CS 152 Computer Architecture and Engineering
CS 252 Graduate Computer Architecture

Midterm #1
March 2, 2020
Professor Krste Asanović

Name:_____
SID:_____
I am taking CS152 / CS252
(circle one)

This is a closed book, closed notes exam.
80 Minutes, 21 pages.

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	CS252 Point Value
1	15	15
2	20	--
3	20	20
4	25	25
5	--	20
TOTAL	80	80

Name: _____

Problem 1: (15 Points) Iron Law of Processor Performance

Mark whether the following modifications will cause each of the *first three* categories to **increase** or **decrease**, or whether the modification will have a **negligible** effect. Assume all other parameters of the system are unchanged whenever possible. Explain your reasoning.

For the rightmost column, mark whether the modification will cause *execution time* to **increase** or **decrease**, or whether the modification will have a **negligible** effect or a potentially significant but **ambiguous** effect. Explain your reasoning. If the modification has an **ambiguous** effect, describe the trade-off in which it might be significantly beneficial or in which it might be significantly detrimental (i.e., as an architect, when would you suggest implementing the modification or not and why?).

Be explicit if you are relying on any specific assumptions.

		Instructions / Program	Cycles / Instruction	Seconds / Cycle	Execution Time
a)	Using wider microcode in a microcoded machine	Negligible Microcode is not visible at the ISA level.	Decrease Fewer microinstructions are needed to implement an ISA instruction since each microinstruction can perform multiple parallel operations.	Increase Due to the sparse encoding, the larger microcode ROM could be slower to access. The datapath might be more complex if more control signals are required to drive it.	Decrease The parallelism from using wider microcode generally outweighs the cycle time impact.
b)	Pipelining the microcode engine in a microcode machine	Negligible Pipelining is not visible at the ISA level.	Decrease Replacing a multi-cycle bus-based implementation with a pipeline enables a CPI=1 to be potentially achieved.	Negligible A single-bus implementation may already have a fast cycle time due to its simplicity, but a sufficiently deep pipeline should have a comparable cycle time.	Decrease The instruction throughput is significantly improved.

Name: _____

c)	Adding an instruction to copy strings	Decrease Several instructions can be replaced with one complex instruction.	Increase Each string copy instruction takes potentially many cycles for multiple memory accesses.	Negligible Although extra control logic is required to sequence the string copy, the state machine is generally simple enough that it should not impact the cycle time for a typical pipeline in which the critical path goes through memory.	Decrease The loop overhead (pointer increment, bounds check, branch penalty) of a software string copy is eliminated, as well as the extra byte-oriented copies to handle non-word-aligned strings. (Same benefits as a DMA engine)
d)	Adding an L2 cache between the L1 cache and DRAM	Negligible An L2 cache is not visible at the ISA level.	Decrease An L2 cache should improve the average memory access time for L1 misses.	Negligible A larger L2 cache would be clocked at ½ the core frequency or slower to avoid impacting cycle time. **OR** Increase Larger SRAM banks incur a longer clock-to-q delay.	Decrease Reducing the latency for L1 misses should provide a benefit to most programs that exhibit locality.
e)	Adding virtual memory	Increase Page faults cause additional instructions to be executed in the OS page fault handler. Software TLB refills involve extra instructions on a TLB miss.	Increase Instruction fetches and memory operations incur extra cycles for page table walks, but this overhead is generally mitigated by a TLB.	Negligible The TLB can be accessed in parallel with a VIPT cache. The privileged architecture state, control logic, and hardware page table walker should not impact the critical path.	Increase A performance impact may be observed for applications with a working size greater than the TLB reach and when not all pages are resident in physical memory.

Name: _____

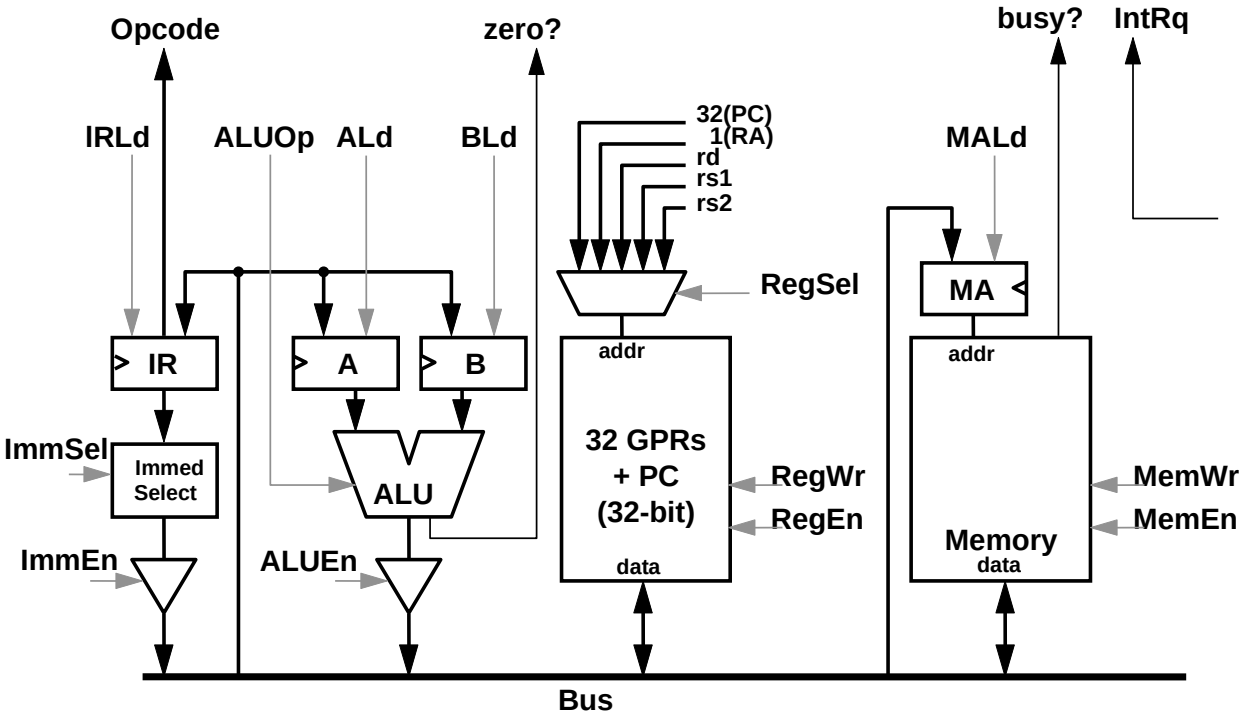
Problem 2: (20 Points) Microprogramming *(CS152 ONLY)*

In this problem, you will write microcode for a bus-based implementation of a RISC-V machine. This microarchitecture is identical to the one described in Handout #1 and Problem Set 1.

The final solution should be efficient with respect to the number of microinstructions used. Make sure to use logical descriptions of data movement in the "pseudocode" column for clarity. Credit will be awarded for optimizing signals using "don't care" or $*$ values as appropriate, but this is less important than producing a correct implementation.

Please comment your code clearly. If the pseudocode for a line does not fit in the space provided, or if you have additional comments, you may write neatly in the margins.

For your reference, the single-bus datapath is reproduced here, as well as some important information about microprogramming in the bus-based architecture.



Name: _____

Arithmetic Logic Unit:

ALUOp	ALU Result Output
COPY_A	A
COPY_B	B
INC_A_1	A+1
DEC_A_1	A-1
INC_A_4	A+4
DEC_A_4	A-4
ADD	A+B
SUB	A-B
SLT	Signed(A) < Signed(B)
SLTU	A < B

Table Q2-1: Available ALU operations

Immediate Selector:

Five immediate types are supported by **ImmSel**: ImmI, ImmU, ImmS, ImmJ, and ImmB.

Microbranches:

The **μBr** column represents a 3-bit field with six possible values: N, J, EZ, NZ, D, and S.

- N (next): The next state is simply (*current state* + 1).
- J (jump): The next state is *unconditionally* the state specified in the Next State column (i.e., it's an unconditional microbranch).
- EZ (branch-if-equal-zero): The next state depends on the value of the ALU's *zero* output signal (i.e., a conditional microbranch). If *zero* is asserted ($zero = 1$), then the next state is that specified in the Next State column, otherwise, it is (*current state* + 1).
- NZ (branch-if-not-zero): This behaves exactly like EZ but instead performs a microbranch if *zero* is not asserted ($zero \neq 0$).
- D (dispatch): The FSM looks at the opcode and function fields in the IR and goes to the corresponding state.
- S (spin): The μPC stalls if *busy?* is asserted; otherwise, it goes to (*current state* +1).

Guidelines for Enable Signals:

- Only one source of data can drive the bus in any cycle.
- Don't worry about marking any of the en__ signals as don't care. However, other types of signals should be marked as don't care where applicable.
- Two control signals determine how the register file is used during a cycle: RegWr and enReg. RegWr determines whether the operation to be performed, if any, is a read or a write. If RegWr=1, then it is a write; otherwise it is a read. enReg is a general enable control for the register file. If enReg=1, then the register reads or writes depending on RegWr. If enReg=0, then nothing is done, regardless of the value of RegWr.
- MemWr and enMem function in an analogous way for the memory.

Name: _____

2.A (16 points) Implement a `SWITCH` instruction

The `SWITCH` instruction performs a multiway indirect branch, corresponding to the C code:

```
switch (index) {
      case 0: goto target_0;
      case 1: goto target_1;
      case 2: goto target_2;
      …
      case limit: goto target_last;
}
// Fall through if index is out of bounds
```

The `SWITCH` instruction has the following format:

```
SWITCH rs1, rs2, imm
```

The operands consist of two source registers and one **B-type** immediate:

`rs1`: Zero-based index to select a branch table entry

`rs2`: Pointer to a branch table in memory

`imm`: Limit, the index of the last table entry ($N - 1$)

The *table* operand (`rs2`) points to an array in memory with N word-sized entries, each holding a branch target address:

Address	Content
`table + 0`	`target_0`
`table + 4`	`target_1`
`table + 8`	`target_2`
…	…
`table + (4×limit)`	`target_last`

The *index* (`rs1`) is compared with *limit* (`imm`) to check that it is within the table range. If *index* ≤ *limit*, then the processor branches to the address stored in the *table[index]* entry. Otherwise, if *index* > *limit*, no branch is taken, and execution continues at PC + 4 as usual.

For simplicity, assume that the immediate representing *limit* must be ≥ 0.

Note: The ALU does not support a multiply or shift operation, but multiplication by a power of 2 (i.e., left shift) can be efficiently handled with repeated doubling.

Fill in the microcode table on the following page.

Name: _____

2.B (4 Points) Performance of your `SWITCH` implementation

How many cycles does your `SWITCH` instruction take to execute in the following situations? Assume that all memory accesses complete in a single cycle (just for the purposes of this CPI calculation – you must still use spin states). Count all cycles starting from `FETCH0` to the last microinstruction that jumps back to `FETCH0`.

1. *index* ≤ *limit*

   12 cycles

   - Fetch/dispatch: 3 cycles
   - SWITCH routine: 9 cycles

2. *index* > *limit*

   6 cycles

   - Fetch/dispatch: 3 cycles
   - SWITCH routine: 3 cycles

Name: _____

State	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En	A Ld	B Ld	ALUOp	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	μBr	Next State
FETCH0:	MA ← PC; A ← PC	*	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR ← Mem	1	*	*	0	0	*	*	0	0	0	1	*	0	S	*
	PC ← A+4	0	PC	1	1	0	*	INC_A_4	1	*	*	0	*	0	D	*
…																
SWITCH0:	A ← Imm	0	*	*	0	1	*	*	0	*	*	0	B	1	N	
	B ← R[rs1]	0	rs1	0	1	0	1	*	0	*	*	0	*	0	N	
	If (A < B) { μBr to FETCH0 } A ← R[rs1]	0	rs1	0	1	1	0	SLTU	0	*	*	0	*	0	NZ	FETCH0
	A, B ← A + B	0	*	*	0	1	1	ADD	1	*	*	0	*	0	N	
	A ← A + B	0	*	*	0	1	*	ADD	1	*	*	0	*	0	N	
	B ← R[rs2]	*	rs2	0	1	0	1	*	0	*	*	0	*	0	N	
	MA ← A + B	*	*	*	0	*	*	ADD	1	1	*	0	*	0	N	
	PC ← Mem	*	PC	1	1	*	*	*	0	0	0	1	*	0	S	
	μBr to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0

Name: _____

Problem 3 (20 Points): Pipelining and Exceptions

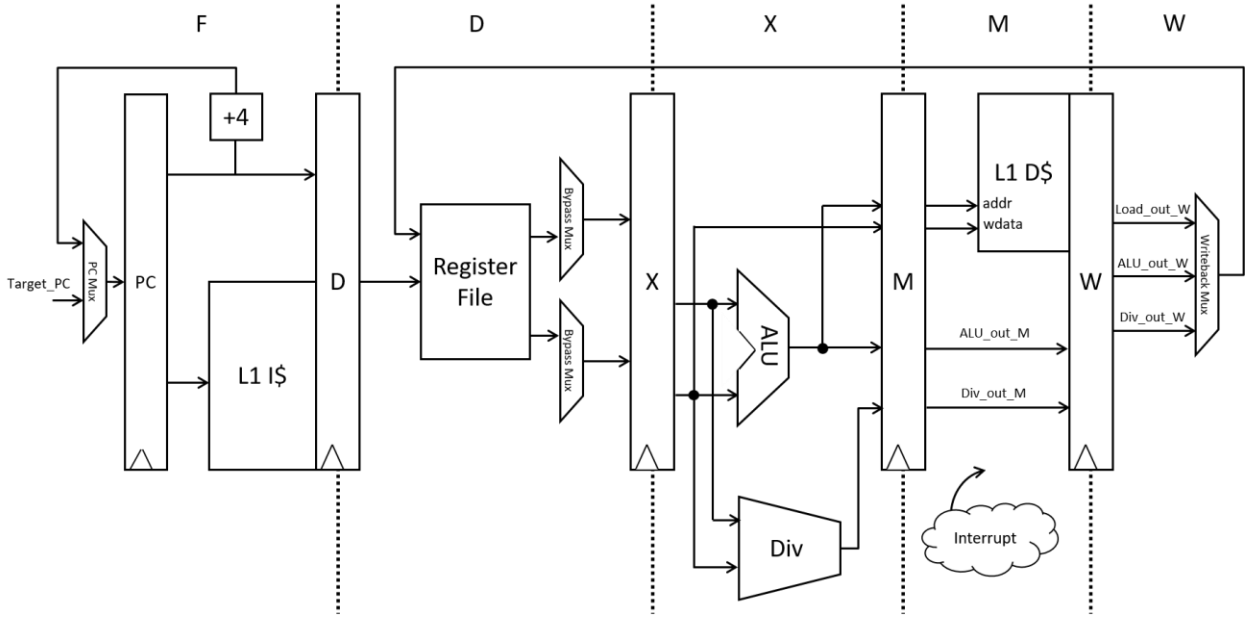


Figure 3.1

3.A (2 Points) Latency vs Occupancy

Figure 3.1 shows a classic fully-bypassed 5-stage pipeline that has been augmented with an unpipelined divider in parallel with the ALU. Bypass paths are not shown in the diagram. This iterative divider produces 2 bits per cycle until it outputs a full 32-bit result.

1. (**1 Point**) What is the latency of a divide operation in cycles?

   32 / 2 = 16 cycles

2. (**1 Point**) What is the occupancy of a divide operation in cycles?

   16 cycles, since the divider is unpipelined

Name: _____

3.B (3 Points) Hazards

Note that the `div` instruction in RISC-V cannot raise a data-dependent exception. To avoid pipeline stalls while a multi-cycle divide operation is in progress, the pipeline control logic allows subsequent instructions that do not depend on the divide result to be issued and completed before the divide has completed.

What additional hazards might be caused by `div` instructions, aside from the structural hazard on the divider itself? If any, describe how they could be resolved using an interlock.

WAW hazards can arise from the out-of-order completion of `div` instructions. For any subsequent instruction that writes to the same destination register as an ongoing `div` instruction, one approach is to stall at the D stage (or alternatively, in X or M at the potential cost of higher-fanout stall signals). This can be implemented using a comparator that checks the destination register of the `div` against the destination of the instruction in decode, or with a scoreboard that contains a bit-vector to track pending writes.

A potential structural hazard also exists on the write port of the register. When a `div` and another instruction that writes a register are both in the W stage, one of them must stall.

Although not a true control hazard, when a branch following a `div` that has not yet completed is taken, the ongoing divide operation should not be killed in the partial pipeline flush.

3.C (10 Points) Interrupts

In this pipeline, asynchronous interrupts are handled in the MEM stage and cause a jump to a dedicated interrupt trap handler address. The *interrupt latency* is defined as the number of cycles from when an interrupt request is raised in the MEM stage until the first instruction of the interrupt handler reaches the MEM stage.

1. (**1 Point**) What is the minimum interrupt latency that the pipeline can achieve in the best-case scenario?

   4 cycles, shown by the shaded cycles in this pipeline diagram. The interrupt is raised in cycle 4. The earliest uncommitted instruction when the interrupt is taken is labeled with "EPC", while "MTVEC" denotes the first instruction of the interrupt trap handler.

	1	2	3	**4**	5	6	7	8	
EPC	F	D	X	M					
…		F	D	-					
…			F	-					
…				-					
MTVEC					F	D	X	M	W

Name: _____

2. **(6 Points)** Consider the execution of the code below. Suppose an interrupt is raised during cycle 8, which causes a jump to `interrupt_handler`. The handler increments a counter at a fixed memory address before returning to the original context.

   Fill in the pipeline-timing diagram on the next page until the `mret` instruction at the end of `interrupt_handler` commits. The architectural guarantee of precise interrupts should be upheld. Assume that all memory accesses take one cycle in the MEM stage.

```
        lw   x2, 0(x1)
        div  x1, x2, x3
        slli x3, x2, 1
        lui  x4, 0x100
        addi x4, x4, 0xf
        xor  x5, x3, x4
        sub  x3, x5, x2

        ...
interrupt_handler:
        sw   x1, 0(x0)   # Save register in known location
        lw   x1, 4(x0)   # Use register to increment counter
        addi x1, x1, 1
        sw   x1, 4(x0)
        lw   x1, 0(x0)   # Restore register before returning
        mret             # Return from interrupt handler
```

3. **(2 Points)** What is the interrupt latency for the code above?

   14 cycles (cycles 8 to 21, inclusive)

4. **(1 Point)** Which instruction should `interrupt_handler` return to in order to ensure that the program will continue to execute correctly?

   The EPC (exception PC) should point to the earliest uncommitted instruction:
   `lui x4, 0x100`

Name: _____

	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
lw	F	D	X	M	W																									
div		F	D	D	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	M	W								
slli			F	F	D	X	M	W																						
lui					F	D	X	M																						
addi						F	D	X																						
xor							F	M																						
sub								F																						
sw									F	D	D	D	D	D	D	D	D	D	D	D	X	M	W							
lw										F	F	F	F	F	F	F	F	F	F	F	D	X	M	W						
addi																					F	D	D	X	M	W				
sw																						F	F	D	X	M	W			
lw																								F	D	X	M	W		
mret																									F	D	X	M	W	

Name: _____

3.D (5 Points) Reducing Interrupt Latency

Propose a modification to the architecture and/or microarchitecture that would reduce the interrupt latency for the code in 3.C, while *ensuring that interrupts are handled precisely*.

Further unrolling the iterative divider (e.g., to output 4 bits per cycle instead of 2) would shorten its latency and alleviate the RAW hazard, but this is not always feasible in terms of cycle time.

We could also prevent subsequent instructions from committing before the `div` commits, which allows the divide to be killed precisely on a taken interrupt, although this option is unattractive for the common case where interrupts are relatively infrequent.

Yet another approach is to constrain the destination register for a `div` instruction. The ABI could be changed to reserve a specific `x` register to always hold the divide result, so the interrupt handler can avoid using it. In a more generalized exception handler, reading the register can be deferred towards the end of the context save, which might hide the divide latency. This does not require any hardware modifications but relies on code voluntarily complying with the ABI.

Alternatively, the `div` instruction could be redefined to write to a special-purpose register separate from the `x` register file. This is the approach adopted by MIPS with its `hi` and `lo` registers, but it involves an ISA change with RISC-V.

A tempting proposition is to kill the ongoing divide operation and save the intermediate pipeline state so that it can be restarted after the interrupt. Although interrupt latency is reduced, it creates an imprecise interrupt, as the instruction following `div` has already committed.

Name: _____

Problem 4: (25 Points) Caches

In this problem, we will investigate how various cache organizations perform on the following loop. Let `A` be a 1024×1024 matrix of 32-bit `int` elements stored in row-major order, aligned to the beginning of a cache line.

```
for (int i = 1; i < 16; i++) {
     int x = A[0][i-1];
     int y = A[i][i];
     A[i][i] = x + y;
}
```

Assume that memory accesses are executed in the order shown in the program – i.e., the compiler does not reorder load and store instructions. Variables `x`, `y`, and `i` are held in registers.

4.A (6 Points) Direct-Mapped Cache

Consider a 4 KiB direct-mapped L1 data cache with 16-byte cache lines.

1. (**4 Points**) Count the numbers of cache hits and misses for each category on the loop shown above. Assume that the cache is initially empty.

   Note that elements `A[0][i]` and `A[i][i]` map to the same set in the cache, which means that the accesses to `A[i][i]` in iteration *i* will evict the line that `A[0][i-1]` would reference in iteration *i+1*. The first four iterations thus behave as follows:

	i = 1	*i = 2*	*i = 3*	*i =4*
`A[0][i-1]` (load)	Compulsory miss	Conflict miss	Conflict miss	Conflict miss
`A[i][i]` (load)	Compulsory miss	Compulsory miss	Compulsory miss	Compulsory miss
`A[i][i]` (store)	Hit	Hit	Hit	Hit

   The pattern repeats for *i = 5* through *i = 15* (i.e., *i = 5* will hit/miss for each access in the same way as *i = 1*).

   - Hits: 15 (`A[i][i]` store)
   - Compulsory misses: 4 (`A[0][i-1]` load) + 15 (`A[i][i]` load) = 19
   - Conflict misses: 11 (`A[0][i-1]` load)
   - Capacity misses: 0

Name: _____

(**2 Points**) What is the average memory access time (AMAT) in cycles if the hit time of the direct-mapped cache is 1 cycle and the L1 miss penalty to DRAM is 100 cycles? (You do not need to calculate the exact number; just write the formula with the individual terms substituted with the appropriate values.)

The miss rate is 2/3 (30/45).

$$AMAT = 1 + (2/3)(100)$$

4.B (6 Points) 2-way Set-Associative Cache

Now we double the capacity by switching to an 8 KiB two-way set-associative L1 data cache with LRU eviction and a write-allocate policy. The cache line size remains 16 bytes.

1. (**4 Points**) Count the numbers of cache hits and misses for each category on the preceding loop. Assume that the cache is initially empty.

   All conflict misses for `A[0][i-1]` are eliminated with 2-way associativity.

	i = 1	*i = 2*	*i = 3*	*i =4*
`A[0][i-1]` (load)	Compulsory miss	Hit	Hit	Hit
`A[i][i]` (load)	Compulsory miss	Compulsory miss	Compulsory miss	Compulsory miss
`A[i][i]` (store)	Hit	Hit	Hit	Hit

   As before, the pattern repeats for *i = 5* through *i = 15*.

   - Hits: 11 (`A[0][i-1]` load) + 15 (`A[i][i]` store) = 26
   - Compulsory misses: 4 (`A[0][i-1]` load) + 15 (`A[i][i]` load) = 19
   - Conflict misses: 0
   - Capacity misses: 0

2. (**2 Points**) What is the AMAT in cycles if the hit time is 2 cycles and the L1 miss penalty to DRAM is 100 cycles? (You do not need to calculate the exact number; just write the formula with the individual terms substituted with the appropriate values.)

   The miss rate is 19/45.

$$AMAT = 1 + (19/45)(100)$$

Name: _____

4.C (7 Points) 2-way Column-Associative Cache

Suppose we convert our 8 KiB 2-way set-associative cache from 4.B into an 8 KiB 2-way *column-associative cache* with 16-byte lines. A column-associative (or pseudo-associative) cache is similar in structure, except that instead of accessing both ways simultaneously, the ways are accessed sequentially over consecutive cycles.

In other words, each way is treated as a separate 4 KiB direct-mapped cache. On a cache access, Way 0 is searched first. If the line is not found in Way 0, then Way 1 is accessed the next cycle. If there is a hit in Way 1, the lines in the two ways are swapped. On a miss, the new line is placed in Way 0, and the previous line is moved to Way 1.

1. (**1 Point**) What is an advantage of a column-associative cache compared to a set-associative cache of the same associativity?

   A column-associative cache can have a lower hit time comparable to a direct-mapped cache, since the way muxing is eliminated and the tag check does not need to be parallel.

2. (**4 Points**) Count the numbers of cache hits and misses for each category on the preceding loop. Assume that the cache is initially empty.

   The miss on `A[i][i]` causes the line in which `A[0][i]` resides to be swapped to Way 1, which increases the hit time slightly for `A[0][i-1]` in the next iteration.

	i = 1	*i = 2*	*i = 3*	*i =4*
`A[0][i-1]` (load)	Compulsory miss	Hit (Way 1)	Hit (Way 1)	Hit (Way 1)
`A[i][i]` (load)	Compulsory miss	Compulsory miss	Compulsory miss	Compulsory miss
`A[i][i]` (store)	Hit (Way 0)	Hit (Way 0)	Hit (Way 0)	Hit (Way 0)

   As before, the pattern repeats for *i = 5* through *i = 15*.

   - Hits in Way 0: 15 (`A[i][i]` store)
   - Hits in Way 1: 11 (`A[0][i-1]` load)
   - Compulsory misses: 4 (`A[0][i-1]` load) + 15 (`A[i][i]` load) = 19
   - Conflict misses: 0
   - Capacity misses: 0

3. (**2 Points**) What is the AMAT in cycles if accessing each way takes 1 cycle and the L1 miss penalty to DRAM is 100 cycles? (You do not need to calculate the exact number; just write the formula with the individual terms substituted with the appropriate values.)

   The fraction of Way 1 hits is 11/45, and the miss rate to DRAM is 19/45.

   $$AMAT = 1 + (11/45)(1) + (19/45)(1 + 100)$$

4.D (6 Points) Virtual Memory

To further reduce hit time while maintaining capacity, we now consider moving back to an 8 KiB direct-mapped VIPT (virtually indexed, physically tagged) cache.

1. (**2 Points**) Explain how virtual memory aliasing can occur with 4 KiB pages.

   Since the most significant bit of the cache index is shared with the least significant bit of the VPN, it is possible for two virtual addresses which map to same physical address to be located in different sets in the cache.

2. (**4 Points**) Describe a mechanism to prevent aliases from co-existing in the 8 KiB direct-mapped VIPT cache.

   Aliases can exist only in two sets for a given physical address. First, we index the cache and perform the tag check as usual. If there is no match, we flip the MSB of the index (which overlaps with the VPN) and check the corresponding set on the next cycle. This prevents a new copy of a physical line from being allocated in the cache on a miss if it is already present through an alias.

Name: _____

Problem 5: (25 Points) Runahead Processing *(CS252 ONLY)*

An in-order *runahead processor* is one technique to reduce the impact of cache misses. A runahead processor has two execution modes (regular and runahead) and two corresponding copies of all architectural registers (regular and runahead). The runahead registers each have an additional valid bit indicating if the register contains valid data.

In regular execution mode, the processor behaves as a regular in-order processor and updates the regular architectural registers. But instead of stalling when the processor encounters a data cache miss on a load instruction, it switches to runahead mode. First the processor copies the regular architectural registers including the program counter into the runahead architectural registers, and sets all the runahead register valid bits, except on the register corresponding to the target of the load which is marked invalid. The processor then begins execution in runahead mode.

In runahead mode, the processor continues to execute instructions but now uses the runahead registers. If the result of an instruction depends on a source register marked invalid, its destination runahead register is also marked invalid. If a runahead instruction is a load that causes a new data cache miss, the destination runahead register is marked invalid, the data cache issues a prefetch for the missing line, and the processor continues execution.

When the original data cache miss returns, the missing load's destination register in the regular register set is updated, then the processor re-enters regular execution mode with the regular program counter pointing to the instruction after the load that caused the original data cache miss.

5.A (3 Points) Branches

What should runahead mode do when encountering a conditional branch that compares one or more invalid registers?

Name: _____

5.B (3 Points) Jumps

What should runahead mode do when encountering a jump register (`jr`) instruction where the target is an invalid register?

5.C (4 Points) Stores

How should runahead mode handle store instructions?

Name: _____

5.D (5 Points) Vector Accumulate

Consider the following loop, which accumulates elements in a vector, running on a runahead processor:

```
        li   x15, 0          # Clear accumulator
loop:
        lw   x8, (x10)       # Get next word
        addi x10, x10, 4     # Bump address pointer
        add  x15, x15, x8    # Accumulate new word into sum
        bne  x10, x11, loop # Loop if not at end of vector
```

This runahead processor has a regular 5-stage RISC pipeline, and the copy from architectural registers to runahead registers uses special data paths to complete in one cycle.

The system has 32-byte cache lines. Assume the loop accumulates over many elements. What is the smallest cache miss penalty for which the runahead processor will exhibit a performance improvement on this loop over a non-runahead processor?

Name: _____

5.E (5 Points) Linked List Accumulate

Consider the following loop, which accumulates values in a linked list, running on a runahead processor:

```
        li   x15, 0        # Clear accumulator
        beqz x10, exit     # Check if pointer is null
loop:
        lw   x8,  0(x10)   # Get next value
        lw   x10, 4(x10)   # Get next pointer
        add  x15, x15, x8  # Accumulate value into sum
        bnez x10, loop     # Loop if next pointer is not null
exit:
```

Describe if and how the runahead processor can provide a benefit in this case over a simple in-order processor that stalls on a load cache miss.