Computer Architecture (263-2210-00L), Fall 2017
HW 3: Branch handling and GPU
SOLUTIONS

Instructor: Prof. Onur Mutlu
TAs: Hasan Hassan, Arash Tavakkol, Mohammad Sadr, Lois Orosa, Juan Gomez Luna

Assigned: Wednesday, Oct 25, 2017
Due: **Wednesday, Nov 8, 2017**

- **Handin - Critical Paper Reviews (1).** You need to submit your reviews to `https://safari.ethz.ch/review/architecture/`. Please check your inbox. You should have received an email with the password you can use to login to the paper review system. If you have not received any email, please contact comparch@lists.ethz.ch. In the first page after login, you should click in "Architecture - Fall 2017 Home", and then go to "any submitted paper" to see the list of papers.
- **Handin - Questions (2-10).** Please upload your solution to the Moodle (`https://moodle-app2.let.ethz.ch/`) as a single PDF file. **Please use a typesetting software (e.g., LaTeX) or a word processor (e.g., MS Word, LibreOfficeWriter) to generate your PDF file. Feel free to draw your diagrams either using an appropriate software or by hand, and include the diagrams into your solutions PDF.**

1 Critical Paper Reviews [150 points]

Please read the following handout on how to write critical reviews. We will give out extra credit that is worth 0.5% of your total grade for each good review.

- Lecture slides on guidelines for reviewing papers. Please follow this format.
  `https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=onur-comparch-f17-how-to-do-the-paper-reviews.pdf`
- Some sample reviews can be found here: `https://safari.ethz.ch/architecture/fall2017/doku.php?id=readings`

(a) Write a one-page critical review for the following paper:
B. C. Lee, E. Ipek, O. Mutlu and D. Burger. "Architecting phase change memory as a scalable DRAM alternative." ISCA 2009. `https://people.inf.ethz.ch/omutlu/pub/pcm_isca09.pdf`

(b) Write a one-page critical review for **two** of the following papers:

- McFarling, Scott. "Combining branch predictors". Vol. 49. Technical Report TN-36, Digital Western Research Laboratory, 1993. `https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=combining.pdf`
- Yeh, Tse-Yu, and Yale N. Patt. "Two-level adaptive training branch prediction." Proceedings of the 24th annual international symposium on Microarchitecture. ACM, 1991. `https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=yeh_patt-adaptive-training-1991.pdf`
- Keckler, S. W., Dally, W. J., Khailany, B., Garland, M., and Glasco, D. "GPUs and the future of parallel computing." IEEE Micro, 2011. `https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=ieee-micro-gpu.pdf`

2 GPUs and SIMD [100 points]

We define the *SIMD utilization* of a program run on a GPU as the fraction of SIMD lanes that are kept busy with *active threads* during the run of a program. As we saw in lecture and practice exercises, the SIMD utilization of a program is computed across the *complete run* of the program.

The following code segment is run on a GPU. Each thread executes **a single iteration** of the shown loop. Assume that the data values of the arrays A, B, and C are already in vector registers so there are no loads and stores in this program. (Hint: Notice that there are 6 instructions in each thread.) A warp in the GPU consists of 64 threads, and there are 64 SIMD lanes in the GPU. Please assume that all values in array B have magnitudes less than 10 (i.e., $|B[i]| < 10$, for all i).

```
for (i = 0; i < 1024; i++) {
  A[i] = B[i] * B[i];
  if (A[i] > 0) {
    C[i] = A[i] * B[i];
    if (C[i] < 0) {
      A[i] = A[i] + 1;
    }
    A[i] = A[i] - 2;
  }
}
```

Please answer the following five questions.

(a) [10 points] How many warps does it take to execute this program?

Warps = (Number of threads) / (Number of threads per warp)
Number of threads = 2^{10} (i.e., one thread per loop iteration).
Number of threads per warp = 64 = 2^6 (given).
Warps = $2^{10}/2^6 = 2^4$

(b) [10 points] What is the maximum possible SIMD utilization of this program?

100%

(c) [30 points] Please describe what needs to be true about array B to reach the maximum possible SIMD utilization asked in part (b). (Please cover all cases in your answer)

B: For every 64 consecutive elements: every value is 0, every value is positive, or every value is negative. Must give all three of these.

(d) [15 points] What is the minimum possible SIMD utilization of this program?

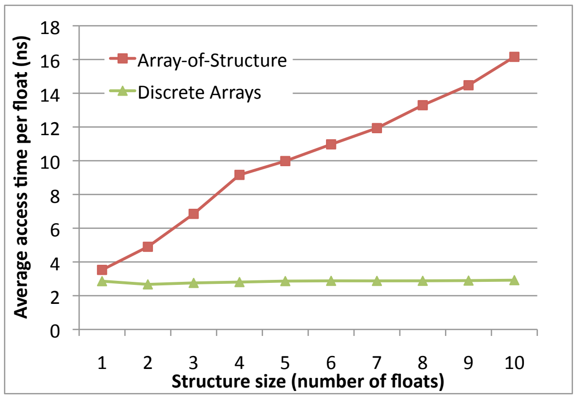
Answer: 132/384
Explanation: The first two lines must be executed by every thread in a warp (64/64 utilization for each line). The minimum utilization results when a single thread from each warp passes both conditions on lines 2 and 4, and every other thread fails to meet the condition on line 2. The thread per warp that meets both conditions, executes lines 3-6 resulting in a SIMD utilization of 1/64 for each line. The minimum SIMD utilization sums to $(64 * 2 + 1 * 4)/(64 * 6) = 132/384$

(e) [35 points] Please describe what needs to be true about array B to reach the minimum possible SIMD utilization asked in part (d). (Please cover all cases in your answer)

B: Exactly 1 of every 64 consecutive elements must be negative. The rest must be zero. This is the only case that this holds true.

3 AoS vs. SoA on GPU [50 points]

The next figure shows the execution time for processing an array of data structures on a GPU. Abscissas represent the number of members in a data structure. Consecutive GPU threads read consecutive structures, and compute the sum reduction of their members. The result is stored in the first member of the structure.



The green line is the time for a kernel that accesses an array that is stored as discrete sub-arrays, that is, all i-th members of all array elements are stored in the i-th sub-array, in consecutive memory locations. The red line is the kernel time with an array that contains members of the same structure in consecutive memory locations.

- Why does the red line increase linearly? Why not the green line?

> GPU global memory accesses are carried out on a per-warp basis. If all threads in the same warp access the same cache line or memory segment, the efficiency is maximal. This is the case of the green line.
> In the A-o-S case, consecutive threads have a stride between them. This increases the number of memory transactions that are necessary for a single warp.

- How can the effect on the red line be alleviated?

> The effect for this kernel could be alleviated by the use of caches that store structure members that will be accessed during the reduction operation.

- How would both kernels perform on a single-core CPU with one level of cache? And on a dual-core CPU with individual caches? And on a dual-core CPU with a shared cache?

> On a single-core CPU, the A-o-S layout benefits from the cache: structure members are cached when the first member is accessed. The DA layout might result in a similar performance, as long as the relation between structure size and cache size allows for enough cache hits per data structure.
> On a dual-core CPU with individual caches, the DA layout might provoke cache conflicts in the output writing. With a shared cache, it is more likely that A-o-S and DA obtain a similar performance.

4 SIMD Processing [50 points]

Suppose we want to design a SIMD engine that can support a vector length of 16. We have two options: a traditional vector processor and a traditional array processor.

Which one is more costly in terms of chip area (circle one)?

The traditional vector processor [✓ The traditional array processor] Neither

Explain:

An array processor requires 16 functional units for an operation whereas a vector processor requires only 1.

Assuming the latency of an addition operation is five cycles in both processors, how long will a `VADD` (vector add) instruction take in each of the processors (assume that the adder can be fully pipelined and is the same for both processors)?

For a vector length of 1:

The traditional vector processor: 5 cycles

The traditional array processor: 5 cycles

For a vector length of 4:

The traditional vector processor: 8 cycles (5 for the first element to complete, 3 for the remaining 3)

The traditional array processor: 5 cycles

For a vector length of 16:

The traditional vector processor: 20 cycles (5 for the first element to complete, 15 for the remaining 15)

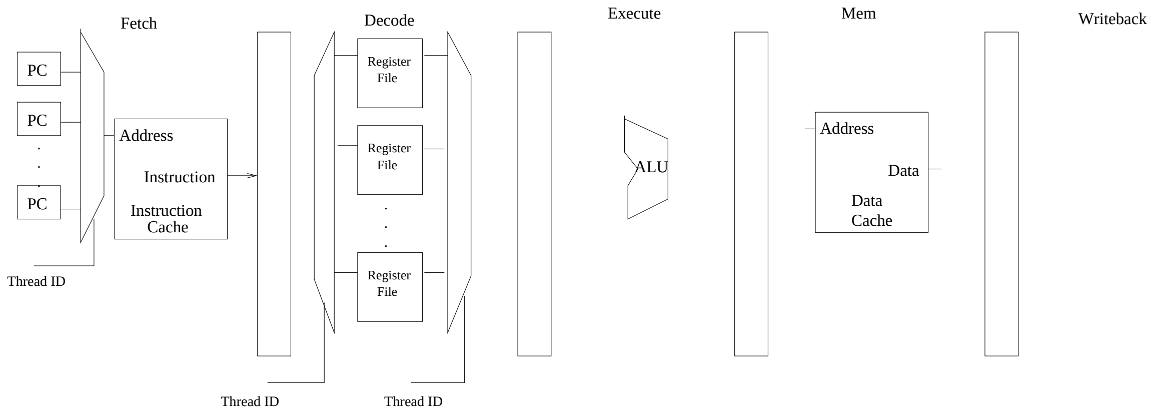
The traditional array processor: 5 cycles

5 Fine-Grained Multithreading [100 points]

Consider a design "Machine I" with five pipeline stages: fetch, decode, execute, memory and writeback. Each stage takes 1 cycle. The instruction and data caches have 100% hit rates (i.e., there is never a stall for a cache miss). Branch directions and targets are resolved in the execute stage. The pipeline stalls when a branch is fetched, until the branch is resolved. Dependency check logic is implemented in the decode stage to detect flow dependences. The pipeline does not have any forwarding paths, so it must stall on detection of a flow dependence.

In order to avoid these stalls, we will consider modifying Machine I to use fine-grained multithreading.

(a) In the five stage pipeline of Machine I shown below, clearly show what blocks you would need to add in each stage of the pipeline, to implement fine-grained multithreading. You can replicate any of the blocks and add muxes. You don't need to implement the mux control logic (although provide an intuitive name for the mux control signal, when applicable).



(b) The machine's designer first focuses on the branch stalls, and decides to use fine-grained multithreading to keep the pipeline busy no matter how many branch stalls occur. What is the minimum number of threads required to achieve this?

3

Why?

Since branches are resolved in the Execute stage, it is necessary that the Fetch stage does not fetch
for a thread until the thread's previous instruction has passed Execute. Hence three threads are
needed to cover Fetch, Decode, Execute.

(c) The machine's designer now decides to eliminate dependency-check logic and remove the need for flow-dependence stalls (while still avoiding branch stalls). How many threads are needed to ensure that no flow dependence ever occurs in the pipeline?

4

Why?

The designer must ensure that when an instruction is in Writeback, the next instruction in the same
thread has not reached Decode yet. Hence, at least 4 threads are needed.

A rival designer is impressed by the throughput improvements and the reduction in complexity that FGMT brought to Machine I. This designer decides to implement FGMT on another machine, Machine II. Machine II is a pipelined machine with the following stages.

Fetch	1 stage
Decode	1 stage
Execute	8 stages (branch direction/target are resolved in the first execute stage)
Memory	2 stages
Writeback	1 stage

Assume everything else in Machine II is the same as in Machine I.

(d) Is the number of threads required to eliminate branch-related stalls in Machine II the same as in Machine I?

[✓ **YES**] **NO** (Circle one)

If yes, why?

Branches are resolved at the third pipeline stage in both machines, and distance from fetch to branch
resolution determines the minimum number of threads to avoid branch stalls.

If no, how many threads are required?

N/A

(e) What is the minimum CPI (i.e., maximum performance) of each thread in Machine II when this minimum number of threads is used?

3 (if no flow dependence stalls occur)

(f) Now consider flow-dependence stalls. Does Machine II require the same minimum number of threads as Machine I to avoid the need for flow-dependence stalls?

YES [✓ **NO**] (Circle one)

If yes, why?

N/A

If no, how many threads are required?

12 (the Decode, Execute 1 – 8, Memory, and Writeback stages must all have instructions from independent threads.)

(g) What is the minimum CPI of each thread when this number of threads (to cover flow-dependence stalls) is used?

12

(h) After implementing fine grained multithreading, the designer of Machine II optimizes the design and compares the pipeline throughput of the original Machine II (without FGMT) and the modified Machine II (with FGMT) both machines operating at their maximum possible frequency, for several code sequences. On a particular sequence that has no flow dependences, the designer is surprised to see that the new Machine II (with FGMT) has lower overall throughput (number of instructions retired by the pipeline per second) than the old Machine II (with no FGMT). Why could this be? Explain concretely.

The additional FGMT-related logic (MUXes and thread selection logic) could increase the critical
path length, which will reduce maximum frequency and thus performance.

6 Multithreading [50 points]

Suppose your friend designed the following fine-grained multithreaded machine:

- The pipeline has 22 stages and is 1 instruction wide.
- Branches are resolved at the end of the 18th stage and there is a 1 cycle delay after that to communicate the branch target to the fetch stage.
- The data cache is accessed during stage 20. On a hit, the thread does not stall. On a miss, the thread stalls for 100 cycles, fixed. The cache is non-blocking and has space to accommodate 16 outstanding requests
- The number of hardware contexts is 200

Assuming that there are always enough threads present, answer the following questions:

(a) Can the pipeline **always** be kept full and non-stalling? Why or why not? *(Hint: think about the worst case execution characteristics)*

Circle one: **YES** [✓ **NO**]

NO - will stall when more than 16 outstanding misses in pipe

(b) Can the pipeline **always** be kept full and non-stalling if all accesses hit in the cache? Why or why not?
Circle one: [✓ **YES**] **NO**

YES - switching between 200 threads is plenty to avoid stalls due to branch prediction delay

(c) Assume that all accesses hit in the cache and your friend wants to keep the pipeline **always** full and non-stalling. How would you adjust the hardware resources (if necessary) to satisfy this while minimizing hardware cost? You cannot change the latencies provided above. Be comprehensive and specific with numerical answers. If nothing is necessary, justify why this is the case.

Reduce hardware thread contexts to 19, the minimum to keep pipe full/non-stalling

(d) Assume that all accesses miss in the cache and your friend wants to keep the pipeline **always** full and non-stalling. How would you adjust the hardware resources (if necessary) to satisfy this while minimizing hardware cost? You cannot change the latencies provided above. Be comprehensive and specific with numerical answers. If nothing is necessary, justify why this is the case.

Reduce hardware thread contexts to 100, the minimum to keep pipe full/non-stalling. Increase capability to support 100 outstanding misses

7 Branch Prediction [100 points]

Assume the following piece of code that iterates through a large array populated with **completely (i.e., truly) random** positive integers. The code has four branches (labeled B1, B2, B3, and B4). When we say that a branch is *taken*, we mean that the code *inside* the curly brackets is executed.

```
for (int i=0; i<N; i++) { /* B1 */
  val = array[i];        /* TAKEN PATH for B1 */
  if (val % 2 == 0) {    /* B2 */
    sum += val;        /* TAKEN PATH for B2 */
  }
  if (val % 3 == 0) {    /* B3 */
    sum += val;        /* TAKEN PATH for B3 */
  }
  if (val % 6 == 0) {    /* B4 */
    sum += val;        /* TAKEN PATH for B4 */
  }
}
```

(a) Of the four branches, list all those that exhibit *local correlation*, if any.

Only B1.

B2, B3, B4 are not locally correlated. Just like consecutive outcomes of a die, an element being a multiple of N (N is 2, 3, and 6, respectively for B2, B3, and B4) has no bearing on whether the next element is also a multiple of N.

(b) Which of the four branches are *globally correlated*, if any? Explain in less than 20 words.

B4 is correlated with B2 and B3. 6 is a common multiple of 2 and 3.

Now assume that the above piece of code is running on a processor that has a global branch predictor. The global branch predictor has the following characteristics.

- Global history register (GHR): 2 bits.
- Pattern history table (PHT): 4 entries.
- Pattern history table entry (PHTE): 11-bit signed saturating counter (possible values: -1024–1023)
- Before the code is run, all PHTEs are initially set to 0.
- As the code is being run, a PHTE is incremented (by one) whenever a branch that corresponds to that PHTE is taken, whereas a PHTE is decremented (by one) whenever a branch that corresponds to that PHTE is not taken.

(d) After 120 iterations of the loop, calculate the **expected** value for only the first PHTE and fill it in the shaded box below. (Please write it as a base-10 value, rounded to the nearest one's digit.)

Hint. For a given iteration of the loop, first consider, what is the probability that both B1 and B2 are taken? Given that they are, what is the probability that B3 will increment or decrement the PHTE? Then consider...

Show your work.

Without loss of generality, let's take a look at the numbers from 1 through 6. Given that a number is a multiple of two (i.e., 2, 4, 6), the probability that the number is also a multiple of three (i.e., 6) is equal to 1/3, let's call this value Q. Given that a number is a multiple of two and three (i.e., 6), the probability that the number is also a multiple of six (i.e., 6) is equal to 1, let's call this value R.

For a **single** iteration of the loop, the PHTE has four chances of being incremented/decremented, once at each branch.

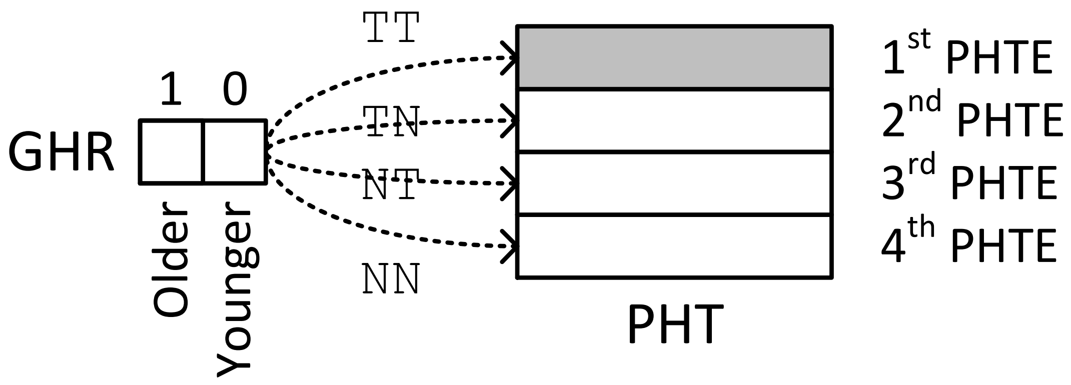
• B3's contribution to PHTE. The probability that both B1 and B2 are taken is denoted as P(B1_T && B2_T), which is equal to P(B1_T)*P(B2_T) = 1*1/2 = 1/2. Given that they are, the probability that B3 is taken, is equal to Q = 1/3. Therefore, the PHTE will be incremented with probability 1/2*1/3 = 1/6 and decremented with probability 1/2*(1-1/3) = 1/3. The net contribution of B3 to PHTE is 1/6-1/3 = -1/6.

• B4's contribution to PHTE. P(B2_T && B3_T) = 1/6. P(B4_T | B2_T && B3_T) = R = 1. B4's net contribution is 1/6*1 = 1/6.

• B1's contribution to PHTE. P(B3_T && B4_T) = 1/6. P(B1_T | B3_T && B4_T) = 1. B1's net contribution is 1/6*1 = 1/6.

• B2's contribution to PHTE. P(B4_T && B1_T) = 1/6*1 = 1/6. P(B2_T | B4_T && B1_T) = 1/2. B2's net contribution is 1/6*1/2 - 1/6*1/2 = 0.

For a single iteration, the net contribution to the PHTE, summed across all the four branches, is equal to 1/6. Since there are 120 iterations, the expected PHTE value is equal to 1/6*120=**20**.



8 Branch Prediction [100 points]

Suppose we have the following loop executing on a pipelined MIPS machine.

```
DOIT  SW    R1, 0(R6)
      ADDI  R6, R6, 2
      AND   R3, R1, R2
      BEQ   R3, R0  EVEN
      ADDI  R1, R1, 3
      ADDI  R5, R5, -1
      BGTZ  R5  DOIT
EVEN  ADDI  R1, R1, 1
      ADDI  R7, R7, -1
      BGTZ  R7 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
SW	3
ADDI	2
AND	3
BEQ/BGTZ	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 branch instruction is in ST stage. For example, the execution of an AND instruction followed by a BEQ would look like:

```
AND       F | D | E1 | E2 | E3 | ST
BEQ           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 SW instruction can be overlapped with the ST stage of the BGTZ 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 SW instruction can be overlapped with the execution of the BGTZ 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 BEQ 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 BEQ 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 BEQ EVEN branch is predicted right = 10 cycles
Number of cycles taken by an iteration when the BEQ EVEN branch is not predicted right = 12 cycles

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

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

9 Interference in Two-Level Branch Predictors [50 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.

```
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:

```
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.

10 Branch Prediction vs Predication [100 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 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 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:

```
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 <- r7, r2               F|D1|D2|D3|D4|D5|D6|- |- |- |- |D7|D8|E1|E2|E3|E4|E5|WB|
add r1 <- r11, r9                  F|D1|D2|D3|D4|D5|- |- |- |- |D6|D7|D8|E1|E2|E3|E4|E5|WB|
X: addi r15 <- r2, 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|
beq r3, r5, X             F|D1|D2|D3|D4|D5|D6|D7|D8|- |- |- |- |E1|E2|E3|E4|E5|WB|
X: addi r15 <- r2, 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 three 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 <- r7, r2            F|D1|D2|D3|D4|D5|D6|- |- |- |- |D7|D8|E1|E2|E3|E4|E5|WB|
add.ne r14 <- r11, r9              F|D1|D2|D3|D4|D5|- |- |- |- |D6|D7|D8|E1|E2|E3|E4|E5|WB|
addi r15 <- r2, 10                    F|D1|D2|D3|D4|- |- |- |- |D5|D6|D7|D8|E1|E2|E3|E4|E5|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.

$(1-P) \times 17 < 3 \times 0.4$

Therefore, for $P > 0.9294$, machine A has higher instruction throughput than machine B.

2. 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 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|
X: 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 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|
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 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.