CS 152 Computer Architecture and Engineering
CS 252A Graduate Computer Architecture

Midterm #2

SOLUTIONS

April 14, 2021

Professor Krste Asanović

Name:________________________

SID:________________________

I am taking CS152 / CS252A

(circle one)

80 Minutes, 4 Questions

Notes:

- Not all questions are of equal difficulty, so look over the entire exam!
- Please carefully state any assumptions you make.
- Please write your name on every page in the exam.
- Do not discuss the exam with other students who haven't taken the exam.
- If you have inadvertently been exposed to an exam prior to taking it, you must tell the instructor or TA.
- You will receive no credit for selecting multiple-choice answers without giving explanations if the instructions ask you to explain your choice.

Question	CS152 Point Value	CS252A Point Value
1	20	20
2	20	20
3	16	16
4	18	28
TOTAL	74	84

Problem 1: Out-of-order Execution (20 Points) (CS152)
We execute the following function on an out-of-order core with a unified physical register file.

```
void scale_6(int* a0, int a1) {
  for (int i = 0; i < 6; i++) {
    a0[i] = a0[i] * a1;
  }
}
```

```
      addi  t1, a0, 0x18
loop: lw    t0, 0(a0)
      mul   t0, t0, a1
      sw    t0, 0(a0)
      addi  a0, a0, 0x4
      bnez  a0, t1, loop
      ret
```

1.A (10 points) OOO Structures
Fill out the contents of the reorder-buffer, freelist, and map table as the decode unit would until the ROB is full. Additionally, indicate the final values for the ROB head and tail pointers. The first instruction has been filled out for you. Assume no instruction will complete execution.
Note: The instruction column of the ROB is optional and will not be graded. Space is provided to assist you in bookkeeping.

FIFO Free List															
~~p11~~	~~p12~~	~~p13~~	~~p14~~	~~p15~~	~~p16~~	p17	p18	p19							

Map Table								
a0	~~p0~~	p14						
a1	p1							
t0	~~p2~~	~~p12~~	~~p13~~	~~p15~~	p16			
t1	~~p3~~	p11						

IDX	Instruction	PRd	LPRd
0	bnez		
1	lw t0	p15	p13
2	mul t0	p16	p15
3	addi t1	p11	p3
4	lw t0	p12	p2
5	mul t0	p13	p12
6	sw		
7	addi a0	p14	p0

Final ROB Head:	3	**Final ROB Tail:**	2

Problem 1: (20 Points) Out-of-order Execution (CS152)
We execute the following function on an out-of-order core with a unified physical register file.

```
int acc_6(int* a0) {
  int a1 = 0;
  for (int i = 0; i < 6; i++) {
    a1 += a0[i];
  }
  return a1;
}
```

```
      addi  t1, a0, 0x18
      addi  a1, zero, 0x0
loop: lw    t0, 0(a0)
      add   a1, a1, t0
      addi  a0, a0, 0x4
      bnez  a0, t1, loop
      mv    a0, a1
      ret
```

1.A (10 points) OOO Structures
Fill out the contents of the reorder-buffer, freelist, and map table as the decode unit would until the ROB is full. Additionally, indicate the final values for the ROB head and tail pointers. The first instruction has been filled out for you. Assume no instruction will complete execution.
Note: The instruction column of the ROB is optional and will not be graded. Space is provided to assist you in bookkeeping.

FIFO Free List															
~~p11~~	~~p12~~	~~p13~~	~~p14~~	~~p15~~	~~p16~~	~~p17~~	p18	p19							

Map Table								
a0	~~p0~~	p15						
a1	~~p1~~	~~p12~~	~~p14~~	p17				
t0	~~p2~~	~~p13~~	p16					
t1	~~p3~~	p11						

ROB

IDX	Instruction	PRd	LPRd
0	bnez		
1	lw t0	p16	p13
2	add a1	p17	p14
3	addi t1	p11	p3
4	addi a1	p12	p1
5	lw t0	p13	p2
6	add a1	p14	p12
7	addi a0	p15	p0

Final ROB Head:	3	**Final ROB Tail:**	2

1.B (4 points) Precise exceptions
Note: For this problem, you should not update the structures in 1A.
After the ROB becomes full, there is a protection fault on the first load in the instruction sequence. The ROB recovers architectural state using the iterative exception recovery procedure discussed in class.

How many cycles does it take to recover architectural state?

scale_6: 7 cycles, 1 for each instruction between ROB tail and the load, including the load
acc_6: 6 cycles

Which registers are returned to the free list during this process?

p12-p16, inclusive in scale_6
p13-p17, inclusive in acc_6

1.C (3 points) Checkpointing
To provide fast pipeline restarts after branch mispredictions, some structures and values must be checkpointed when branches are dispatched. Circle which of the following structures must be checkpointed for fast pipeline restarts. No explanation is necessary. Assume all buffers are implemented as circular buffers, with head and tail pointers.

- ROB head pointer
- ROB tail pointer
- Physical register contents
- Physical register present bits
- Issue queue contents
- Physical register free list
- Architectural register map table
- Speculative store buffer head pointer
- Speculative store buffer tail pointer

1.D (3 points) Branch prediction
You build a PC-indexed BHT to improve prediction accuracy of the branch in this loop. What is the prediction accuracy for the conditional branch after this function is called many times when using 1-bit counters?

1-bit counters will always mispredict both the entry and the exit, while the 2-bit saturating counter will only mispredict the exit.

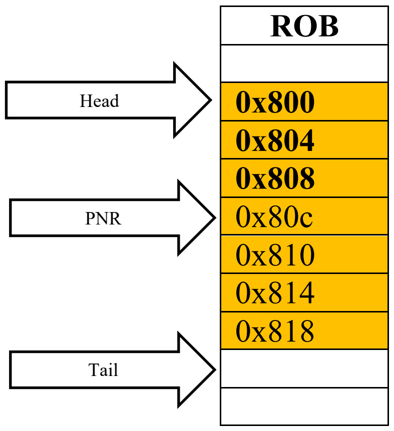
Prediction accuracy with 1-bit counters = ________4/6__________

What is the prediction accuracy for the conditional branch after this function is called many times when using 2-bit saturating counters?

Prediction accuracy with 2-bit counters = ________5/6___________

Problem 1: (20 Points) Out-of-order Execution (CS252A)

You consider implementing another pointer in the ROB, the “Point-of-no return” (PNR) pointer. The PNR pointer sits between the ROB head, and tail. Instructions between the PNR and the ROB head **are guaranteed to eventually commit**. In the diagram below, the highlighted entries indicate valid entries in the ROB, while the bolded instructions indicate those that are guaranteed to eventually commit (will never be squashed).



1.A (5 points) Point-of-no-return policy

Describe a policy for incrementing the PNR pointer, that maximizes the number of instructions between the PNR and the ROB head.

Increment the PNR pointer if any of the following is true for the instruction the PNR is pointing to.

1. The instruction may never cause an exception or PC redirection
2. The instruction is a branch, and has been completed (resolved)
3. The instruction is a memory instruction, has been completed, and will never cause an address disambiguation misspeculation

1.B (5 points) Point-of-no-return safety

You consider freeing stale physical registers for instructions between the PNR and the ROB head, ahead of when the instructions commit. Under what conditions is this optimization safe to apply?

This is safe if all other instructions between the PNR and the ROB head which write to or read from the physical register have either completed (for writes) or read their operands (for reads).

Note: Following questions are unrelated to 1.A/1.B

1.C (5 points) MOV optimization

The "`mv rd, rs1`" instruction copies the contents of `rs1` into the register rd. In a core with register renaming, this instruction can be implemented by allowing multiple architectural registers to map to the same physical register, avoiding an unnecessary copy. How does this optimization impact recovery from mispredictions?

The challenge is maintaining a free list. Since multiple architectural registers might point to the same physical register, we can no longer free physical registers using the simple scheme whenever we remap an architectural register, either at commit, or during misprediction recovery.

Some form of reference-counting would be necessary. A physical register can only be freed when no inflight instructions reference it.

1.D (5 points) History registers

You observe that a long global history register captures the dynamic execution path of the program, and that long global history registers are positively correlated with branch prediction performance. You develop an alternative formulation of a history register, called an "infinite path history". In your proposed formulation, the "infinite path history" register is a 32 bit register that is repeatedly hashed with the target PC of taken branches or jumps, to capture infinite history from dynamic execution.

PHIST <= hash(PHIST, target_pc)

Discuss the merits of this approach.

The key is that temporal and spatial correlation is finite, as all programs are finite. In an infinite history register, noise from information arbitrarily far into the past would mask any spatial or temporal correlations carried in the history register.

Problem 2: Multithreading (20 points)

2.A (12 points) True/False

For each statement below, indicate whether that statement is true or false and briefly explain your reasoning.

Adding multithreading to a single-issue out-of-order core will in general yield a greater relative performance improvement than adding multithreading to a similar single-issue in-order core with an identical memory hierarchy.

False. An out-of-order core can already reduce vertical waste by exploiting more ILP. An in-order core will stall much more frequently than an out-of-order core, so there is more potential for performance improvement by hiding stalls using multithreading.

A multithreaded processor does not benefit from branch prediction with a sufficiently large number of threads.

True. With a sufficiently large number of threads, branch resolution latency will be completely hidden since the core will be busy executing instructions from other threads.

The ISA must contain special multithreading instructions to support multithreading.

False. From the perspective of the ISA, a multithreaded processor is architecturally identical to a multicore system. No special instructions are necessary, as switching between threads is handled at the hardware level.

2.A (8 points) Performance

Consider execution of the following code on a single-issue in-order processor. Loads have 50-cycle latency and are fully pipelined. Adds have 4-cycle latency and are fully pipelined. Assume perfect branch prediction and that branches execute in one cycle.

(Alternate version: 70-cycle load latency, 5-cycle add latency)

```
// pointer chasing
loop: addi a1, a1, -0x1
      lw a0, 0(a0)
      bnez a1, loop
```

We add multithreading to this processor.

Fixed Switching: How many threads are needed to avoid stalls if threads are switched every cycle in a fixed round-robin schedule? Show your work.

The longest stall is the load-to-load dependency between consecutive iterations. If thread 0 executes a `lw` on cycle 0, the next `lw` will happen on cycle 3N, where N is the number of threads.

$50 > 3N$
The number of threads is at least 17.

In the alternate version, the equation is $70 > 3N$, so number of threads is at least 24.

Data-dependent Switching: How many threads are needed to avoid stalls if threads are switched only when an instruction cannot issue due to a data dependency?

`lw` → `bnez` → `addi` can be issued without stalls from a single thread. Then there are 50 - 3 cycles of latency to hide before the next `lw` can execute again.

$\text{ceil}((50 - 3) / 3) + 1 = 17$ threads, again

In the alternate version, the equation is $\text{ceil}((70 - 3) / 3) + 1 = 24$ threads.

Problem 3: VLIW (16 points)

In this problem, we will optimize the following code sequence, which performs a scatter operation with scaling, for a VLIW architecture.

```
loop:
     flw  ft0, 0(a0)        # x = src[i]
     lw   t0, 0(a1)         # p = idx[i]
     addi a0, a0, 0x4       # bump idx
     addi a1, a1, 0x4       # bump src
     addi a2, a2, -0x1      # decrement n
     fmul ft1, ft0, fa0     # scale x
     fsw  ft1, (t0)         # *p = x
     bnez a2, loop
```

- The source, index, and destination arrays do not overlap in memory. (Assume there are no memory-memory dependencies.)
- The number of iterations (initial value of a2) is greater than 0.
- Assume that no exceptions arise during execution.

The code is executed on an in-order VLIW machine with four execution units. All execution units are fully pipelined and latch their operands at issue.

- Two integer ALUs, 1-cycle latency, also used for branches
- One load/store unit, 3-cycle latency for loads (*Alternate version: 2-cycle latency*)
- One floating-point unit, 2-cycle latency (*Alternate version: 3-cycle latency*)

Instructions are statically scheduled with no interlocks; all latencies are exposed in the ISA. All register operands are read before any writes from the same instruction take effect (i.e., no WAR hazards between operations within a single VLIW instruction).

Execution units write to the register file at the end of their last pipeline stage, and the results become visible at the beginning of the following cycle. There is no bypassing. **Old values can be read from registers until they have been overwritten.** (You may leverage this to more efficiently schedule VLIW code.)

The unoptimized scheduling of the above assembly code is shown in the following table.

Label	ALU0	ALU1	MEM	FPU
`loop:`	`addi a0, a0, 0x4`		`flw ft0, 0(a0)`	
	`addi a1, a1, 0x4`		`lw t0, 0(a1)`	
	`addi a2, a2, -0x1`			
				`fmul ft1,ft0,fa0`
	`bnez a2, loop`		`fsw ft1, 0(t0)`	

3.A (12 points) Software Pipelining

Schedule operations with VLIW instructions using only software pipelining (no loop unrolling). Include the prologue and epilogue to initiate and drain the software pipeline. Try to optimize for efficiency and minimize the number of cycles, but prioritize correctness. Your code should behave correctly for any `a2` > 0.

You do not have to write out the full instructions unless they differ from the original – just the opcode and the destination register are sufficient. Entries for NOPs can be left blank.

Label	ALU0	ALU1	MEM	FPU
	`addi a0, a0, 0x4`	`addi a2, a2, -0x1`	`flw ft0, 0(a0)`	
	`addi a1, a1, 0x4`		`lw t0, 0(a0)`	
	`beqz a2, end`			
`loop:`	`addi a0, a0, 0x4`	`addi a2, a2, -0x1`	`flw ft0, 0(a0)`	`fmul ft1,ft0,fa0`
	`addi a1, a1, 0x4`		`lw t0, 0(a0)`	
	`bnez a2, loop`		`fsw ft1, 0(t0)`	
				`fmul ft1,ft0,fa0`
			`fsw ft1, 0(t0)`	

Alternate version:

Label	ALU0	ALU1	MEM	FPU
	`addi a0, a0, 0x4`	`addi a2, a2, -0x1`	`flw ft0, 0(a0)`	
	`addi a1, a1, 0x4`		`lw t0, 0(a0)`	
	`beqz a2, end`			`fmul ft1,ft0,fa0`
`loop:`	`addi a0, a0, 0x4`	`addi a2, a2, -0x1`	`flw ft0, 0(a0)`	
	`addi a1, a1, 0x4`		`lw t0, 0(a0)`	
	`bnez a2, loop`		`fsw ft1, 0(t0)`	`fmul ft1,ft0,fa0`
			`fsw ft1, 0(t0)`	

3.B (4 points) Loop Unrolling

Could this code achieve higher throughput by combining loop unrolling with software pipelining? If so, briefly explain the approach of applying loop unrolling to the software-pipelined code. If not, explain your reasoning.

With the correct solution to 3.A, the answer is no. Since the MEM unit is fully utilized with only software pipelining, no additional utilization can be gained by further unrolling the code.

Alternatively, if the solution to 3.A did not fully utilize the MEM unit (the software pipelining was sub-optimal), then the answer is yes. Unrolling would allow the code to more densely pack instructions in a single inner loop iteration until either the MEM or the FPU unit is fully utilized.

Problem 4: Vectors (18 points CS152, 28 points CS252A)

4.A (12 points) Vectorization

Vectorize the following C function. Write your code in the table. You may not need to use all the spaces. Refer to Appendix A for an abbreviated RISC-V vector instruction listing.

- Fill in the blanks for the `vsetvli` instruction.
- The vector unit is configured with SEW=32 for 32-bit elements.
- Note that LMUL=1, so you may use any of the 32 standard vector registers. You may freely use any temporary registers.
- Assume there is no aliasing between the `a`, `b`, and `idx` arrays.

```
// Argument registers:              a0           a1           a2            a3
void leaky_activation(unsigned int n, int32_t *a, int32_t *b, uint32_t *idx,
//                             a4                a5
                      int32_t scale, uint32_t dilation) {
    for (unsigned int i = 0; i < n; i++) {
        if (a[idx[i]] < 0) {
            b[dilation*i] = scale * a[idx[i]];
        }
    }
}
```

Label	**Instruction**	**Comment** (optional)
	`// Prologue (if any)`	
	`slli a5, a5, 2`	set stride in bytes
loop:	`vsetvli t0, a0, e32, m1, ta, mu`	
	`// Do vector operations`	
	`vle32.v v1, (a3)`	load idx[i]
	`vsll.vi v1, v1, 2`	convert to byte offsets
	`vluxei32.v v2, (a1), v1`	load a[idx[i]]
	`vmslt.vx v0, v2, x0`	set mask (a[idx[i]] < 0)
	`vmul.vx v2, v2, a4, v0.t`	scale * a[idx[i]]
	`vsse32.v v2, (a2), a5, v0.t`	store b[dilation*i]
	`// Update pointers`	
	`sub a0, a0, t0`	decrement n
	`mul t1, t0, a5`	compute vl*dilation in bytes
	`slli t0, t0, 2`	convert vl to bytes
	`add a3, a3, t0`	bump pointer to idx
	`add a2, a2, t1`	bump pointer to b
	`// Branch to loop`	
	`bnez a0, loop`	
end:	`ret`	

Alternate version:

```
// Argument registers:              a0           a1          a2            a3
void leaky_activation(unsigned int n, int32_t *a, int32_t *b, uint32_t *idx,
//                            a4              a5
                      int32_t scale, uint32_t dilation) {
    for (unsigned int i = 0; i < n; i++) {
        if (a[dilation*i] < 0) {
            b[idx[i]] = scale * a[dilation*i];
        }
    }
}
```

Label	Instruction	Comment (optional)
	`// Prologue (if any)`	
	`slli a5, a5, 2`	`set stride in bytes`
`loop:`	`vsetvli  t0, a0, e32, m1, ta, mu`	
	`// Do vector operations`	
	`vlse32.v v1, (a1), a5`	`load a[dilation*i]`
	`vmslt.vx v0, v1, x0`	`set mask (a[dilation*i] < 0)`
	`vmul.vx v1, v1, a4, v0.t`	`scale * a[dilation*i]`
	`vle32.v v2, (a3), v0.t`	`load idx[i]`
	`vsll.vi v2, v2, 2, v0.t`	`convert to byte offsets`
	`vsoxei32.v v1, (a2), v2, v0.t`	`store b[idx[i]]`
	`// Update pointers`	
	`sub a0, a0, t0`	`decrement n`
	`mul t1, t0, a5`	`compute vl*dilation in bytes`
	`slli t0, t0, 2`	`convert vl to bytes`
	`add a3, a3, t0`	`bump pointer to idx`
	`add a1, a1, t1`	`bump pointer to a`
	`// Branch to loop`	
	`bnez a0, loop`	
`end:`	`ret`	

To preserve the same exception behavior as the original scalar code, all memory operations within the conditional block should be predicated to ensure that inactive elements do not fault.

4.B (6 points) Iron Law

For each of the following modifications to a baseline vector machine, indicate whether it has an **increasing**, **decreasing**, **negligible**, or **ambiguous** effect on CPI and on overall runtime (time per program) for a trivially vectorizable program, such as vector-vector add.

Briefly explain your reasoning in one or two sentences.

	Cycles / Instruction	**Time / Program**
Increase maximum hardware vector length (MAXVL)	**Increase** – Each vector instruction operates on more elements and therefore takes more cycles to execute, assuming that the application vector length of the program is greater than the original hardware vector length.	**Decrease** – As each vector instruction performs more work, fewer dynamic instructions and stripmine iterations are executed, reducing loop overhead and instruction cache pressure. Longer vectors mitigate dead time. If MAXVL is increased by repartitioning the vector register file (i.e., LMUL), the hardware implementation does not change and cycle time is unaffected.
Increase number of lanes	**Decrease** – More elements can be processed in parallel each cycle, reducing the number of cycles each vector instruction takes to execute.	**Decrease** – CPI decreases while the number of instructions executed remains constant. Since the lanes are largely independent structures, the critical path is normally impacted to a lesser degree.

4.C (5 points) Precise Exceptions (CS252A)

Describe the difficulties of implementing precise exceptions in a vector machine that allows multiple vector instructions to overlap in execution (e.g., chaining). Propose a solution that avoids significantly impacting performance.

The danger is that the first elements of a younger vector instruction may have already written back or stored to memory when one of the last elements of an older vector instruction causes an exception, which creates an imprecise architectural state.

Executing vector instructions strictly sequentially would avoid this but may reduce performance unacceptably. One solution is to only allow subsequent vector instructions to issue once all preceding instructions are known not to fault. For memory-related exceptions, the addresses for unit-stride loads and stores can be checked quickly in batches with a few TLB lookups; however, the address checks can be prolonged for indexed memory operations, as they often must be performed individually. Another solution is to rename the vector registers and buffer uncommitted stores in a store queue, so architectural state can be rolled back after an exception.

4.D (5 points) Decoupling (CS252A)

A decoupled vector machine splits the vector unit into two components connected by queues: a *vector execution unit* (VXU) that performs data computation and a *vector memory unit* (VMU) that handles address generation and memory accesses.

After being decoded, vector arithmetic instructions are sent to the VXU instruction queue and are issued in program order to the functional units when their operands are available.

Vector memory instructions are split into two parallel operations. For vector loads, the first operation is sent to the VMU instruction queue and generates a stream of memory requests that fill an internal vector load buffer. The second operation is sent to the VXU instruction queue and moves data from the vector load buffer to the vector register file. For vector stores, the first operation generates a stream of memory addresses to a vector store buffer where they wait for vector store data. The second operation is sent to the VXU instruction queue and reads out vector register data to the vector store data queue. The vector store buffer sends store requests to memory once both address and data are present for an entry.

What are the advantages of vector decoupling when executing a stripmine loop over many iterations?

As a form of access/execute decoupling, this scheme enables some degree of dynamic scheduling to help tolerate memory latency. Vector arithmetic instructions stalled on data dependencies do not block the issue stage. This frees the processor to run ahead and fetch vector memory instructions in upcoming stripmine iterations, allowing the vector unit to generate addresses and initiate memory requests earlier before older vector arithmetic instructions have begun execution.