CMU 18-447 Introduction to Computer Architecture, Spring 2014
HW 3: Branch Prediction, Out-of-order Execution

Instructor: Prof. Onur Mutlu
TAs: Rachata Ausavarungnirun, Varun Kohli, Xiao Bao Zhao, Paraj Tyle

Assigned: Wed., 2/12, 2014
Due: **Wed., 2/26, 2014 (Midnight)**
Handin: `/afs/ece/class/ece447/handin/hw3`

1 Delay Slots (Optional)

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?

(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:
```

(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:
```

(c)
```
ADD R2 <- R4, R3
OR  R5 <- R1, R2
SUB R7 <- R5, R6
BEQ R5 <- R7, X
[Delay Slots]
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.

2 Hardware vs Software Interlocking [30 points]

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 3 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.

(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.

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

(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?

(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)?

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

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

3 Branch Prediction [55 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 % 5 == 0) {   /* B3 */
        sum += val;       /* TAKEN PATH for B3 */
    }
    if (val % 10 == 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, 5, and 10, 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. 10 is a common multiple of 2 and 5.

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 five (i.e., 15) is equal to 1/24, let's call this value Q. Given that a number is a multiple of two and five (i.e., 20), the probability that the number is also a multiple of ten (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 = 1/5. Therefore, the PHTE will be incremented with probability 1/2*1/5 = 1/10 and decremented with probability 1/2*(1-1/5) = 2/5. The net contribution of B3 to PHTE is 1/10-2/5 = -3/10.

• B4's contribution to PHTE. P(B2_T && B3_T) = 1/10. P(B4_T | B2_T && B3_T) = R = 1. B4's net contribution is 1/10*1 = 1/10.

• B1's contribution to PHTE. P(B3_T && B4_T) = 1/10. P(B1_T | B3_T && B4_T) = 1. B1's net contribution is 1/10*1 = 1/10.

• B2's contribution to PHTE. P(B4_T && B1_T) = 1/10*1 = 1/10. P(B2_T | B4_T && B1_T) = 1/2. B2's net contribution is 1/10*1/2 - 1/10*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 -1/10*120=**-12**.

4 Interference in Two-Level Branch Predictors [15 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.

5 Branch Prediction vs Predication [30 points]

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 75% of the time and not taken 25% of the time. On an average, for what range of P would you expect machine A to have a 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 an average, for what range of P would you expect machine A to have a higher instruction throughput than machine B?

6 Out-of-order Execution [50 points]

In this problem, we will give you the state of the Register Alias Table (RAT) and Reservation Stations (RS) for a Tomasulo-like out-of-order execution engine. Your job is to determine the original sequence of **five instructions** in program order.

The out-of-order machine in this problem behaves as follows:

- The frontend of the machine has a one-cycle fetch stage and a one-cycle decode stage. The machine can fetch one instruction per cycle, and can decode one instruction per cycle.
- The machine dispatches one instruction per cycle into the reservation stations, in program order. Dispatch occurs during the decode stage.
- An instruction always allocates the first reservation station that is available (in top-to-bottom order) at the required functional unit.
- When a value is captured (at a reservation station) or written back (to a register) in this machine, the old tag that was previously at that location is *not cleared*; only the valid bit is set.
- When an instruction in a reservation station finishes executing, the reservation station is cleared.
- Both the adder and multiplier are fully pipelined. Add instructions take 2 cycles. Multiply instructions take 4 cycles.
- When an instruction completes execution, it broadcasts its result, and dependent instructions can begin execution in the next cycle if they have all operands available.
- When multiple instructions are ready to execute at a functional unit, the *oldest* ready instruction is chosen.

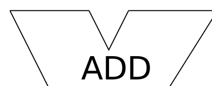
Initially, the machine is empty. Five instructions then are fetched, decoded, and dispatched into reservation stations, before any instruction executes. Then, one instruction completes execution. Here is the state of the machine at this point, after the single instruction completes:

RAT

Reg	V	Tag	Value
R0	1		20
R1	1		50
R2	0	A	37
R3	1	X	500
R4	0	Y	255
R5	1		17
R6	0	Z	73
R7	1		10

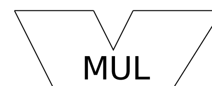
ADD reservation stations:

	Src 1 Tag	Src 1 V	Src 1 Value	Src 2 Tag	Src 2 V	Src 2 Value
A	X	1	500	Y	0	-
B	-	1	20	-	1	17
C						



MUL reservation stations:

	Src 1 Tag	Src 1 V	Src 1 Value	Src 2 Tag	Src 2 V	Src 2 Value
X						
Y	-	1	50	-	1	37
Z	A	0	-	B	0	-



(a) Give the five instructions that have been dispatched into the machine, in program order. The source registers for the first instruction can be specified in either order. Give instructions in the following format: "opcode destination ⇐ source1, source2."

MUL R3 ⇐ R1, R7

MUL R4 ⇐ R1, R2

ADD R2 ⇐ R3, R4

ADD R6 ⇐ R0, R5

MUL R6 ⇐ R2, R6

(b) Now assume that the machine flushes all instructions out of the pipeline and restarts execution from the first instruction in the sequence above. Show the full pipeline timing diagram below for the sequence of five instructions that you determined above, from the fetch of the first instruction to the writeback of the last instruction. Assume that the machine stops fetching instructions after the fifth instruction.

As we saw in class, use "F" for fetch, "D" for decode, "E1," "E2," "E3," and "E4" to signify the first, second, third and fourth cycles of execution for an instruction (as required by the type of instruction), and "W" to signify writeback. You may or may not need all columns shown.

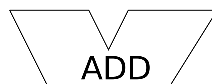
Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
MUL R3 ⇐ R1, R7	F	D	E1	E2	E3	E4	W							
MUL R4 ⇐ R1, R2		F	D	E1	E2	E3	E4	W						
ADD R2 ⇐ R3, R4			F	D				E1	E2	W				
ADD R6 ⇐ R0, R5				F	D	E1	E2	W						
MUL R6 ⇐ R2, R6					F	D				E1	E2	E3	E4	W

Finally, show the state of the RAT and reservation stations after **8 cycles** in the blank figures below.

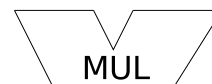
RAT

Reg	V	Tag	Value
R0	1		20
R1	1		50
R2	0	A	37
R3	1	X	500
R4	1	Y	1850
R5	1		17
R6	0	Z	73
R7	1		10

	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
A	X	1	500	Y	1	1850
B						
C						



	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
X						
Y						
Z	A	0	-	B	1	37

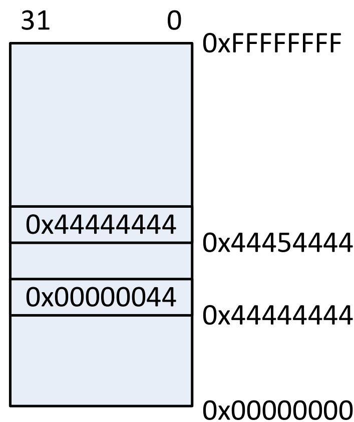


7 0x44444444 [35 points]

A 32-bit processor implements paging-based virtual memory using a single-level page table. The following are the assumptions about the processor's virtual memory.

- The number of bytes in a page is greater than four and is also a power of two.
- The base address of the page table is page-aligned.
- A page table entry (PTE) stores **only** the physical page number and has the following format. All of the unused bits in the PTE are set to 0.

The following figure shows the physical memory of the processor at a particular point in time.



At this point, when the processor executes the following piece of code, it turns out that the processor accesses the page table entry residing at the physical address of `0x44444444`.

```
char *ptr = 0x44444444;
char val = *ptr; // val == 0x44
```

Let n be equal to $\log_2(pagesize)$.
What is the page size of the processor? Please show work for partial credit.
Hint 1: PTBA is page-aligned. What are the possible values of n?
Hint 2: Physical address 0x44454444 is not a PTE. What are the possible values of n?

Clue 1: PTBA is page-aligned.

```
PTBA & ~(1<<n) = 0
∴ (PTE_PA - VPN * PTE_SIZE)        & ~(1<<n) = 0
∴ (0x44444444-(0x44444444>>n)*4)   & ~(1<<n) = 0
∴ (0x44444444-(0x44444444>>(n-2)) & ~(1<<n) = 0
```

This means that n is of the form: $4k + 2$, where k is an integer.
Possible values of n: 6, 10, 14, 18, 22, 26, 30.
For these values of n, let us check whether the following equation is indeed equal to 0.

```
(0x44444444-(0x44444444>>(n-2)) & ~(1<<n)

n=6:   (0x44444444-(0x44444444>>4))  & ~(1<<6)  = 0x40000000 & 0x0000003F = 0
n=10:  (0x44444444-(0x44444444>>8))  & ~(1<<10) = 0x44000000 & 0x000003FF = 0
n=14:  (0x44444444-(0x44444444>>12)) & ~(1<<14) = 0x44400000 & 0x00003FFF = 0
n=18:  (0x44444444-(0x44444444>>16)) & ~(1<<18) = 0x44440000 & 0x0003FFFF = 0
n=22:  (0x44444444-(0x44444444>>20)) & ~(1<<22) = 0x44444000 & 0x003FFFFF != 0
n=26:  (0x44444444-(0x44444444>>24)) & ~(1<<26) = 0x44444400 & 0x03FFFFFF != 0
n=30:  (0x44444444-(0x44444444>>28)) & ~(1<<30) = 0x44444440 & 0x3FFFFFFF != 0
```

Possible values of n: 6, 10, 14, 18.

Clue 2: Physical address 0x44454444 is not a PTE

For the possible values of n, let us check whether the last PTE of the page table is stored at a lower physical address than 0x44454444.

```
PA of Last PTE (LPTE_PA) = PTBA + ((1<<(32-n)) - 1) * PTE_SIZE
∴ LPTE_PA = PTBA + ((1<<(34-n))-4)

n=6:   0x40000000+((1<<28)-4) = 0x40000000+0x0ffffffc = 0x4ffffffc > 0x44454444
n=10:  0x44000000+((1<<24)-4) = 0x44000000+0x00fffffc = 0x44fffffc > 0x44454444
n=14:  0x44400000+((1<<20)-4) = 0x44400000+0x000ffffc = 0x444ffffc > 0x44454444
n=18:  0x44440000+((1<<16)-4) = 0x44440000+0x0000fffc = 0x4444fffc < 0x44454444
```

The only possible value of n: 18.

8 GPUs and SIMD [35 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.

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 4 instructions in each thread.) A warp in the GPU consists of 64 threads, and there are 64 SIMD lanes in the GPU.

```
for (i = 0; i < 1024768; i++) {
    if (B[i] < 4444) {
       A[i] = A[i] * C[i];
       B[i] = A[i] + B[i];
       C[i] = B[i] + 1;
    }
}
```

(a) How many warps does it take to execute this program?

> Warps = (Number of threads) / (Number of threads per warp)
> Number of threads = 2^{20} (i.e., one thread per loop iteration).
> Number of threads per warp = 64 = 2^6 (given).
> Warps = $2^{20}/2^6 = 2^{14}$

(b) When we measure the SIMD utilization for this program with one input set, we find that it is 67/256. What can you say about arrays A, B, and C? Be precise (Hint: Look at the "if" branch, what can you say about A, B and C?).

A: 1 in every 64 of A's elements less than 4444.

B: Nothing.

C: Nothing.

(c) Is it possible for this program to yield a SIMD utilization of 100% (circle one)?

[YES] NO

If YES, what should be true about arrays A, B, C for the SIMD utilization to be 100%? Be precise.

If NO, explain why not.

A: Either:
(1) All of A's elements are greater than 0, or
(2) All of A's elements are less than or equal to 0.

(d) Is it possible for this program to yield a SIMD utilization of 25% (circle one)?

If YES, what should be true about arrays A, B, and C for the SIMD utilization to be 25%? Be precise.

If NO, explain why not.

The smallest SIMD utilization possible is the same as part (b), 67/256, but this is g
than 25%.