CMU 18-447 Introduction to Computer Architecture, Spring 2015

HW 3: Branch Prediction, Out-of-order Execution, SIMD, and GPUs

Instructor: Prof. Onur Mutlu
TAs: Rachata Ausavarungnirun, Kevin Chang, Albert Cho, Jeremie Kim, Clement Loh

Assigned: Wed., 2/11, 2015
Due: **Wed., 2/25, 2015 (Midnight)**
Handin: **Autolab**

1 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?

2 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.

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

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.

3 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?

4 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

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

ADD

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

MUL

(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."

_____ ____ ⇐ ____, ____

_____ ____ ⇐ ____, ____

_____ ____ ⇐ ____, ____

_____ ____ ⇐ ____, ____

_____ ____ ⇐ ____, ____

(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														
MUL R4 ⇐ R1, R2														
ADD R2 ⇐ R3, R4														
ADD R6 ⇐ R0, R5														
MUL R6 ⇐ R2, R6														

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

RAT

Reg	V	Tag	Value
R0			
R1			
R2			
R3			
R4			
R5			
R6			
R7			

	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
A						
B						
C						

ADD

	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
X						
Y						
Z						

MUL

5 Load Store Handling [36 points]

A modern out-of-order execution processor has a store buffer (also called a store queue) as discussed in class. Remember that this buffer is implemented as a searchable structure, i.e., content addressable memory. Such a processor also has a load buffer (also called a load queue), which contains all information about decoded but not yet retired load instructions, in program order. Also as discussed in class, most modern processors intelligently schedule load instructions in the presence of "unknown address" stores. The process of handling load/store dependencies and scheduling is called "memory disambiguation". For this question, assume a processor that aggressively schedules load instructions even in the presence of older store instructions with unknown addresses. Within this context, answer the following questions.

1. What is the purpose of the store buffer?

2. Exactly when is the store buffer searched (if at all)?

3. What is the purpose of the load buffer?

4. Exactly when is the load buffer searched (if at all)?

5. You are given the following state of the store buffer and the load buffer at a given instant of time. Assume the ISA allows unaligned loads and stores, registers are 4 bytes, and byte and double-byte loads are sign-extended to fill the 4-byte registers. Assume no pipeline flushes during the execution of the code that lead to this state. Starting addresses and sizes are denoted in bytes; memory is little-endian. Data values are specified in hexadecimal format.

Store buffer

Inst num.	Valid	Starting Address	Size	Data
3	Yes	10	1	x00000005
5	Yes	36	4	x00000006
6	Yes	64	2	x00000007
7	Yes	11	1	x00000004
9	Yes	44	4	x00000005
10	Yes	72	2	x00000006
11	Yes	12	1	x00000006
13	Yes	52	4	x00000008
14	Yes	80	2	x00000007
15	Yes	13	1	x00000006
32	Yes	Unknown	4	x00000009

Load buffer

Inst num.	Valid	Starting Address	Size	Destination
8	Yes	10	2	R1
12	Yes	11	2	R2
22	Yes	12	4	R3
28	Yes	10	4	R4
29	Yes	46	2	R5
30	Yes	13	2	R6
36	Yes	10	4	R7
48	Yes	54	2	R8

What is the value (in hexadecimal notation) loaded by each load into its respective destination register after the load is committed to the architectural state?

R1:

R2:

R3:

R4:

R5:

R6:

R7:

R8:

6 Restartable vs. Precise Interrupts [6 points]

As we discussed in one of the lectures, an exception (or interrupt) is "restartable" if a (pipelined) machine is able to resume execution exactly from the state when the interrupt happened and after the exception or interrupt is handled. By now you also should know what it means for an interrupt to be precise versus imprecise.

Can a pipelined machine have restartable but imprecise exceptions or interrupts?

What is the disadvantage of such a machine over one that has restartable and precise exceptions or interrupts? Explain briefly.

7 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?

(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:

B:

C:

(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.

B:

(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.

8 Vector Processing [40 points]

You are studying a program that runs on a vector computer with the following latencies for various instructions:

- VLD and VST: 50 cycles for each vector element; fully interleaved and pipelined.
- VADD: 4 cycles for each vector element (fully pipelined).
- VMUL: 16 cycles for each vector element (fully pipelined).
- VDIV: 32 cycles for each vector element (fully pipelined).
- VRSHF (right shift): 1 cycle for each vector element (fully pipelined).

Assume that:

- The machine has an in-order pipeline.
- The machine supports chaining between vector functional units.
- In order to support 1-cycle memory access after the first element in a vector, the machine interleaves vector elements across memory banks. All vectors are stored in memory with the first element mapped to bank 0, the second element mapped to bank 1, etc.
- Each memory bank has an 8KB row buffer.
- Vector elements are 64 bits in size.
- Each memory bank has two ports (so that two loads/stores can be active simultaneously), and there are two load/store functional units available.

(a) What is the minimum power-of-two number of banks required in order for memory accesses to never stall? (Assume a vector stride of 1.)

(b) The machine (with as many banks as you found in part (a)) executes the following program (assume that the vector stride is set to 1):

```
VLD V1 <- A
VLD V2 <- B
VADD V3 <- V1, V2
VMUL V4 <- V3, V1
VRSHF V5 <- V4, 2
```

It takes 111 cycles to execute this program. What is the vector length?

If the machine did not support chaining (but could still pipeline independent operations), how many cycles would be required to execute the same program? Show your work.

(c) The architect of this machine decides that she needs to cut costs in the machine's memory system. She reduces the number of banks by a factor of 2 from the number of banks you found in part (a) above. Because loads and stores might stall due to bank contention, an arbiter is added to each bank so that pending loads from the oldest instruction are serviced first. How many cycles does the program take to execute on the machine with this reduced-cost memory system (but with chaining)?

Now, the architect reduces cost further by reducing the number of memory banks (to a lower power of 2). The program executes in 279 cycles. How many banks are in the system?

(d) Another architect is now designing the second generation of this vector computer. He wants to build a multicore machine in which 4 vector processors share the same memory system. He scales up the number of banks by 4 in order to match the memory system bandwidth to the new demand. However, when he simulates this new machine design with a separate vector program running on every core, he finds that the average execution time is longer than if each individual program ran on the original single-core system with 1/4 the banks. Why could this be (in less than 20 words)? Provide concrete reason(s).

What change could this architect make to the system in order to alleviate this problem (in less than 20 words), while *only* changing the shared memory hierarchy?

9 Research Paper Summaries [20 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 summary/review.

- Lecture slides on guidelines for reviewing papers.
  `http://www.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=onur-447-s15-how-to-do-the-paper-reviews.pdf`
- Good example reviews by your fellow classmates.
  `http://www.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=reviews_patt_447.txt`
  `http://www.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=reviews_memory_447.txt`

(a) Write a half-page summary for the following paper:
McFarling, "Combining Branch Predictors," WRL Technical Note TN-36, 1993. `http://www.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=mcfarling_-_1993_-_combining_branch_predictors.pdf`

(b) Write a half-page summary for the following paper:
Smith and Plezskun, "Implementing Precise Interrups in Pipelined Processors," in IEEE Trans on Computers, 1988. `http://www.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=00004607.pdfo`