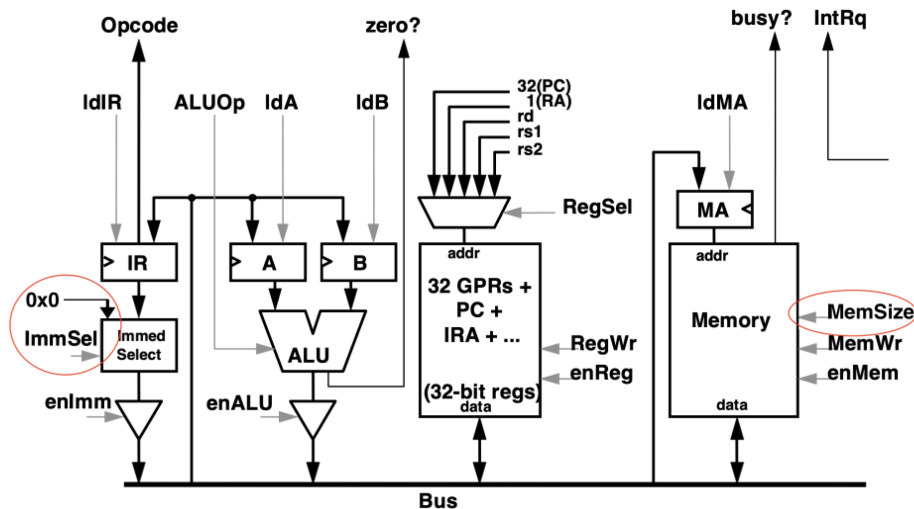
CS152 Spring 2023 Final Review

MICROCODING

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

In this problem, we explore microprogramming by writing microcode for a bus-based implementation of the RISC-V machine. This microarchitecture is largely the same as the one described in Handout #1, Problem Set 1, and Lab 2, with a few key differences. For clarity, we have reproduced the full microarchitectural diagram with new control signals in boldface.



New control signals

- `ImmSel` may take the value `zero`; this puts a zero on the bus when `enImm` is high
- Memory now receives an additional `MemSize` control signal, which takes the value 0, 1, or 2 to mean a 8-, 16-, or 32-bit load or store. Assume that load values are zero-extended, and that the upper bits are ignored when performing stores of less than 32 bits.
- **Memory may take multiple cycles to return—make sure to use spin states!**

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 realizing that signals may take a "don't care" or `X` value, but this is less important than producing a correct implementation!

2.A (15 points) Implement a `strchridx` instruction

Given a string of **single-byte** characters, find the first occurrence of a specified character. Return the index of the first occurrence or -1 if the character does not appear in the string. If bits `[31:8]` of `rs2` are not all zero, the behavior of the instruction is undefined. When the instruction commits, `rs1` and `rs2` (and all other architectural registers other than `rd`) must have their original values!

```
strchridx rd, rs1, rs2
```

Arguments:	`rs1`	A pointer to the null-terminated string `s`
	`rs2`	The character `c` to search for
Result:	`rd`	The index of the first appearance of `c` in `s`, or -1 if it doesn't appear

For simplicity, you may assume that `rd != rs1`.

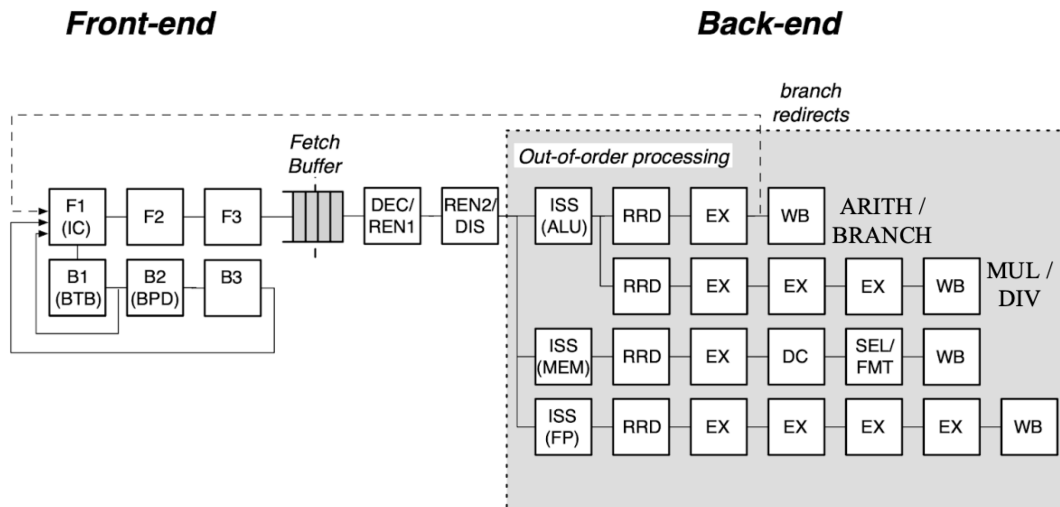
Fill in the microcode table on the next page.

State	PseudoCode	ldIR	Reg Sel	Reg Wr	en Reg	ldA	ldB	ALUOp	en ALU	ld MA	Mem Wr	en Mem	Mem Size	Imm Sel	en Imm	uBr	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; dispatch	0	PC	1	1	*	*	INC_A_4	1	*	*	0	*	*	0	D	*
...																	
NOP0	uBr to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	*	0	J	FETCH0
STRCHRIDX0																	

OOO

Problem 3: Unified Physical Register File Out-of-Order Machines

Throughout this question, assume the following machine specifications:



- The machine can fetch, dispatch, issue, and commit at most **one** instruction per cycle.
- The processor runs the RISC-V instruction set with the F and D extensions.
- Assume every load hits in the single-cycle-hit L1 D$ (indicated as DC in the pipeline).
- Register renaming follows the **Unified Physical Register File** scheme.
- Unless otherwise directed, **assume there are no bypass paths for data.**
- Instructions are written into the ROB at the end of the DEC/REN1 stage.
- Instructions are written into the issue window at the end of the REN2/DIS stage.
- Instructions are released from the issue window in the ISS stage.
- Commit is handled by a decoupled unit that looks at the ROB entries.
- Jump instructions issue and complete immediately on the same cycle that they dispatch.
- Assume all jump targets are perfectly predicted.
- Instructions may issue as soon as the same cycle that the writer of their last outstanding operand is in the writeback stage.
- **Ignore structural hazards on the register file ports**
- **Each functional unit has its own issue window, separate from the ROB**

F) (14 points) Consider the following code sequence that begins at address 0x00010000

```
loop:   fld    f0, 0(a0)
        fld    f1, 0(a1)
        fadd.d f0, f0, f1
        fsd    f0, 0(a2)
        addi   a0, a0, 0x8
        addi   a1, a1, 0x8
        addi   a2, a2, 0x8
        addi   t0, t0, 0x1
        bne    t0, a3, loop
```

Assume that the machine enters this loop with all instructions fetched, zero valid entries in the ROB, and the following initial rename table and free list contents before the first `fld` enters the ROB. Dequeue free list entries from the top.

Unused architectural registers are omitted from the rename table for clarity.

Arch. register	Phys. register
a0	p1
a1	p5
a2	p33
t3	p17
t0	p41
f0	p62
f1	p28

Free List
p4
p55
p18
p30
p39
p11
p59
p60

Now consider the case in which the `fsd` takes an exception. Fill in the following table which describes the execution of each instruction) for eight instructions, beginning with the first fld. In the "Time" columns, fill in the cycles in which the instruction dispatches, issues, completes, and commits (if it commits), respectively.

PC	Physical Register Specifiers				Cycle #			
	PRd	LPRd	PR1	PR2	Dispatch	Issue	Complete	Commit
0x00010000								
0x00010004								
0x00010008								
0x0001000C								
0x00010010								
0x00010014								
0x00010018								
0x0001001C								

MULTITHREADING

(a) **Thread Scheduling** [6 pts]

Consider the loop below, performing a saturating addition of two vectors; `Y[i]`, `X[i]`, and `MAX` are all of type `int32`.

```
for (int i = 0; i < N; i++) {
    Y[i] += X[i];

    if (Y[i] > MAX)
        Y[i] = MAX;
    }
}
```

In assembly, the loop is as follow:

```
# x1 is a pointer to the beginning of "X"
# x2 is a pointer to the beginning of "Y"
# x3 is "MAX"
# x4 is "i"

    loop: ld x5, 0(x1) # x5 has "X[i]"
          ld x6, 0(x2) # x6 has "Y[i]"

          addi x1, x1, 4 # increment X pointer
          addi x2, x2, 4 # increment Y pointer
          addi x4, x4, -1 # decrement "i"

          add x7, x5, x6

          blt x7, x3, skip_sat
          mv x7, x3

skip_sat: sw x7, 0(x2)

          bnez x4, loop
     end:
```

Suppose that we run this loop on a multi-threaded 5-stage classic RISC processor where:

- Loads and stores have a latency of 20 cycles
- Taken branches have a latency of two cycles
- All other instructions (including non-taken branches) have a latency of one cycle
- There is perfect branch prediction

Do not reorder the instructions in the loop.

i. How many threads are needed to guarantee that we will never stall with fixed round-robin scheduling?

ii. Suppose that we instead use a coarse-grained thread scheduling policy, which switches threads whenever a stall would occur due to a RAW hazard or due to a taken branch. (WAR and WAW hazards do not cause stalls).
How many threads are needed to guarantee that would never stall with this scheduling policy? Consider only the steady-state execution.

(b) **Short Questions** [2 pts]

Suppose that we have two machines, A and B. They are both used to run a wide variety of multithreaded workloads.

A has one CPU. The CPU is an OoO 8-wide superscalar with SMT. Two threads can run simultaneously on the CPU.

B has two CPUs. Each CPU is an OoO 4-wide superscalar without multithreading.

Check off the correct answers below:

i. A would be expected to have:
   - ☐ the same clock cycle time as B?
   - ☐ a lower clock cycle time than B?
   - ☐ a higher clock cycle time than B?

ii. A would be expected to have:
   - ☐ the same CPI as B?
   - ☐ a lower CPI than B?
   - ☐ a higher CPI than B?

VECTORS

(b) **Short Questions** [4 pts]

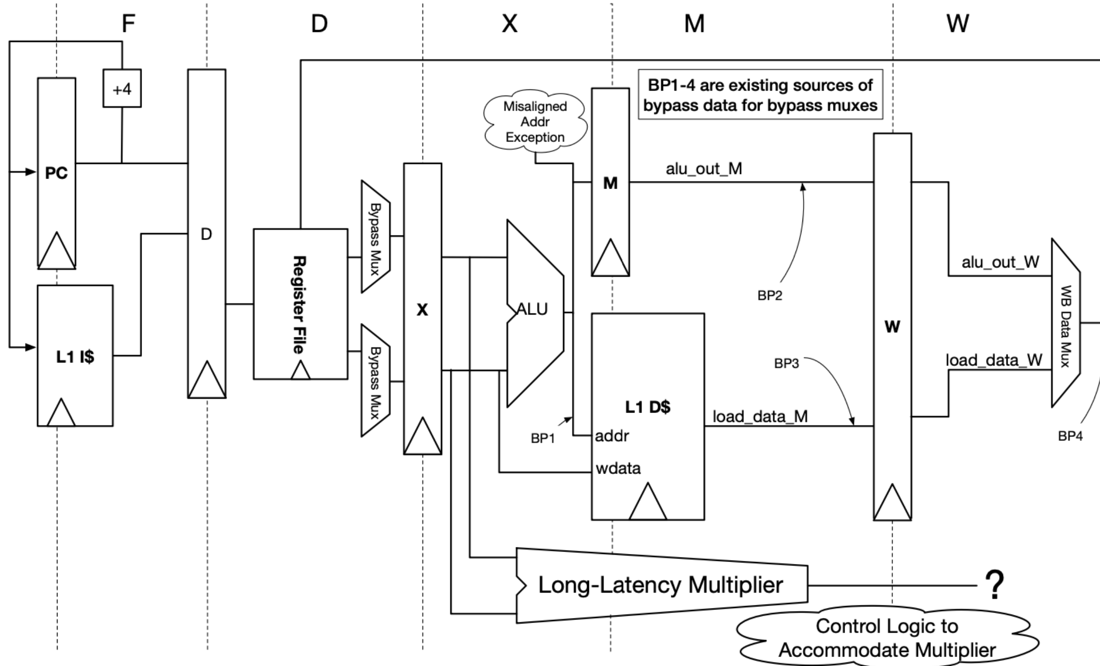
i. Suppose we are running a program with extremely long vector lengths. Why might we want to have fewer vector lanes than the vector register length? Assume that cycle time is not affected by the number of lanes, and that hardware complexity costs are not a concern.

ii. Why are precise exceptions difficult to support with vector processors?

PIPELINES+IRON LAW

	Instructions / Program	Cycles / Instruction	Seconds / Cycle	Execution Time
Pipelining a single-cycle implementation				
Pipelining an unpipelined multi-cycle implementation, while keeping the *latency* of each instruction constant				
Reducing the number of stages in a pipeline				

Modifying the 5-stage Pipeline: Long Latencies



A) **(2 Point)** If we allow non-dependent operations to proceed while a multiply is outstanding, which parts of execution are proceeding "out-of-order" in the processor? Assume that multiply operations cannot generate exceptions based on their input or output values. **Check all that apply.**

[ ] Issue [X] Completion [ ] Commit [ ] None of these

B) **(3 Points)** Suppose we want to add a multiplier with an 8-cycle latency and an 8-cycle occupancy to a 5-stage pipeline, as shown above. We must keep around some information about registers whose values are pending an outstanding multiply operation. What type of microarchitectural structure is commonly used for this purpose?

CACHES

A) (3 Points) How does total capacity, access latency, and bandwidth scale as we move between layers of a uniprocessor's memory hierarchy? Indicate the general trend each with an arrow pointing in the direction of an **increase** in that parameter.

Memory	Example Parameter	Capacity	Access Latency	Bandwidth
Register File	↑			
On-chip Caches				
Off-Chip Memory				

B) (3 Points) Suppose we build a processor with a 1-cycle L1 hit latency, a 10-cycle L1 miss penalty, and a 20-cycle L2 miss penalty. Assuming a L1 hit rate of 90% and a L2 local hit rate of 80%, what is the average memory access time seen by this processor?

C) (4 Points) What is the difference between an inclusive and exclusive multi-level cache? Give one advantage of each approach.

The remainder of this problem evaluates cache performance for different loop orderings. Consider the following two loops, written in C, which calculates the sum of all elements of a 16 by 512 matrix of 32-bit integers:

Loop A	***Loop B***

`int sum = 0;`
`for (i = 0; i < 16; i++)`
`  for (j = 0; j < 512; j++)`
`    sum  += A[i][j];`
	`int sum = 0;`
	`for (j = 0; j < 512; j++)`
	`  for (i = 0; i < 16; i++)`
	`    sum += A[i][j];`

The matrix `A` is stored contiguously in memory in row-major order. Row-major order means that elements in the same row of the matrix are adjacent in memory. You may assume A starts at 0x0, thus `A[i][j]` resides in memory location `[4*(512*i + j)]`.

Assume:

- caches are initially empty.
- only accesses to matrix `A` cause memory references and all other necessary variables are stored in registers.
- instructions are in a separate instruction cache.

D) (7 points) Consider a 4KiB direct-mapped data cache with 64-byte cache lines. Calculate the number of cache misses that will occur when running Loop A. Calculate the number of cache misses that will occur when running Loop B. You must show your work for full credit!

E) (7 points) Consider a 4KiB fully-associative data cache with 64-byte cache lines. This data cache uses a first-in/first-out (FIFO) replacement policy. Calculate the number of cache misses that will occur when running Loop A, and when running Loop B. You must show your work for full credit!

VIRTUAL MEMORY

Consider a system which uses a two-level page-based virtual memory system.

- Pages are 16 bytes
- PTEs are 4 bytes
- Memory is byte-addressed
- The system is initialized with only the base page table allocated
- Physical pages are allocated from lower to higher PPNs incrementally
- The base page table is architecturally mandated to be at physical address 0x00, so a PTE containing value 0x00 is effectively an "invalid" PTE (no valid bit is necessary)
- The PTE is entirely reserved for a PPN (no valid, status, or permission bits)

Fill out the contents of physical memory after value **0x6C** is written to virtual address **0x94**.
Fill out the contents of physical memory after value **0x94** is written to virtual address **0x6C**.
You only need to show the values of the memory locations that are written/changed.

Address	Value
0x00	
0x04	
0x08	
0x0c	
0x10	
0x14	
0x18	
0x1c	
0x20	
0x24	
0x28	
0x2c	
0x30	
0x34	
0x38	
0x3c	
0x40	
0x44	
0x48	
0x4c	

2.B (4 pt) Virtual Address Space

What is the size of the virtual address space of this virtual memory system in bytes?

2.C (4 pt) Physical Address Space

How much physical memory does this virtual memory system support?

2.D (4 pt) VIPT L1

Explain briefly why **L1** caches are often designed to be **VIPT** (virtually indexed, physically tagged).

2.E (4 pts) PIPT L2

Explain briefly why **L2** caches are often designed to be **PIPT** (physically indexed, physically tagged).

CACHE COHERENCE

5.A (6 pt) Out-of-order Coherence

Consider an out-of-order processor that implements conservative out-of-order load execution as discussed in lecture. A load is issued as soon as its address calculation is completed (potentially out of program order) and the following conditions are met:

- All addresses for older stores in the speculative store buffer are known.
- If the load address matches one of those entries in the speculative store buffer, the store data from the youngest store older than the load is available for bypassing.

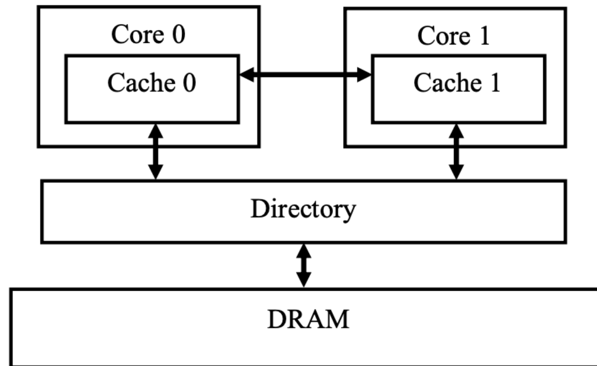
i. **(4 pt)** This approach behaves correctly in a single-core system. How can this approach cause a coherence violation in a multi-core system?

(2 pt) Propose a simple solution for the coherence problem discussed above.

5.A (16 pt) Directory-based MOSI Coherence

Consider the baseline directory-based cache-coherence protocol discussed in Handout 6 (distributed with exam), which implements an MSI protocol. We consider extending that protocol to support MOSI coherence in a system which implements cache-to-cache links.

In the diagram of the adjusted system below, notice that DRAM is distinct from the directory.



To support the MOSI protocol in the directory-based system, we make the following modifications:

- New cache state **C-owned** for the O state in MOSI
  - If a cache line is in this state, the line is **dirty** and **read-only**, and the owning cache is responsible for providing data to other caches.
  - The C-owned state can only be entered from the C-modified state.
  - A single cache may have the line in C-owned while multiple other caches have the same line in C-shared.
- New directory state **O(id, dir)**
  - Cache <id> is the owner of the line, and all caches in *dir* are sharers.
- New message type **FwdShReq(Home, id, id`, a)**
  - This is sent from the directory to cache <id> when the directory is in the W or O state and has received a ShReq from cache <id`>.
  - When cache <id> receives this message, it moves the line to the C-owned state and sends ShRep directly to cache <id`>.
  - Note that FwdShReq subsumes WbReq/WbRep in the original MSI protocol.
- New message type **FwdExReq(Home, id, id`, a)**
  - This is sent from the directory to cache <id> when the directory is in the W or O state and has received an ExReq from cache <id`>.
  - When cache <id> receives this message, it invalidates its copy of the line and sends ExRep directly to cache <id`>.
  - Note that FwdExReq subsumes FlushReq/FlushRep in the original MSI protocol.
- Caches can send **ShRep(id, id`, data(a))** and **ExRep(id, id`, data(a))**.
  - These messages go through the cache-to-cache links, bypassing the directory.

(4 pt) Describe a system in which this directory-based MOSI protocol would provide significant advantages compared to the baseline MSI protocol.

MEMORY CONSISTENCY + SYNCHRONIZATION

(3 Points) Is it possible to translate code that assumes sequential consistency to the RISC-V weak memory consistency model? Explain.

B) (3 Points) Given the instruction sequences below, check all possible combinations of P1.r2, P2.r2 after both threads have executed, assuming an ISA that is sequentially consistent. X and Y are non-overlapping, and initially X = 0, Y = 0,

P1:	**P2:**
`li r1, 1`	`li r1, 2`
`st r1, X`	`st r1, Y`
`lw r2, Y`	`ld r2, X`

```
[ ] P1.r2 = 0; P2.r2 = 0
[X] P1.r2 = 0; P2.r2 = 1
[X] P1.r2 = 2; P2.r2 = 0
[X] P1.r2 = 2; P2.r2 = 1
```

C) (2 Points) Given the same instruction sequences and initial conditions from part B, which of the following combinations are possible under an ISA with TSO memory consistency model.

```
[X] P1.r2 = 0; P2.r2 = 0
[X] P1.r2 = 0; P2.r2 = 1
[X] P1.r2 = 2; P2.r2 = 0
[X] P1.r2 = 2; P2.r2 = 1
```

D) (4 Points) In general, what is the difference between a weak and a strong memory consistency model? Give one advantage of each.

(10 points) The following RISC-V assembly encodes a request-response relationship between two threads. The requestor thread puts work in a memory location `request` and then sets `go` = 1. It then spins waiting on the responder to produce the response. The responder thread spins waiting for work from the master, by checking if `go` has been set. Once available, it reads the request data, computes the response, and writes the response data to a memory location `response` before setting `done` = 1.

Requestor thread:

```
      li a0, 1
      sw data, request
      sw a0, go
spin:
      lw a1, done
      beqz a1, spin
      lw a2, response
      sw zero, done
```

Responder thread:

```
spin:
      lw a1, go
      beq a1, spin
      lw a2, request
      sw zero, go
      … a3 = process request
      sw a3, response
      li a0, 1
      sw a0, done
```

Under a fully relaxed memory consistency model, insert fences where necessary to ensure this code functions correctly? Use the least restrictive fences for full credit. **Assume each thread executes its code only once.**