
add r12 <- r10, r2
add r14 <- r12, r9
X: addi r15 <- r14, 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 <- 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 50% of the time and not taken 50% 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|D9|E1|E2|E3|E4|E5|E6|WB|
sub r5 <- r6, r7     F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|WB|
beq r3, r5, X           F|D1|D2|D3|D4|D5|D6|D7|D8|D9|- |- |- |- |- |E1|E2|E3|E4|E5|E6|WB|
addi r10 <- r1, 5          F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|- |- |- |- |- |WB|
add r12 <- r10, r2            F|D1|D2|D3|D4|D5|D6|D7|D8|D9|- |- |- |- |- |E1|E2|E3|E4|E5|E6|WB|
add r14 <- r12, r9               F|D1|D2|D3|D4|D5|D6|D7|D8|- |- |- |- |- |- |- |- |- |- |- |E1|E2|E3|E4|E5|E6|WB|
X: addi r15 <- r14, 10              F|D1|D2|D3|D4|D5|D6|D7|D8|D9|- |- |- |- |- |- |- |- |- |- |- |- |- |- |- |E1|E2|E3|E4|E5|E6|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|D9|E1|E2|E3|E4|E5|E6|WB|
sub r5 <- r6, r7     F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|WB|
cmp r3, r5              F|D1|D2|D3|D4|D5|D6|D7|D8|D9|- |- |- |- |- |E1|E2|E3|E4|E5|E6|WB|
addi r15 <- r14, 10        F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|- |- |- |- |- |WB|
.....
```

Machine A encounters a misprediction penalty of 20 cycles (9 decode stages + 6 execution stages + 5 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 seven cycles as shown below.

```
add r3 <- r1, r2  F|D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|WB|
sub r5 <- r6, r7    F |D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|WB|
cmp r3, r5             F |D1|D2|D3|D4|D5|D6|D7|D8|D9|- |- |- |- |- |- |E1|E2|E3|E4|E5|E6|WB|
addi.ne r10 <- r1, 5      F |D1|D2|D3|D4|D5|D6|D7|D8|D9|E1|E2|E3|E4|E5|E6|- |- |- |- |- |- |WB|
add.ne r12 <- r10, r2        F |D1|D2|D3|D4|D5|D6|D7|D8|D9|- |- |- |- |- |E1|E2|E3|E4|E5|E6|- |WB|
add.ne r14 <- r12, r9           F |D1|D2|D3|D4|D5|D6|D7|D8|- |- |- |- |- |D9|- |- |- |- |E1|E2|E3|E4|E5|E6|WB|
addi r15 <- r14, 10                F |D1|D2|D3|D4|D5|D6|D7|- |- |- |- |- |D8|- |- |- |- |D9|E1|E2|E3|E4|E5|E6|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.

$(1 - P) \times 20 < 7 \times 0.5$

Therefore, for $P > 0.825$, machine A has higher instruction throughput than machine B.