Full Name:________________________________

15-740/18-740, Spring 2018

Exam 2

May 3, 2018, 3:00pm-4:20pm

Instructions:

- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer. A few pages of scratch paper are provided at the end of the text booklet, but your final answer should be written in the space provided.
- Show your work and discuss your answer. You will be graded more on your explanation than on your final answer.
- The exam has a maximum score of 75 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is CLOSED BOOK, CLOSED NOTES. You may use a calculator, but no networked devices (e.g., phones, laptops, etc.).
- The exam introduces techniques and key ideas from research papers and asks you to analyze them. *Confine your answers to the ideas that are presented in the exam.* We are not looking for answers that involve other techniques in these papers which we do not present.

Do not write below this line

Problem	Your Score	Possible Points
1		15
2		50
3		10
Total		75

Pipelining and Data Hazards

Problem 1. (15 points total):

Recall that the stages in the simple 5-stage pipeline that we discussed in class are: **IF** (instruction fetch), **ID** (instruction decode and register fetch), **EX** (execute), **MEM** (memory access), and **WB** (write back).

Assume that data hazards are handled by *forwarding* (aka bypassing) whenever possible. Whenever forwarding is insufficient, assume that stalling is used to resolve data hazards. You can also assume that there are no cache misses, and that data dependences through memory (e.g., a store to memory location A followed immediately by a load from A) do not cause any additional stalls.

Consider the following assembly

```
lw $t0, 0($s0)
add $t0, $t0, $s0     # $t0 = $t0 + $s0
addi $t1, $t0, 9      # $t1 = $t0 + 9
sw $t1, 0($s0)
lw $t2, 0($t1)
lw $t3, 0($t2)
lw $t5, 4($t4)
addi $s0, $s0, 32
lw $t6, 0($t5)
sw $t6, 0($s0)
```

A. In the diagram below, indicate which instruction (if any) is in each pipeline stage during a given cycle by filling in the appropriate box with one of the following: **F** (IF stage), **D** (ID stage), **E** (EX stage), **M** (MEM stage), or **W** (WB stage). If an instruction stalls, just fill in the same letter twice.

You should also indicate all instances of forwarding by **drawing an arrow** from the stage from which data is forwarded to the stage in which data is used. Also **indicate the register** being forwarded.

5 points

Instr	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}	c_{17}
lw $t0, 0($s0)	F	D	E	M	W $t0													
add $t0, $t0, $s0		F	D	D	E	M $t0	W											
addi $t1, $t0, 9			F	F	D	E	M $t1	W $t1										
sw $t1, 0($s0)					F	D	E	M	W									
lw $t2, 0($t1)						F	D	E	M	W $t2								
lw $t3, 0($t2)							F	D	D	E	M	W						
lw $t5, 4($t4)								F	F	D	E	M	W $t5					
addi $s0, $s0, 32										F	D	E	M	W $s0				
lw $t6, 0($t5)											F	D	E	M	W $t6			
sw $t6, 0($s0)												F	D	E	M	W		

B. Billy heard that memory latency is a problem for processors, and decided to modify the pipeline in an attempt to fix it. Consider a new microarchitecture where the **MEM** stage happens before the **EX** stage. Make minimal changes to the assembly so that it fits into the new pipeline. Do not aggressively reschedule the instructions.

2 points

Displacement addresses are not possible

lw $t5, 4($t4) $\rightarrow$ addi $t4, $t4, 4
lw $t5, ($t4)

C. Fill in the pipeline diagram for the above instructions as in part A. Handle data hazards by *forwarding* whenever possible, and stalling otherwise.

5 points

Instr	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}	c_{17}	c_{18}	c_{19}	c_{20}
lw $t0, ($s0)	F	D	M	E	W $t0																
add $t0, $t0, $s0		F	D	M	E	W $t0															
addi $t1, $t0, 9			F	D	M	E	W $t1														
sw $t1, ($s0)				F	D	D	M	E ~~$t1~~	W												
lw $t2, ($t1)					F	F	D	M	E $t2	W											
lw $t3, ($t2)							F	D	M	E	W										
addi $t4, $t4, 4								F	D	M	E	W $t4									
lw $t5, ($t4)									F	D	D	M	E	W $t5							
addi $s0, $s0, 32										F	F	D	M	E	W $s0						
lw $t6, ($t5)												F	D	M $t6	E	W					
sw $t6, ($s0)													F	D	M	E	W				

D. Discuss the pros and cons of Billy's microarchitecture. Is it effective at hiding memory latency?
3 points

Pros :

→ consecutive loads benefited from M → M forwarding

→ Ex doesn't stall (No more Ex stall ~~ct No stall Ex stalls~~ due to Mem → Ex forwarding)

Cons :

→ More instructions (introduced for displacement adresses)

→ Mem stalls ~~due to~~ for Ex → Mem forwarding

No, it's not effective at hiding memory latency.

Potpourri

Problem 2. (50 points total):

A. **Direct-to-Data Cache**

Modern processors optimize for cache energy and performance by employing multiple levels of caching that give the illusion of a large, high-bandwidth, low-latency memory on access patterns with good locality. Direct-to-Data (D2D) cache[1] is a scheme that locates data across the entire cache hierarchy with a single lookup. To navigate the cache hierarchy, D2D extends the TLB with per-cache-line location information that indicates in *which cache* and *which way within that cache* the cache line is located.

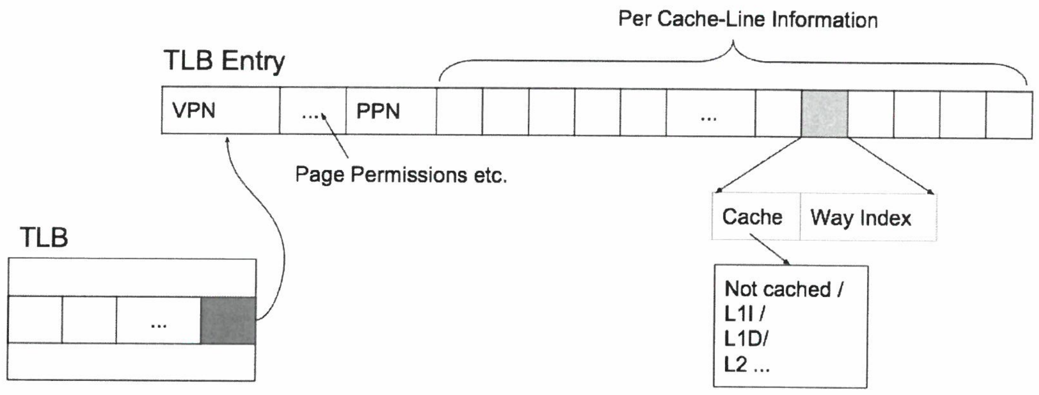

Figure 1: D2D lookup mechanism via the TLB.

(a) D2D claims to significantly reduce the L2 latency and total cache hierarchy energy. Explain why.
5 points

Skip L2 tag access.
Only activate one L2 way (perfect way prediction.)

Note: L1 is still accessed! It's in parallel w/ TLB;
don't want to put TLB on critical path.

[1]Sembrant A, Hagersten E, Black-Schaffer D. The Direct-to-Data (D2D) cache: navigating the cache hierarchy with a single lookup. ISCA 2014

(b) Discuss correctness- and performance-related challenges of implementing D2D in real systems. E.g., give a scenario that could cause a problem if the system only had the mechanism presented in Figure 1.
5 points

Many possibilities...

E.g., evictions must broadcast TLB updates.
(w/out other mechanisms)

(c) Huge pages of 2 MB or 1 GB are commonly used to improve the effectiveness of TLBs for in-memory big-data processing. How do huge pages interact with D2D cache? Do they make the scheme more or less practical?
5 points

Huge pages contain tons of cache lines

⇒ TLB is enormous

⇒ Too much area, long access time, etc.

B. **Accelerators**

(a) You are the chief architect of a forthcoming SoC. One of your engineers presents you with an accelerator design that she claims improves energy-efficiency by 2,000× over an optimized software implementation. Would you include this design in your SoC? Based on the case studies presented in lecture, what questions would you want answered before moving forward?
5 points

Many good answers here. E.g.,

① Does 2000X include memory?

② What is it accelerating?

③ How does it interface w/ rest of the system?

(b) Why are accelerators particularly interesting now vs. 10–20 years ago?
5 points

- "Dark silicon" means we can't just scale core count
- Multicores ~~cant~~ are hard to use well anyway.
- Interesting & widely used app: deep learning.

C. Nano Instruction Set Computers

Huang et al. proposed an ISA extension that decouples the data access and register write operations in a load instruction[2]. Each load instruction is split into 2 *nano-instructions*:

```
ld rD, I(rA)        ⇒        ld.D lt, I(rA)
                             ld.wb rD, lt
```

The first nano-instruction performs the data access, and the second nano-instruction orders the load and writes the result back to the register. The former behaves much like a conventional load except that it places the contents at memory address `I(rA)` into the destination *load tag* `lt`, which is stored in a new associative structure called the Load Tag Table (LTT). The LTT tracks the address, value, status bits, *and exception information* of pending loads.

As an example, the following sequence of instructions translates into NISC with load decoupling as shown:

```
                                      add r5, r6, r7
    add r5, r6, r7                    blt r3, r5, B
    blt r3, r5, B                     j C
    j C                           B:  ld.D lt0, 0(r20)
B:  ld r3, 0(r20)         ⇒           ld.wb r3, lt0
    div r4, r3, r8                    div r4, r3, r8
    j D                               j D
C:  ld r14, -40(r2)               C:  ld.D lt1, -40(r2)
                                      ld.wb r14, lt1
```

[2]Huang Z, Hilton AD, Lee BC. Decoupling loads for nano-instruction set computers. ISCA 2016.

(a) What are the advantages of this decoupling for in-order processors?
5 points

Decoupling lets the compiler hide load latency by moving the access much earlier in the program.

(b) Can the performance benefits of decoupled loads be achieved by simply increasing the register file size by the number of LTT entries? Why or why not?
5 points

No, moving loads across branches is unsafe. They can cause exceptions and there could be memory aliasing with another store.

(Answers that discuss register file size got partial credit.)

(c) Do you expect this design to be significantly beneficial with OoO processors? Why or why not?
5 points

No. OoO already has mechanisms to move loads earlier in the program.

D. **Register allocation**

Registers are often used to communicate values between nearby instructions such that the register is read only once before it is overwritten. E.g., in SPEC 2000, 86% of instructions have their destination register overwritten before they are committed.[3] A large body of research exploits this property to improve OoO processor performance and efficiency. Give an example of how one could do so.

5 points

Basic idea: Don't need to allocate a physical register all the way until commit.

Many variants on this observation.

E. **EPIC on Itanium**

Intel's Itanium architecture is an EPIC ISA. EPIC ISAs group independent instructions into *bundles*, with a *stop bit* in each instruction indicating the end of a bundle. The semantics are that all instructions inside a bundle can be safely executed in parallel. Unlike VLIW, bundles can contain a variable number of instructions of different types. EPIC claims to address two major drawbacks of VLIW. What are these drawbacks?

5 points

① Portability.

② Code bloat.

③ Allows for dynamic execution in hardware to some degree.

I wasn't intending this but it got credit.

[3]Balkan D, Sharkey J, Ponomarev D, and Ghose K. SPARTAN: ..., PACT 2006.

VLIW

Problem 3. (10 points total):

Consider the execution of following code segment to find the maximum value and its index in an array on a VLIW machine.

```
for (i = 0; i < N; i++) {
  if (max < a[i]) {
    idx = i;
    max = a[i];
  }
}
```

The code above translates to the following instructions:

```
# t0: i, s0: idx, f0: max, a0: N, a1: address of a[i]
loop:   fld f1, 0(a1)           # load a[i]
        flt.d t1, f0, f1        # set if max < a[i]
        fmax.d f0, f0, f1       # max = max < a[i] ? a[i] : max
        beqz t1, skip           # if max >= a[i], jump to skip
        addi s0, t0, 0          # update idx
skip:   addi a1, a1, 8          # bump a
        addi t0, t0, 1          # increment i
        bltu t0, a0, loop       # loop
```

Our VLIW machine has five execution units:

- two integer units, latency **one** cycle, also used for branches
- one memory unit, latency **two** cycles, fully pipelined
- two floating point units, latency **three** cycles, fully pipelined, each unit can perform flt.d and fmax.d

A. Schedule instructions for the VLIW machine naively **without loop unrolling or software pipelining**. **5 points**

Label	ALU1	ALU2	MEM	FPU1	FPU2
loop:		addi a1,a1,8	fld f1,0(a1)		
				flt.d t1,f0,f1	fmax.d f0,f0,f1
		beqz t1 skip			
		addi s0,t0,0			
skip:	addi t0,t0,1	bltu t0,a0,loop			

Multiple solutions possible. Any correct solution ~~[illegible] considered~~ given full points.

B. What is the maximum possible efficiency (i.e., fewest cycles-per-iteration) that can be attained by software pipelining on this instruction sequence? (What is the bottleneck that limits the efficiency?)

5 points

3 cycles per iteration

There is a true dependency between the fmax.d instruction across iterations.

C. **Bonus:** How would you change the instruction sequence to achieve higher efficiency?

5 points

→ Split the array into strides (of 3) and maintain a separate max and idx for each index in the stride. Compute the max of each of these and the corresponding index in the epilogue.

→ Use predicated instructions ~~[illegible]~~

D. **Bonus:** Software pipeline the above code as efficiently as you can.
(15 − 3 × cycles-per-iteration of your code) points

	Label	ALU1	ALU2	MEM	FPU1	FPU2
		addi a1,a1,24		fld f1,0(a1)		
		addi t0,t0,3		fld f2,-16(a1)		
prologue				fld f3,-8(a1)		
		addi a1,a1,24		fld f1,0(a1)	flt.d t1,f10,f1	fmax f10,f10,f1
		addi t0,t0,3		fld f2,-16(a1)	flt.d t2,f20,f2	fmax f20,f20,f2
				fld f3,-8(a1)	flt.d t3,f30,f3	fmax f30,f30,f3
loop	Loop:	addi a1,a1,24	[t1] addi s10,t0,-6	fld f1,0(a1)	flt.d t1,f10,f1	fmax f10,f10,f1
body		addi t0,t0,3	[t2] addi s20,t0,-5	fld f2,-16(a1)	flt.d t2,f20,f2	fmax f20,f20,f2
		bltu t0,a0,Loop	[t3] addi s30,t0,-4	fld f3,-8(a1)	flt.d t3,f30,f3	fmax f30,f30,f3
			[t1] addi s10,t0,-6		flt.d ~~f~~t1,f10,f1	fmax f10,f10,f1
			[t2] addi s20,t0,-5		flt.d t2,f20,f2	fmax f20,f20,f2
			[t3] addi s30,t0,-4		flt.d t3,f30,f3	fmax f30,f30,f3
			[t1] addi s10,t0,-3		flt.d t2,f10,f20	fmax f0,f10,f20
epilog-		addi s0,s10,0	[t2] addi s20,t0,-2			
ue			[t3] addi s30,t0,-1			
		[t2] addi s0,s20,0			flt.d t3,f0,f30	fmax f0,f0,f30
		[t3] addi s0,s30,0				

This is for 1 cycle per iteration. ~~[illegible]~~
Ones with 3/4 cycles per iteration are easier

Scratch paper

Scratch paper

Scratch paper