CS 152/252A Spring 2023 Computer Architecture Midterm 2

Please read all exam instructions carefully. They may have changed since your last exam.

- You have 110 minutes to complete the exam unless you have DSP accommodations. Exam questions are in no particular order (except 252a questions are last).
- You must write your student ID on the bottom-left of every page of the exam (except this first one). You risk losing credit for any page you don't write your student ID on.
- **For questions with length limits, do not use semicolons or dashes to lengthen your explanation.**
- The exam is closed book, no calculator, and closed notes, other than one double-sided cheat sheet that you may reference.
- For multiple choice questions,

  ☐ means mark **all options** that apply

  ◯ means mark a **single choice**

First name	
Last name	
SID	
Exam Room	
Name and SID of person to the right	
Name and SID of person to the left	
Discussion TAs (or None)	

Honor code

"As a member of the UC Berkeley community, I act with honesty, integrity, and respect for others."

While the statement of the Honor Code itself is brief, it is an affirmation of our highest ideals as Golden Bears. By signing below, I affirm that all work on this exam is my own work, and honestly reflects my own understanding of the course material. I have not referenced any outside materials (other than one double-sided cheat sheet), nor collaborated with any other human being on this exam. I understand that if the exam proctor catches me cheating on the exam, that I may face the penalty of an automatic "F" grade in this class and a referral to the Center for Student Conduct.

Signature: ______________________________

THIS PAGE IS INTENTIONALLY LEFT BLANK

Q1. [12 pts] Predict me, predict me not

(a) [6 pts] **Who's That Branch Predictor!**
In this part, consider the implementation of the following predictors on a traditional implementation of the RISC-V ISA. Reminder: a direct branch is a branch in which the destination address can be fully determined simply by inspecting the instruction bits and the current `PC` (i.e. `j label`). An indirect branch is one which requires *additional* registers other than PC (i.e. `jr x1`, etc.) to determine the destination.

(i) [2 pts] Which type(s) of branches could a Bi-Modal Counter help resolve?

- [✓ Conditional branches]
- ☐ Direct branches
- ☐ Indirect branches
- ☐ Return instructions
- ○ None of the above

(ii) [2 pts] Which type(s) of branches could a Branch Target Buffer (BTB) help resolve?

- [✓ Conditional branches]
- [✓ Direct branches]
- [✓ Indirect branches]
- [✓ Return instructions]
- ○ None of the above

(iii) [2 pts] Which type(s) of branches could a Return Address Stack (RAS) help resolve?

- ☐ Conditional branches
- ☐ Direct branches
- ☐ Indirect branches
- [✓ Return instructions]
- ○ None of the above

(b) [6 pts] **Branch Predictor Design**

With the knowledge gained from taking CS152/252, you and your classmates decide to launch a startup named SiSix (no relation to SiFive) which offers a range CPUs designed for different applications. You are tasked with developing a branch prediction scheme for a few of these processors.

For each of the following processors, select an **appropriately efficient and performant** branch prediction scheme that is suitable for each application. In no more than **two sentences**, justify your choice.

The first CPU is a very low power, in-order, 3-stage microcontroller designed to be used in simple four-function calculators, TV remotes, and other small battery powered devices. The processor runs at 100KHz and is not intended to support applications with hard real time deadlines.

(i) [1 pt] Which kind of branch prediction system should this processor use?

- [✓ Predict all branches are not taken]
- ○ A PC indexed Bi-Modal Counter table
- ○ A hierarchy of TAGE and perceptron based predictors

(ii) [2 pts] In no more than two sentences, justify why such a choice is appropriate for this processor.

Not taken prediction is the most reasonable option for extremely low power and low performance microcontrollers since it allows the processor to be as uncomplicated as possible, which saves power through area reduction. While high quality branch prediction can lower power usage by avoiding unnecessary work, the energy cost of maintaining the data structures required to achieve such high quality predictions likely outweighs any gains that could be had by reducing mispredictions on a relatively short three stage processor.

The second CPU is a very deeply pipelined out-of-order processor meant for desktop computers. It is meant to support computationally heavy, interactive tasks like video games, scientific simulations, and code compilation.

(iii) [1 pt] Which kind of branch prediction system should this processor use?

- ○ Predict all branches are not taken
- ○ A PC indexed Bi-Modal Counter table
- [✓ A hierarchy of TAGE and perceptron based predictors]

(iv) [2 pts] In no more than two sentences, justify why such a choice is appropriate for this processor.

Deeply out-of-order processors rely on very high quality branch prediction to provide high performance and avoid costly flushes. Power is much less of a concern for a desktop computers as power is plentiful (from the outlet) and so privileging maximum performance over energy efficiency is reasonable, especially given the demanding workloads the processor is intended for.

https://www.overleaf.com/project/641a8fcb850fd9df71704d5b

Q2. [9 pts] (CS152 Only) CPUs, GPUs, and bears (oh my!)

(a) **(i)** [1 pt] What kind of parallelism does a multi-core CPU take advantage of?

- ☐ DLP (Data Level Parallelism)
- ☐ ILP (Instruction Level Parallelism)
- [✓ TLP (Thread Level Parallelism)]
- ○ None of the above

(ii) [1 pt] What kind of parallelism does a superscalar CPU take advantage of?

- ☐ DLP (Data Level Parallelism)
- [✓ ILP (Instruction Level Parallelism)]
- ☐ TLP (Thread Level Parallelism)
- ○ None of the above

(iii) [1 pt] What kind of parallelism does a GPU take advantage of?

- [✓ DLP (Data Level Parallelism)]
- ☐ ILP (Instruction Level Parallelism)
- [✓ TLP (Thread Level Parallelism)]
- ○ None of the above

(b) For each of the following tasks, choose which architecture or processor would be best suited for the task. Then, explain why your selection would be best in **at most two** sentences.

(i) [1 + 2 pts] Contains many nested if-else statements in addition to very frequent jumps. A majority of the code involves branches.

- [✓ Superscalar OoO (Out-of-Order) CPU]
- ○ GPU

Why is your selection best? Explain in **at most two** sentences.

For a branch-heavy task, GPU threads will diverge more and more, degrading performance in comparison to the superscalar OoO CPU. The CPU can handle the branch-heavy task likely using an accurate branch predictor as well as with speculated instructions.

(ii) [1 + 2 pts] Running large independent matrix multiplication simulations in parallel.

- [✓ Single core within GPU]
- ○ Standard 8-lane vector processor

Why is your selection best? Explain in **at most two** sentences.

As seen in lecture, a GPU consists of many multithreaded SIMD cores, which (because of its extra architectural state) lends itself to switching contexts between workloads at a lesser cost. Furthermore, multithreaded SIMD cores usually have more lanes (16 or 32) and can afford more data level parallelism, which is suitable for a large matrix multiplication simulation.

Q3. [14 pts] (CS152 Only) Day in the life of a VLIW debugger

(a) You're asked by your colleague to help debug their software pipelined VLIW implementation of the following program:

```
Loop:
    addi x8, x8, 1
    // Loads
    fld f1, 0(x1)
    fld f2, 0(x2)
    fld f3, 0(x3)
    fld f4, 0(x4)
    // Computation
    fmul f5, f1, f2
    fadd f0, f0, f5
    fmul f6, f3, f4
    fsub f0, f0, f6
    // Iterate
    addi x1, x1, 4
    addi x2, x2, 4
    addi x3, x3, 4
    addi x4, x4, 4
    // Loop
    bne x8, x9, loop
```

Your colleague's implementation is shown on the next page (shapes are used to help differentiate the software pipelining):

#: Label	ALU	ALU	MEM	FADD	FMUL
1:		🔷 addi x1	🔷 fld f1		
2:		🔷 addi x2	🔷 fld f2		
3:		🔷 addi x3	🔷 fld f3		
4:		🔷 addi x4	🔷 fld f4		
5:		● addi x1	● fld f1		🔷 fmul f5
6:		● addi x2	● fld f2		🔷 fmul f6
7:		● addi x3	● fld f3		
8:		● addi x4	● fld f4		
9: loop		★addi x1	★fld f1	🔷 fadd f0	● fmul f5
10:		★addi x2	★fld f2		● fmul f6
11:		★addi x3	★fld f3	🔷 fsub f0	
12:	bne loop	★addi x4	★fld f4		
13:				● fadd f0	★fmul f5
14:					★fmul f6
15:				● fsub f0	
16:					
17:				★fadd f0	
18:					
19:				★fsub f0	

CORRECT:

Line: Label	ALU	ALU	MEM	FADD	FMUL
1:		🔷addi x1	🔷fld f1		
2:		🔷addi x2	🔷fld f2		
3:		🔷addi x3	🔷fld f3		
4:		🔷addi x4	🔷fld f4		
5:		●addi x1	●fld f1		🔷fmul f5
6:		●addi x2	●fld f2		🔷fmul f6
7:		●addi x3	●fld f3		
8:		●addi x4	●fld f4		
9: loop		⋆addi x1	⋆fld f1		●fmul f5
10:		⋆addi x2	⋆fld f2	🔷fadd f0	●fmul f6
11:		⋆addi x3	⋆fld f3		
12:	bne loop	⋆addi x4	⋆fld f4	🔷fsub f0	
13:					⋆fmul f5
14:				●fadd f0	⋆fmul f6
15:				●fsub f0	
16:					
17:					
18:				⋆fadd f0	
19:				⋆fsub f0	

When trying to run their implementation on a machine with the following latencies,

1. ALU: 1 cycle
2. MEM: 2 cycles
3. FADD: 2 cycles
4. FSUB: 2 cycles
5. FMUL: 5 cycles

Your colleague discovers that their implementation is broken and asks for your help in finding the error.

(i) [1 pt] For this part **ONLY**, assume that your colleague's implementation above magically begins to work properly somehow. In other words, their implementation as given mysteriously runs **without any errors**.

Assuming that is the case, what is the FLOPs per cycle of their implementation, considering only the main loop (starting from `loop` to `bne loop`, inclusive)? For the purposes of this question, please do not consider floating point loads as FLOPs; **consider only floating point arithmetic instructions** as FLOPs.

Write the number below, e.g. write "0" if you think there are 0 FLOPs per cycle for the implementation.

4/4 OR 1

(ii) [1 pt] In the "hot loop" of the above implementation, which of the following lines contains the **first** error?

[✓ Line 9]
Line 10
Line 11
Line 12

(iii) [2 pts] In the line you selected above, which instruction(s) is / are invalid?

addi
bne
fld
[✓ fadd]
fsub
fmul

(iv) [2 pts] Why is / are the instruction(s) you selected above invalid? Explain using **at most two sentences**.

Instruction `fadd f0` in line 9 depends on the result of instruction `fmul f5` in line 5, which only becomes available on and after line 10. However, instruction `fadd f0` is issued in line 9 which is *before* line 10, and thus is invalid.

(v) [2 pts] Can you rewrite the main loop (from `loop` to `bne loop`) to resolve the issues mentioned above? Assume that the code before the main loop does **not** change, and that the code **after** the main loop will be adjusted accordingly to your solution (and therefore won't result in any additional errors).

#: Label	ALU	ALU	MEM	FADD	FMUL
`9: loop`					
`10:`					
`11:`					
`12:`	`bne loop`				

SID: ________________________

#: Label	ALU	ALU	MEM	FADD	FMUL
`9: loop`		`⋆addi x1`	`⋆fld f1`		`● fmul f5`
`10:`		`⋆addi x2`	`⋆fld f2`	`🔷 fadd f0`	`● fmul f6`
`11:`		`⋆addi x3`	`⋆fld f3`		
`12:`	`bne loop`	`⋆addi x4`	`⋆fld f4`	`🔷 fsub f0`	

(b) [3 pts] This question is independent of all previous parts.
Your colleague seeks you out again a different day, wanting to debate about the pros and cons of software pipelining. It's your turn to present a potential benefit of software pipelining.

Why might software pipelining be useful? Explain using **at most two sentences**.

Software pipelining allows us to only need overhead for starting and ending once overall, instead of once per iteration (lecture 14, slide 13).

(c) [3 pts] A friend learning the CS 152 / 252A material for the first time asks you about software pipelining vs loop unrolling – they're confused about the difference between the two, and would appreciate an explanation from you, a seasoned CS 152 / 252A student.

How is loop unrolling different from software pipelining? Explain using **at most two sentences**.

Loop unrolling means performing multiple iterations of the original loop in a single new iteration, or thus striding faster through the loop. On the other hand, software pipelining considers the same number of iterations as the original loop, just with potentially multiple iterations offset in execution at a single time.

Q4. [36 pts] (CS152 Only) Out of Order Execution

While out of order pipelines can detect and resolve most hazards through register renaming, memory instructions pose a special challenge that cannot be alleviated through simple renaming. For example, consider the following code snippet:

```
li x1, 0x100
li x2, 0x100
sw x0, 0(x1)
lw x3  0(x2)
```

Despite the `sw` and `lw` instructions using completely different registers, the `lw` is still dependent on the `sw` since it writes to the same memory location that the `lw` later reads from. In this way, the `lw` instruction has an *implicit WAR hazard* with the `sw` instruction. This is a challenging hazard to workaround, however, because it can only be detected when the address values in `x1` and `x2` resolve, which may occur at an unknown future time and in an unknown order.

In this problem we'll explore different techniques for handling this implicit dependency. You'll be asked to schedule assembly instructions for each policy by filling out a table similar to the one below:

Time								
Enter ROB	Issue	WB	Commit	Operation	Dest	Src1	Src2	Rolled Back?

This table is similar to the one seen in Homework 3, with the addition of a new `Rolled Back?` column, which you should fill with Y if the instruction throws an exception -OR- if the instruction has to be rolled back due to an exception and N if it does not. If an instruction results in an exception, you should fill out a second line in the table indicating its second execution after the exception is resolved. For example, consider the following assembly code:

```
    mul x1, x2, x3
    sub x2, x0, x0
```

Assume our CPU's `ALU` takes 6 cycles to execute a multiply, and 1 cycle for other integer arithmetic instructions. Stores and loads both take 2 cycles, but the load/store unit is pipelined so we can issue one load/store instruction per cycle. The CPU can fetch 1 instruction, dispatch 1 instruction, and commit 1 instruction per cycle. Our CPU can issue an instruction 1 cycle after the instruction enters the ROB, and can commit an instruction 1 cycle after writeback. Functional units write back to the ROB upon completion, and share a single write port to the ROB. In the case that two instructions write back in the same cycle, the older instruction writes back first. The instructions are committed in order and only one instruction may be committed per cycle. This OoO CPU has no forwarding available, and data-dependent instructions can issue 1 cycle after the instruction they depend on writes back. We can re-issue instructions 1 cycle after they throw an exception. If the `mul` instruction throws an exception on the sixth cycle of the overall execution, our table would look like:

Time								
Enter ROB	Issue	WB	Commit	Operation	Dest	Src1	Src2	Rolled Back?
1	2	-	-	mul	x1	x2	x3	Y // cycle 6
2	3	-	-	sub	x2	x0	x0	Y // mul exception
1	7	13	14	mul	x1	x2	x3	N // re-issued
2	8	9	15	sub	x2	x0	x0	N // re-issued

As shown in the table above, if an instruction gets re-issued, its Enter ROB cycle does not change (while its issue cycle does).

(a) [12 pts] First, we consider a conservative scheduling policy for loads and stores. In this policy, both loads and stores are blocked from issuing until all older stores can be confirmed to have non-conflicting destination addresses and, in case there is a conflict, any dependent stores have successfully written back. Assume that the address of any load and store is computed before the instruction enters the ROB, and is stored alongside the other metadata. This means we can detect address conflicts between load/store instructions before we issue the instruction.

Given the code snippet below, fill in the out-of-order scheduling table with the assumptions stated earlier using a conservative scheduling policy. Assume that initially register x8 holds the value 2, x7 != 0, and that the ROB has enough free entries to hold all the instructions at once.

```
add x5, x7, x0 // x7 + 0
add x3, x7, x7 // x7 + x7
mul x2, x7, x8 // x7 * 2
sw x1, 0(x2)
lw x4, 0(x3)
lw x6, 0(x5)
mul x6, x6, x6
```

Time								
Enter ROB	Issue	WB	Commit	Operation	Dest	Src1	Src2	Rolled Back?
1	2	3	(A) 4	add	x5	x7	x0	*N*
2	3	4	5	add	x3	x7	x7	*N*
3	4	10	11	mul	*x2*	*x7*	*x8*	(B) *N*
4	(C) 11	13	14	sw	*N/A*	*x2*	*x1*	*N*
5	(D) 14	16	17	lw	*x4*	*x3*	*N/A*	*N*
6	12	(E) 14	18	lw	*x6*	*x5*	*N/A*	*N*
7	15	21	(F) 22	mul	*x6*	*x6*	*x6*	(G) *N*

(B) and (G) are worth 1 point. All other boxes are worth 2 points.

(A)	4	(B)	N	(C)	11	(D)	14	(E)	14
(F)	22	(G)	N						

SID: ________________________

(b) [17 pts] While conservative scheduling does correctly resolve ordering hazards, in practice it tends to severely hurt performance as most loads and stores are unrelated and non-dependent. Thus, holding back loads until all older stores are known to be safe causes us to launch loads much later than necessary and leave a lot of performance on the table.

Instead, one common solution in real production processors is to *optimistically* issue loads as soon as their source addresses are available so long as there are no known dependencies at time of issue (i.e. an issued but not written back store to the same address). If the processor later discovers that it issued a load before a dependent store, it generates a "micro-exception" on the younger load, rolls it back, and restarts execution at the faulting load so that it can retrieve the correct value from memory. Assume that exceptions are detected before the commit stage, at which point any subsequent instructions that entered the ROB in-order will be rolled back as well.

For the same code snippet, fill in the out-of-order scheduling table using an "optimistic" scheduling policy. Assume that initially register x8 holds the value 2, x7 != 0, and that the ROB has enough free entries to hold all the instructions at once.

```
add x5, x7, x0
add x3, x7, x7
mul x2, x7, x8
sw x1, 0(x2)
lw x4, 0(x3)
lw x6, 0(x5)
mul x6, x6, x6
```

Time								
Enter ROB	Issue	WB	Commit	Operation	Dest	Src1	Src2	Rolled Back?
1	2	3	4	add	x5	x7	x0	*N*
2	3	4	5	add	x3	x7	x7	*N*
3	4	10	(A) 11	mul	*x2*	*x7*	*x8*	*N*
4	11	13	14	sw	*N/A*	*x2*	*x1*	(B) *N*
5	6	(C) 8	–	lw	*x4*	*x3*	*N/A*	*Y*
6	7	9	–	(D) lw	*x6*	*x5*	*N/A*	(E) *Y*
7	(F) 10	–	–	mul	*x6*	*x6*	*x6*	*Y*
5	14	16	(G) 17	lw	*x4*	*x3*	*N/A*	(H) *N*
6	12	14	(I) 18	lw	*x6*	*x5*	*N/A*	*N*
7	15	21	22	(J) mul	*x6*	*x6*	*x6*	*N*

(B), (E), and (H) are worth 1 point. All other boxes are worth 2 points.

(A)	11	(B)	N	(C)	8	(D)	lw	(E)	Y
(F)	10	(G)	17	(H)	N	(I)	18	(J)	mul

(c) [7 pts] **This part may be completed independently from the others**.
Although optimistic scheduling can be great and allow the processor to issue loads sooner, its rollbacks are rather expensive and often force the processor to unnecessarily redo work. **In four sentences or fewer**, describe a potential modification which could be made to the processor to reduce the frequency or cost of these rollbacks while still providing better performance than the conservative scheduling policy.
Hint: Many solutions exist. You might consider adding new micro-architectural structures to the processor, working around the need to rollback, or changing how rollbacks themselves are handled.

One common solution in real world processors is to *predict* whether a load has a dependent store and, if so, which store it is dependent on. Loads which are predicted to have dependencies are held back until it can be determined that the prediction was incorrect or, if it was correct, the dependent store completes. This provides the best of both optimistic and conservative scheduling as it allows the processor to learn where rollbacks typically happen and selectively stall instructions in order to avoid needing to rollback.

—or—

Another solution is to make rollbacks more fine grained. Currently, the optimistic scheduler 's micro-exception completely restarts execution at the faulting load in order to undo the damage caused by the load returning incorrect data. This leads to unrelated and correct computations which did not consume the incorrect load data being thrown away. By tracking which instructions consumed this invalid data (i.e. speculative taint tracing) and selectively reissuing only impacted ones, the cost of a rollback can be decreased.

—or—

A third potential solution is to allow loads to *issue* optimistically but only *write back* conservatively. Conservative write back means that loads are only written back once all older stores have resolved and are determined to be non-dependent. If an older dependent store is found, the stored data can be forwarded to the load ("store to load forwarding") or the load can be replayed. This yields a performance improvement over completely conservative scheduling as loads can still be issued optimistically (and thus once verified to be safe can be written back much sooner) while also eliminating the need to rollback on conflict (dependent instructions are never issued with incorrect data). This yields a somewhat meager but still valuable speedup.

Q5. [18 pts] Vector Processors

For the following problems, you are provided a simplified vector ISA. Refer to Appendix 1 at the end of the exam for exact ISA semantics.

(a) You are provided the following vector code snippet representing a dot product of vector length (VL).

```
vld v1, (x1) # vector load
vld v2, (x2)
vmul v3, v2, v1 # vector multiply
vredsum v4, v3 # vector reduce sum
vst v4, (x3), 0 # vector store
```

For the following sub-parts, compute the number of cycles with the following **important assumptions**:

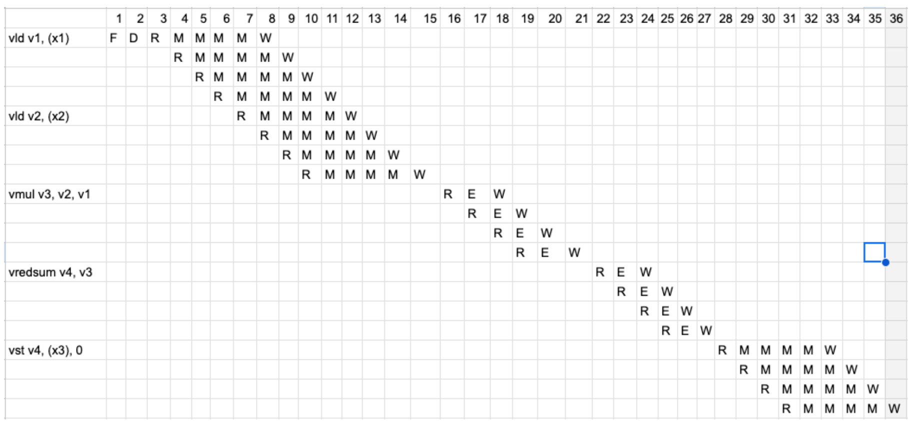
- Vector length is 32 elements
- 8 vector lanes
- 32 elements per vector register
- One ALU per lane: 1 cycle latency
- One load/store unit per lane: 4 cycle latency, fully pipelined
- No dead time
- Magic memory system with no bank conflicts or cache misses
- Reduction operations (i.e. `vredsum`) work like arithmetic vector instructions (i.e. like `vmul`)

A vector instruction is also fetched (F) and decoded (D). Then, it stalls (-) until its required vector functional unit is available. With no chaining, a dependent vector instruction stalls until the previous instruction finishes writing back all of its elements. A vector instruction is pipelined across all the lanes in parallel. For each element, the operands are read (R) from the vector register file, the operation executes on the load/store unit (M) or the ALU (X), and the result is written back (W) to the vector register file.

(i) [3 pts] How many cycles would it take without vector chaining?

36 or 35 (if the final W of the store is not counted)

Scratch space (non-graded)



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
vld v1, (x1)	F	D	R	M	M	M	M	W																												
				R	M	M	M	M	W																											
					R	M	M	M	M	W																										
						R	M	M	M	M	W																									
vld v2, (x2)							R	M	M	M	M	W																								
								R	M	M	M	M	W																							
									R	M	M	M	M	W																						
										R	M	M	M	M	W																					
vmul v3, v2, v1																R	E	W																		
																	R	E	W																	
																		R	E	W																
																			R	E	W															
vredsum v4, v3																						R	E	W												
																							R	E	W											
																								R	E	W										
																									R	E	W									
vst v4, (x3), 0																												R	M	M	M	M	W			
																													R	M	M	M	M	W		
																														R	M	M	M	M	W	
																															R	M	M	M	M	W

(ii) [3 pts] How many cycles would it take with vector chaining? Vector loads/stores share the load-store unit while vector arithmetic instructions have their own unique execution units. Vector chaining is done through the register file and an element can be read (R) in the same cycle in which it is written back (W), or it can be read on any later cycle (the chaining is flexible). Assume only one set of vector lanes (8 vector lanes) can write back in a single cycle.

28 or 27 (if the final W of the store is not counted)

Scratch space (non-graded)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
vld v1, (x1)	F	D	R	M	M	M	M	W																				
				R	M	M	M	M	W																			
					R	M	M	M	M	W																		
						R	M	M	M	M	W																	
vld v2, (x2)							R	M	M	M	M	W																
								R	M	M	M	M	W															
									R	M	M	M	M	W														
										R	M	M	M	M	W													
vmul v3, v2, v1												R	E			W												
													R	E			W											
														R	E			W										
															R	E			W									
vredsum v4, v3																R	E			W								
																	R	E			W							
																		R	E			W						
																			R	E			W					
vst v4, (x3), 0																				R	M	M	M	M	W			
																					R	M	M	M	M	W		
																						R	M	M	M	M	W	
																							R	M	M	M	M	W

(b) You are now provided the following C++ code for a function that will be hand-translated into vectorized code.

```
void data_parallel_func(uint8_t iterations) {
    for (uint8_t i = 0; i < iterations; ++i) {
        # vectorizable inner loop operations
    }
}
```

First, let's assume we want to and can convert this code into vector code.

(i) [4 pts] Assume `iterations` is greater than the max vector length (MVL) of the machine, so `iterations > MVL`. What can you do to improve the performance of the code and/or processor (do the proper number of calculations)? Explain in **at most two** sentences.

Strip mine the code so that you iterate on MVL items per loop iteration.

(ii) [4 pts] Assume the `iterations` is less than the vector length (VL) of the machine, so `iterations < VL`. What can you do to improve the performance of the code and/or processor (do the proper number of calculations)? Explain in **at most two** sentences.

You can configure the vector length (VL) to be exactly the number of iterations so that you only need to operate on the elements necessary.

(iii) [4 pts] Assume the `iterations` is less than the vector length (VL) of the machine, so `iterations < VL`. Additionally, assume that you have a **packed SIMD** implementation of the code above. In comparison to the vector implementation, what is one downside of the packed SIMD implementation? Explain in **at most two** sentences.

Packed SIMD cannot configure vector lengths like the vector implementation can. This means the packed SIMD implementation will compute unused elements since we cannot save compute by configuring vector lengths as we did under the vector implementation.

Q6. [31 pts] Multithreading

(a) You are presented with the multithreaded C code below, which is compiled into the following RISC-V assembly. Assume that N is a multiple of num_threads.

```
---------- C ----------
float dot[N];
float A[N];
float B[N];

...

// MULTITHREADED CODE
// tid         : thread id
// num_threads : # of threads
// N           : # of array elements
for (int i = tid * (N / num_threads); i < (tid + 1) * (N / num_threads); i++) {
  dot[i] = A[i] * B[i];
}
---------- C ----------

---------- RISC-V ----------
// x1 : &A[tid * (N / num_threads)]
// x2 : &B[tid * (N / num_threads)]
// x3 : &dot[tid * (N / num_threads)]
// x6 : tid * (N / num_threads)
// x7 : (tid + 1) * (N / num_threads)
loop: fld  f1, 0(x1)
      fld  f2, 0(x2)
      fmul f3, f1, f2
      fsd  f3, 0(x3)
      addi x1, x1, 4
      addi x2, x2, 4
      addi x3, x3, 4
      addi x6, x6, 1
      bne  x6, x7 loop
---------- RISC-V ----------
```

We are running this program with two threads on a simple single-core in-order 5-stage (IF -> ID -> EX -> MEM -> WB) multithreaded pipeline similar to the one presented in lecture. Integer operations begin execution in the EX stage, and after 1 cycle the result is available for forwarding to any stage. Floating point operations begin execution in the EX stage, and after 5 cycles the result is available for forwarding to any stage. Memory operations begin execution in the MEM stage, and after 1 cycle (assume the L1 cache always hits for this question) the result is available for forwarding to any stage. Assume that there are two of each functional unit in the EX stage for integer and floating point operations. In other words, there can be two instructions from separate threads in the EX stage at a time, but this is not true for other stages. **FLD** and **FSD** instructions count as memory operations and not floating point operations. There are two different instruction scheduling policies that we can pick between:

Fine-grained Multithreading: Switch threads for every new instruction in a round-robin fashion.

Course-grained Multithreading: Run X instructions from each thread before switching to a new thread in a round-robin fashion. Assume we are scheduling with a granularity of 2 (X == 2).

Schedule a single iteration of the code above using fine-grained and course-grained multithreading. For the instruction and thread that is being scheduled, indicate the cycle on which the instruction entered the pipeline **(Start Cycle)**, and the # of cycles that this instruction needs to stall in the pipeline due to dependencies or structural hazards with previous instructions **(Cycles Stalled)**. Assume that x1, x2, x3, x6, and x7 have their values already set in the register files before execution of the above code begins. The RISC-V code is provided for reference.

```
---------- RISC-V ----------
// x1 : &A[tid * (N / num_threads)]
// x2 : &B[tid * (N / num_threads)]
// x3 : &dot[tid * (N / num_threads)]
// x6 : tid * (N / num_threads)
// x7 : (tid + 1) * (N / num_threads)
loop: fld  f1, 0(x1)
      fld  f2, 0(x2)
      fmul f3, f1, f2
      fsd  f3, 0(x3)
      addi x1, x1, 4
      addi x2, x2, 4
      addi x3, x3, 4
      addi x6, x6, 1
      bne  x6, x7 loop
---------- RISC-V ----------
```

Fine-grained Multithreading

Instruction	Thread ID	Start Cycle	Cycles Stalled
fld f1, 0(x1)	0	0	0
fld f1, 0(x1)	1	1	0
fld f2, 0(x2)	0	2	0
fld f2, 0(x2)	1	(A) 3	0
fmul f3, f1, f2	0	4	(B) 0
fmul f3, f1, f2	1	5	0
fsd f3, 0(x3)	0	(C) 6	(D) 4
fsd f3, 0(x3)	1	7	(E) 4
addi x1, x1, 4	0	(F) 12	0
addi x1, x1, 4	1	13	0
addi x2, x2, 4	0	14	0
addi x2, x2, 4	1	15	0
addi x3, x3, 4	0	16	0
addi x3, x3, 4	1	(G) 17	0
addi x6, x6, 1	0	18	0
addi x6, x6, 1	1	19	(H) 0
bne x6, x7 loop	0	20	0
bne x6, x7 loop	1	21	0

(i) [8 pts] What are the cycle numbers for the cells labeled as A-H?

(A) 3 (B) 0 (C) 6 (D) 4 (E) 4

(F) 12 (G) 17 (H) 0

```
---------- RISC-V ----------
// x1 : &A[tid * (N / num_threads)]
// x2 : &B[tid * (N / num_threads)]
// x3 : &dot[tid * (N / num_threads)]
// x6 : tid * (N / num_threads)
// x7 : (tid + 1) * (N / num_threads)
loop: fld  f1, 0(x1)
      fld  f2, 0(x2)
      fmul f3, f1, f2
      fsd  f3, 0(x3)
      addi x1, x1, 4
      addi x2, x2, 4
      addi x3, x3, 4
      addi x6, x6, 1
      bne  x6, x7 loop
---------- RISC-V ----------
```

Course-grained Multithreading

Instruction	Thread ID	Start Cycle	Cycles Stalled
fld f1, 0(x1)	0	0	0
fld f2, 0(x2)	0	1	0
fld f1, 0(x1)	1	(A) 2	0
fld f2, 0(x2)	1	3	(B) 0
fmul f3, f1, f2	0	4	0
fsd f3, 0(x3)	0	5	(C) 4
fmul f3, f1, f2	1	(D) 6	4
fsd f3, 0(x3)	1	(E) 11	4
addi x1, x1, 4	0	12	(F) 4
addi x2, x2, 4	0	(G) 17	0
addi x1, x1, 4	1	18	0
addi x2, x2, 4	1	19	0
addi x3, x3, 4	0	20	0
addi x6, x6, 1	0	21	0
addi x3, x3, 4	1	22	0
addi x6, x6, 1	1	23	0
bne x6, x7 loop	0	24	0
bne x6, x7 loop	1	(H) 25	0

(ii) [8 pts] What are the cycle numbers for the cells labeled as A-H?

(A) 2 (B) 0 (C) 4 (D) 6 (E) 11

(F) 4 (G) 17 (H) 25

(b) [3 pts] Which scheduling policy is better for this piece of code and why? Explain using **at most two sentences**.

The fine-grained scheduling worked better for this piece of code, because it prevented stalls incurred between consecutive and dependent instructions better than the course-grained scheduling policy.

SID: ________________________

(c) [10 pts] We are now running the same program with two threads on a single-core dual-issue out-of-order multithreaded pipeline. This pipeline allows two instructions at a time (potentially from different threads) to enter execution on the same clock cycle, and allows instructions within threads to execute out-of-order.

Assume that integer operations have a 1-cycle functional delay upon issue, floating point operations have a 5-cycle functional delay upon issue, and Memory operations have a 2-cycle functional delay upon issue. Fill in the following table with the instruction, the thread id, the cycle upon which the instruction is issued, and the # of cycles that this instruction needs to stall in the pipeline due to dependencies or structural hazards with previous instructions. Assume all of both threads' instructions are already present in the ROB. The first two rows have been filled out for you.

For parts c and d, assume that any number of instructions can write back to the ROB in a single cycle.

```
---------- RISC-V ----------
// x1 : &A[tid * (N / num_threads)]
// x2 : &B[tid * (N / num_threads)]
// x3 : &dot[tid * (N / num_threads)]
// x6 : tid * (N / num_threads)
// x7 : (tid + 1) * (N / num_threads)
loop: fld  f1, 0(x1)
      fld  f2, 0(x2)
      fmul f3, f1, f2
      fsd  f3, 0(x3)
      addi x1, x1, 4
      addi x2, x2, 4
      addi x3, x3, 4
      addi x6, x6, 1
      bne  x6, x7 loop
---------- RISC-V ----------
```

Simultaneous Multithreading

Instruction 1	Thread ID	Issue Cycle	Cycles Stalled	Instruction 2	Thread ID	Issue Cycle	Cycles Stalled
fld f1, 0(x1)	0	0	0	fld f2, 0(x2)	0	0	0
fld f1, 0(x1)	1	1	0	fld f2, 0(x2)	1	1	0
fmul f3, f1, f2	0	2	0	addi x1, x1, 4	0	2	0
fmul f3, f1, f2	1	3	0	addi x1, x1, 4	1	3	0
addi x2, x2, 4	0	4	0	addi x2, x2, 4	1	4	0
addi x3, x3, 4	0	5	0	addi x3, x3, 4	1	5	0
addi x6, x6, 1	0	6	0	addi x6, x6, 1	1	6	0
fsd f3, 0(x3)	0	7	0	bne x6, x7 loop	0	7	0
fsd f3, 0(x3)	1	8	0	bne x6, x7 loop	1	8	0

(d) [2 pts] How many total stalls across all instructions are necessary?

0

Q7. [29 pts] (CS252 Only) Branch Prediction

(a) In class, we introduce branch prediction with a single prediction done per cycle (in a non-superscalar fetch/decode processor). However, modern out-of-order machines support superscalar pipelines (fetch/decode/execute multiple instructions) that can exploit more ILP. These pipelines often come with superscalar branch prediction (can predict multiple branch instructions per cycle). Assume you are given a modern processor with a four-wide fetch pipeline (it can fetch four instructions per cycle). For the following code snippets, is it sufficient to have non-superscalar prediction for maximum performance assuming you always fetch four instructions per cycle? Mark yes/no and write no more than three sentences of explanation.

(i) [2+3 pts]

```
      li x1, 0x100
      li x2, 0x200
      li x6, 0x300
loop: lw x3  0(x1)
      lw x4  0(x2)
      add x5, x3, x4
      sw x5, 0(x6)
      addi x1, x1, #4
      addi x2, x2, #4
      addi x6, x6, #4
      bne x0, x1, loop
      addi x1, x1, #4
      addi x2, x2, #4
      addi x3, x3, #4
      ret
```

[✓ Yes. Non-superscalar prediction is sufficient.]

No. Superscalar prediction is needed.

Yes. Non-superscalar prediction is sufficient because, at any point in this loop, there is only one control flow instruction in a 4-wide window.

(ii) [2+3 pts]

```
      li x1, 0x100
      li x2, 0x200
      li x6, 0x300
loop: mul x2, x1, x1
      subi x2, x2, #1
      bne x0, x1, loop
      ret
      <non-control flow instructions>
```

Yes. Non-superscalar prediction is sufficient.

[✓ No. Superscalar prediction is needed.]

Superscalar prediction is needed because you have multiple control flow instructions within a 4-inst. window range and you need to predict every control flow instruction. Bug: Technically, the ret will never be reached (x0 != x1)

(b) [4 pts] You are now given a state-of-the-art branch predictor that is only able to predict one branch per cycle despite the machine being a four-wide fetch machine. What other metadata can be used to reduce branch aliasing? Explain in under 3 sentences.

You can also store the index of the branch in the fetch packet so that when a prediction is made it not only predicts the branch but verifies that it is the branch wanted.

(c) Assume this state-of-the-art single-prediction per cycle predictor (given in Part B) stores its predictor state bimodal table that is mapped to a ported SRAM. What is the amount of SRAM read and write ports needed to support full-throughput predictions?

(i) [1 pt] What is the amount of SRAM read ports needed to support full-throughput predictions? 2

(ii) [1 pt] What is the amount of SRAM write ports needed to support full-throughput predictions? 1

(iii) [4 pts] Please explain your read/write port reasoning in under three sentences.

You need 1 read port for reading out predictions, then on updating the counter you need to read the original counter value (1 more read port) and another port to write the new value (1 write port).

(iv) [4 pts] What is a single way we can reduce the number of ports needed (read and/or write)?

You can pass along the read prediction value through the pipeline, then on update you can use that existing value instead of having to re-read it (saving a read port).

(d) [5 pts] As described in class, we mention that updates to the branch predictor state only happen once the branch commits (i.e. when you know the outcome of the branch. Why is this bad for the following code if the pipeline is deep? Additionally, provide a hardware-based mechanism to fix the issue. Please explain in under **five sentences**.

```
      li x1, 0x100
      li x2, 0x200
      li x6, 0x300
loop: mul x2, x1, x1
      subi x2, x2, #1
      bne x0, x1, loop
      ret
      <non-control flow instructions>
```

Since we delay the update until commit, the branch predictor state may not update in time to handle tight loops (especially if the pipeline is very deep). Thus we should speculatively update the BPU and only on incorrect prediction (known at completion) we roll back the state of the prediction with extra metadata.

Q8. [35 pts] (CS252 Only) MIPS R10000

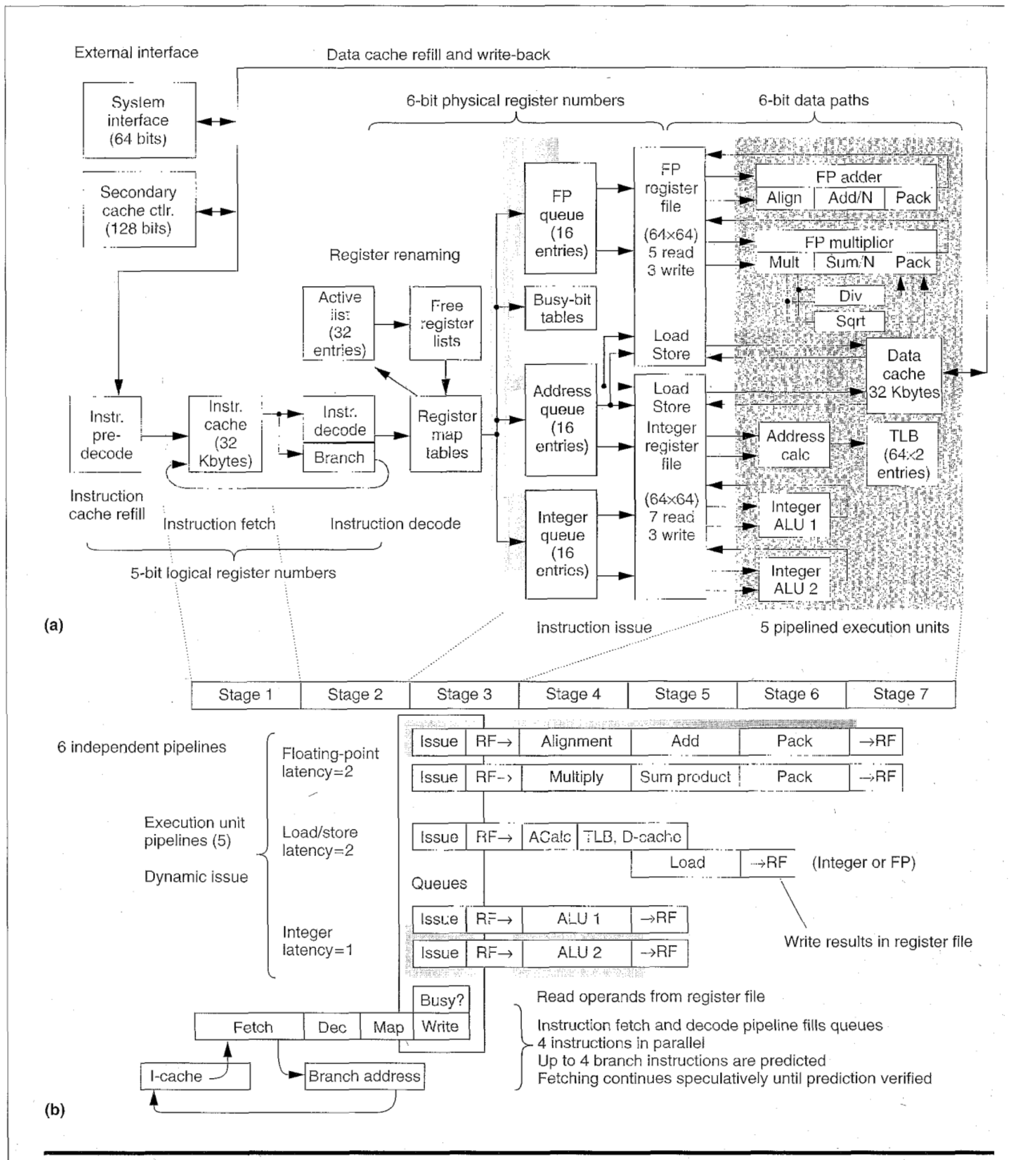


Figure 2. R10000 block diagram (a) and pipeline timing diagram (b). The block diagram shows pipeline stages left to right to correspond to pipeline timing.

Figure 1: MIPS R10000 Pipeline Diagram

Above is the pipeline block diagram of the MIPS R10000, an important design towards modern out-of-order processors that we discussed in paper discussions. Use this diagram to answer the questions on the following page. **Limit your responses for parts (a) and (b) to no more than 3 sentences, and parts (b) onwards to 4 sentences.**

(a) The MIPS R10000 uses a 4-entry branch stack for speculative instructions.

(i) [6 pts] For each decoded branch, what data/structures must the processor save to restore state upon a mispredicted branch? No explanation needed.

Alternate branch address, copies of integer and FP map tables, control bits

(ii) [4 pts] Speculation may cause unneeded cache refills. If a branch is determined to be **not taken**, should the refills still be completed?

Yes, because program execution may soon take the other direction of the branch

(b) [5 pts] The active list contains the logical-destination register number and its old physical-register number for each instruction. Contrast the role of the active list upon graduation vs on exception

On graduation, send instruction tag to active list to set done bit, free old physical-logical mapping.
On exception, restore old mappings from active list

(c) [7 pts] Describe a mechanism by which such a processor may track dependencies between pending loads and stores in the address queue.

The address queue uses a 16x16 matrix to track dependencies between load and store instructions in the queue. Matrix tracks instructions that load the same bytes as a pending store instruction.

(d) [7 pts] The MIPS R10000 currently has 1 FP adder and 1 FP multiplier. What are the additional hardware components we need to add if we wanted to add an additional FP unit?

Additional register file read/write ports (2 read, 1 write), bypass paths from FPU to queue operands

(e) [6 pts] For an instruction that depends on a load instruction, MIPS R10000 tentatively issues the instruction one cycle before it is executed, while the load reads the data cache. What is the load latency in this case? What would happen if the load fails?

2 cycle load latency. If load fails, instruction is aborted

1 Appendix: Simplified Vector Processor ISA

Note: `VL` stands for the vector length and `VE` stands for the vector element size. Vector, scalar, and mask registers are denoted by the `v`, `r`, and `f` prefix, respectively (e.x. `vd`, `rs1`, `fs1`).

- `vld vd, (rs1)`: loads consecutive `VL` values of size `VE` from memory located at `R[rs1]` into vector register `vd`, i.e. `vd = Mem[R[rs1]]`
- `vld vd, vs1`: loads `VL` values from memory located at the elements of vector register `vs1` (i.e. each element of `vs1` is a memory address to load from) into vector register `vd`, i.e. `vd = Mem[vs1]`
- `vst vd, (rs1)`: stores consecutive `VL` values of size `VE` from vector register `vd` into memory located at `R[rs1]`, i.e. `Mem[R[rs1]] = vd`
- `vst vd, vs1`: stores values from vector register `vd` into memory located at the elements of vector register `vs1` (i.e. each element of `vs1` is a memory address to store to) i.e. `Mem[vs1] = vd`
- `vst vd, (rs1), imm`: stores the first consecutive `imm` values of size `VE` from vector register `vd` into memory located at `R[rs1]`, i.e. `Mem[R[rs1]] = vd`
- `vadd vd, vs2, vs1`: adds `vs1` to `vs2` into `vd` element-wise, i.e. `vd = vs1 + vs2`
- `vsub vd, vs2, vs1`: subtracts `vs2` from `vs` into `vd` element-wise, i.e. `vd = vs1 - vs2`
- `vmul vd, vs2, vs1`: multiplies `vs1` by `vs2` into `vd` element-wise, i.e. `vd = vs1 * vs2`
- `vdiv vd, vs2, vs1`: divides `vs1` by `vs2` into `vd` element-wise, i.e. `vd = vs1 / vs2`
- `vredsum vd, vs1`: reduces the sum of elements of `vs1` into the first element of `vd` i.e. $\texttt{vd[0]} = \sum_{\texttt{i=0}}^{\texttt{VL}} \texttt{vs1[i]}$
- `vgt vs1, fd`: for each bit in the mask register `fd`, sets the bit to 1 if the corresponding element in `vs1` is greater than 0; otherwise, sets the corresponding bit to 0 in the mask register, i.e. `fd[i] = vs1[i] > 0 ? 1 : 0` for `i` from 0 to `VL`

  Note that after this instruction, `vd` will now be masked for any future instructions.