CS 152 Computer Architecture and Engineering
CS 252 Graduate Computer Architecture

Midterm #1
March 4th, 2019
SOLUTIONS

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

This is a closed book, closed notes exam.
80 Minutes, 19 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	15	--
4	15	--
5	15	15
Grad Supplement	--	50
TOTAL	80	80

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

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

For the final column "Execution Time", mark whether the following modifications **increase**, **decrease**, have **negligible effect**, or whether the modification will have a potentially significant but **ambiguous** effect. Explain your reasoning. If the modification has an **ambiguous** effect, describe the tradeoff in which it would be a significantly beneficial modification or in which it would a significantly detrimental modification (i.e., as an engineer would you suggest using the modification or not and why?).

		Instructions / Program	Cycles / Instruction	Seconds / Cycle	Execution Time
a)	Improving branch predictor accuracy.	Negligible Branch predictors are not part of the ISA, so they are not visible to software.	Decrease There will be fewer cases where bubbles are inserted to recover from a mispredicted branch.	Negligible No implementation details are specified, so it could just be a more efficient history model.	Decrease The CPI advantage will be large, especially for a complex machine, making this a very positive change.

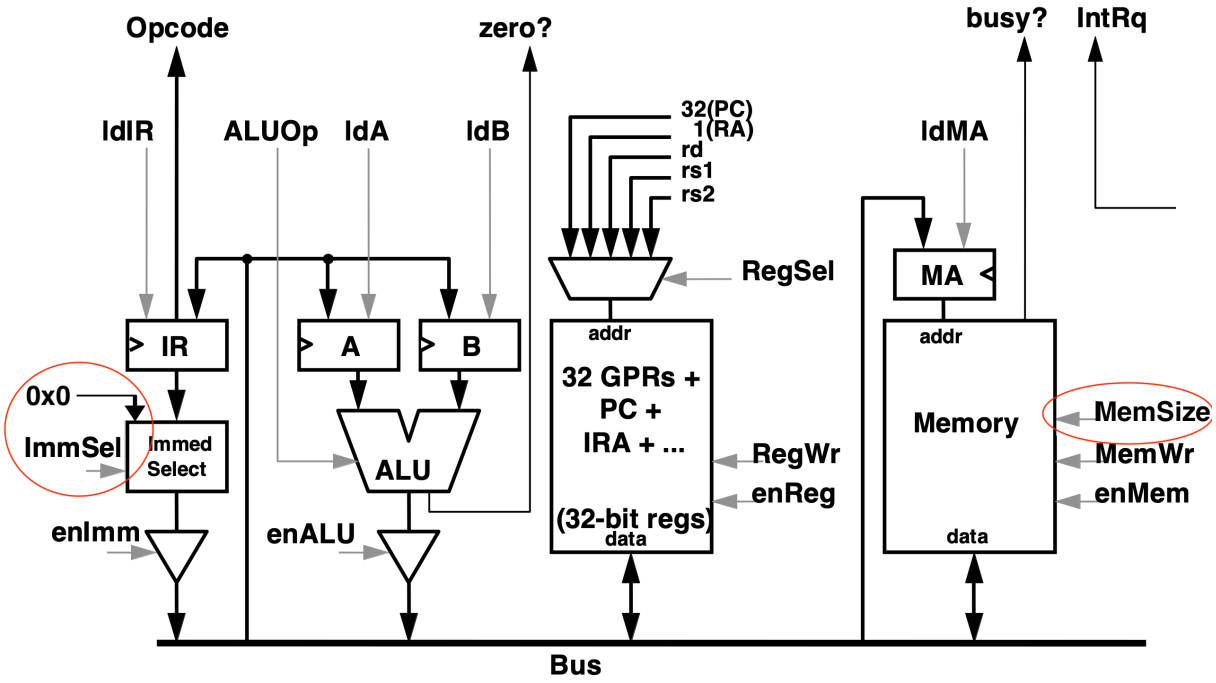
b)	Adding a vector (SIMD) extension to the ISA	Decrease Each instruction does more work, so fewer instructions are required to express a given algorithm, assuming there is some data-level parallelism.	Increase Instructions can have more than one load / store / cache miss each. Also, many easily-parallelizable instructions that would have otherwise had very low CPI can be merged into one instruction, so a higher portion of the remaining instructions will contain control and data hazards, raising CPI. This second factor shows that the CPI increase is basically guaranteed.	Increase One of the most impactful changes on cycle time would be adding wider register files to read out SIMD registers. This likely has an even bigger impact than the parallel ALUs. **OR** Negligible New hardware is intrinsically parallel	Decrease Data-level parallelism that can be expressed as short vectors is ubiquitous, so it is nearly certain that the Instructions / Program advantage will outweigh the drawbacks. Note that the CPI drawback only applies if you use the SIMD instructions, and using a SIMD instruction is still a win over the instruction it replaces.
c)	Adding an explicit load-delay slot.	Increase NOPs must be added wherever the slot cannot be filled with an instruction not needing the load data.	Decrease No bubbles need to be inserted to cover load-use delays.	Negligible This simplifies the control, but the baseline is likely to resolve back-to-back load-use hazards with bubbles, not exceptionally long bypass paths.	Ambiguous Program and microarchitecture dependent.

d)	Adding software-prefetching instructions.	Increase The software prefetching instructions must be added, which increases static and dynamic instruction count.	Decrease Software prefetch instructions execute quickly, since they don't block, and ideally will prevent long stalls that wait for memory.	Negligible Software prefetches are no more expensive to implement than loads from a microarchitectural implementation perspective, so they won't slow the pipeline. **OR** Increase: More instructions means more complexity and delay in complex implementations due to control cost.	Ambiguous The cache pressure and performance hit from the extra instructions may outweigh the benefit, given how hard it is to software prefetch correctly. **OR** Decrease, (under assumption they are being used effectively)

e)	Adding another level in the page-table hierarchy.	Negligible, assuming a hardware page table walker. **OR** Increase, if there is a software page table walker or even with a HWPT walker if taking frequent page faults.	Increase TLB misses take longer to handle, because they involve more memory accesses.	Negligible Traversing an extra level of the hierarchy takes more cycles, but is not going to meaningfully increase the delay or complexity of the circuits that handle TLB misses.	Decrease A deeper hierarchy slows TLB misses, while the advantages of a deeper page-table hierarchy are not in program execution time. Actual advantages: Freeing up physical address space for a constant virtual address length. **OR** Accommodating a larger virtual address space.

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!

A Cheat Sheet for the Bus-based RISC-V Implementation

For your reference, we've also included the actual bus-based datapath as well as rehash of some important information about microprogramming in the bus-based architecture.

Remember that you can use the following ALU operations:

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 H1-2: Available ALU operations

Remember that the μBr (*micro*branch) column in Table H1-3 represents a 3-bit field with six possible values: N, J, EZ, NZ, D, and S.

- If μBr is N (next), then the next state is simply (*current state* + 1).
- If it is J (jump), then the next state is *unconditionally* the state specified in the Next State column (i.e., it's an unconditional microbranch).
- If it is EZ (branch-if-equal-zero), then the next state depends on the value of the ALU's *zero* output signal (i.e., it's a conditional microbranch). If *zero* is asserted (== 1), then the next state is that specified in the Next State column, otherwise, it is (*current state* + 1).
- NZ (branch-if-not-zero) behaves exactly like EZ, but instead performs a microbranch if *zero* is not asserted (!= 1).
- If μBr is D (dispatch), then the FSM looks at the opcode and function fields in the IR and goes into the corresponding state.
- If μBr is S, the μPC spins if *busy?* is asserted, otherwise 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` is 1, then it is a write; otherwise it's a read. `enReg` is a general enable control for the register file. If `enReg` is 1, then the register reads or writes depending on `RegWr`. If `enReg` is 0, then nothing is done, regardless of the value of `RegWr`.
- MemWr and enMem function in an analogous way for the memory.

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.

1.B (5 Points) Performance of your `strchridx` implementation

How many cycles does your `strchridx` instruction take for each of the following inputs? Assume that all memory accesses complete in a single cycle (just for the CPI calculation – i.e., you must still use spin states). Include all the cycles from executing STRCHRIDX0 to the instruction that jumps back to FETCH0.

For the implementation on the next page:

- Fetch/dispatch: 3 cycles
- Prologue: 3 cycles
- Iterations: 4 cycles per unmatched character
- Epilogue: 5 cycles for successful match, 4 for unsuccessful

NOTE: it was confusingly worded whether to count the initial fetch/dispatch. We took both!

```
char *s1 = "hello world";
// Case 1
strchridx(s1, 'h');
```

3 + 3 + 5 = 11 cycles if counting initial fetch 3 + 5 = 8 cycles if not

```
// Case 2
strchridx(s1, 'd');
```

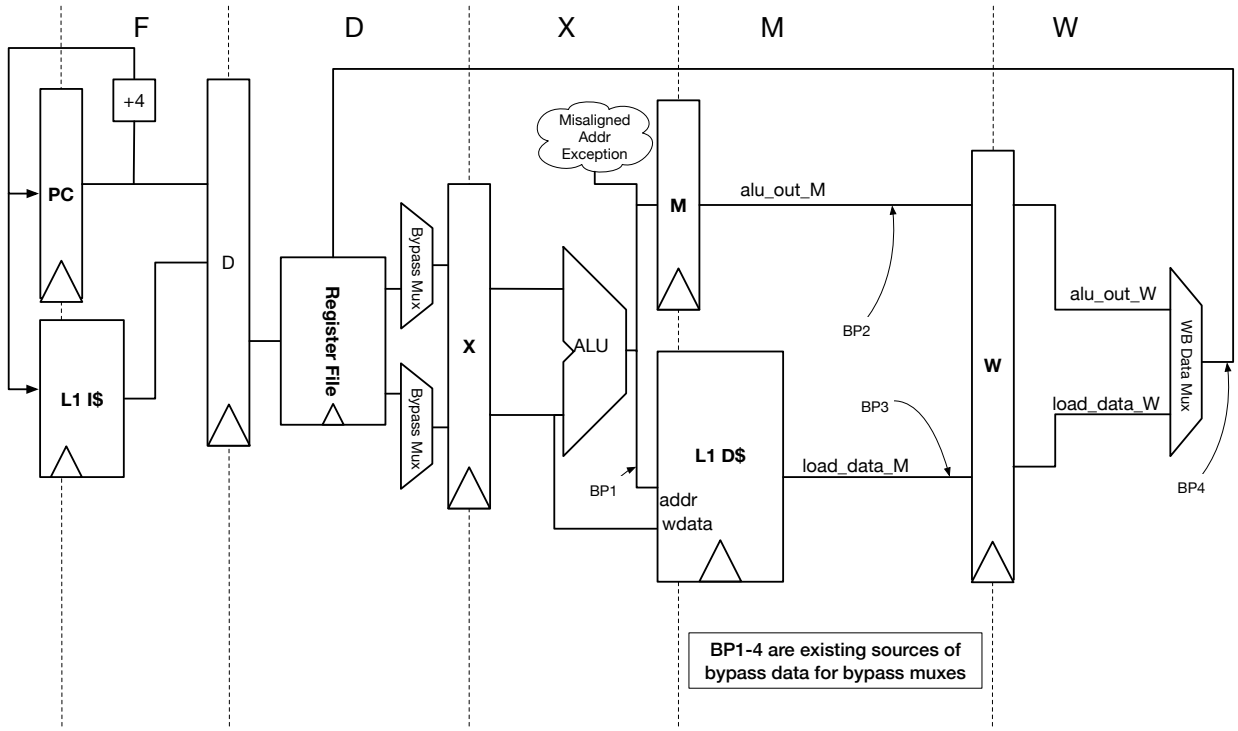
3 + 3 + 4*10 + 5 = 51 cycles if counting initial fetch 3 + 4*10 + 5 = 48 cycles if not

```
// Case 3
strchridx(s1, 'q');
```

3 + 3 + 4*11 + 4 = 54 cycles if counting initial fetch 3 + 4*11 + 4 = 51 cycles if not

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	A, MA = rs1	0	rs1	0	1	1	*	*	0	1	*	0	*	*	0	N	*
	B = rs2	0	rs2	0	1	0	1	*	0	0	*	0	*	*	0	N	*
	rd = A	0	rd	1	1	*	0	*	0	0	*	0	*	*	0	N	*
loop	A = Mem	0	*	*	0	1	0	*	0	0	0	1	0	*	0	S	*
	μbeq A, 0, fail	0	*	*note*	*note*	0	0	COPY_A	*note*	*	*note*	*note*	*	*	*note*	EZ	fail
	μbeq A, B, match A = rd	0	rd	0	1	1	0	SUB	0	*	*	0	*	*	0	EZ	match
	μjump loop MA, rd = A+1	0	rd	1	1	*	0	INC_A_1	1	1	*	0	*	*	0	J	loop
fail	A = 0	0	*	*	0	1	*	*	0	*	*	0	*	zero	1	N	*
	μjump FETCH0 rd = A-1	*	rd	1	1	*	*	DEC_A_1	1	*	*	0	*	*	0	J	FETCH0
match	B = rs1	0	rs1	0	1	0	1	*	0	*	*	0	*	*	0	N	*
	μjump FETCH0 rd = A-B	*	rd	1	1	*	*	SUB	1	*	*	0	*	*	0	J	FETCH0

Problem 3: (15 Points) 5-Stage Pipelines *(CS152 ONLY)*



3.A (2 Points) Speculation in the 5-stage pipeline

Even a simple, in-order pipelined processor makes use of speculative execution. For the 5-stage pipeline above, assume that there is no virtual memory, and that misaligned accesses are checked in the Execute stage. For the instruction sequence below, complete the execution diagram and circle the cycles in which the second `add` is being executed speculatively. Justify your response!

Clock Cycle	0	1	2	3	4	5	6	7	8	9
`add x1, x2, x0`	F	D	X	M	W					
`lw x3, 0(x2)`		F	D	X	M	W				
`add x3, x4, x5`			F	D	X	M	W			

Speculative execution highlighted.

- It is speculative until it is known that neither the `add` nor either preceding instruction will cause an exception.
- The `lw` can except through the end of the Execute stage, so any execution of following instructions is speculative while the `lw` is in the Fetch, Decode, or Execute stages.
- The `add` cannot except after it is decoded, as there are no arithmetic condition exceptions in RISC-V

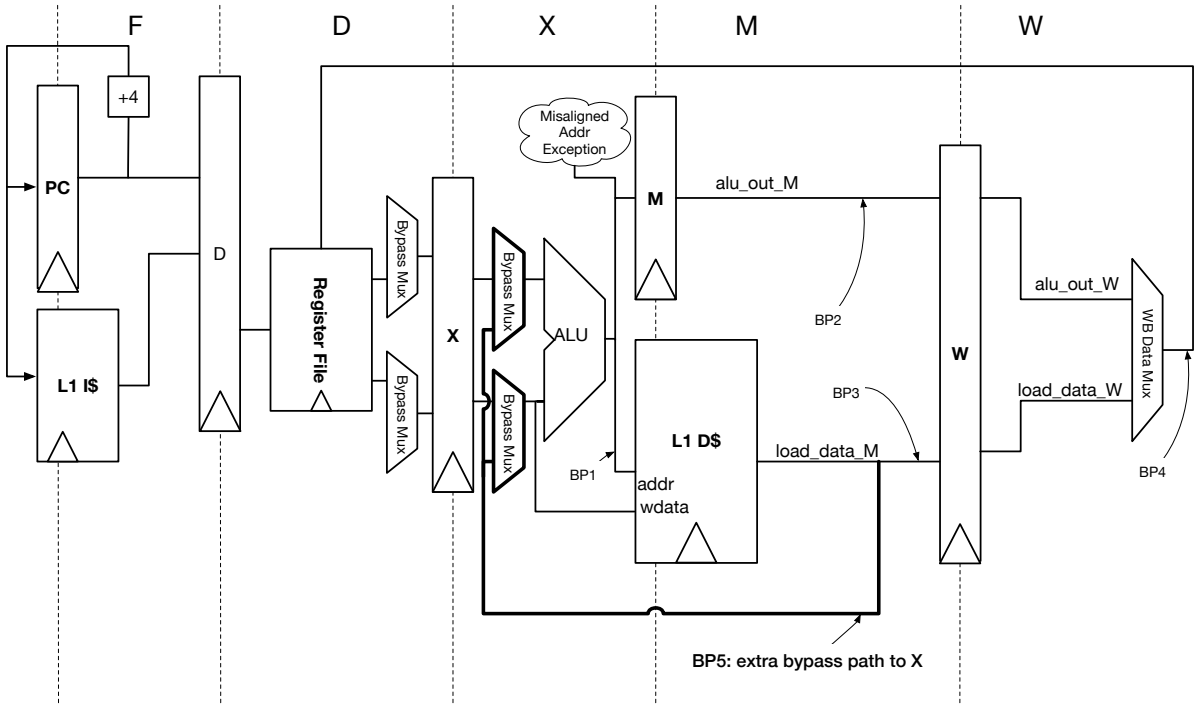
3.B (1 Point) Load-use delay

Given the 5-stage pipeline above, how long is the load-use delay? Answer in terms of how many bubbles must be added between a load and a dependent register-register instruction that is fetched right after the load.

One bubble must be added; this means that the load-use delay is one cycle.

3.C (4 Points) Modifying the load-use delay

Consider the bypass path shown in bold below.



How would this bypass path affect CPI? Seconds per cycle?

- The bypass path would decrease CPI, as no bubbles would need to be added between loads and immediately following instructions depending on the load data.
- Seconds/cycle would increase dramatically, as it nearly merges two stages into one.

Would you recommend adding this bypass path? Justify your response.

This is a bad bypass path. In the best case, it saves a cycle when using load data in the instruction immediately following a load. Given the theoretical best CPI of 1 in the 5-stage pipeline, this represents a 100% maximum theoretical improvement, even in an unrealistically optimistic scenario, such as executing the following code with a single-cycle magic memory. Even in this unrealistic "best case," the seconds/cycle penalty would erode all the gains.

```
lw x2, 4(x2) # unrealistic linked-list pointer chase example
lw x2, 4(x2)
...
```

3.D (2 Points) RAW hazards through memory

Consider the following instruction sequence.

```
sw x1, 0(x2)
lw x3, 0(x2)
```

Assume a "magic memory" that reads and writes in a single cycle, along with the same baseline microarchitecture from 2.A. Draw a pipeline execution diagram and depict the RAW dependency with an arrow. Should any bubbles be inserted for correct execution? How many?

Clock Cycle	0	1	2	3	4	5	6	7	8	9
`add x4, x4, x5`	F	D	X	M	W					
`sw x1, 0(x2)`		F	D	X	M	W				
`lw x3, 0(x2)`			F	D	X	M	W			

No bubbles! A few different arrow drawings were accepted (see rubric).

3.E (6 points) Multi-cycle writes

Now consider a slightly more realistic memory system with caches. These parameters are used throughout all of (2.D). **Cache misses are ignored throughout this question.**

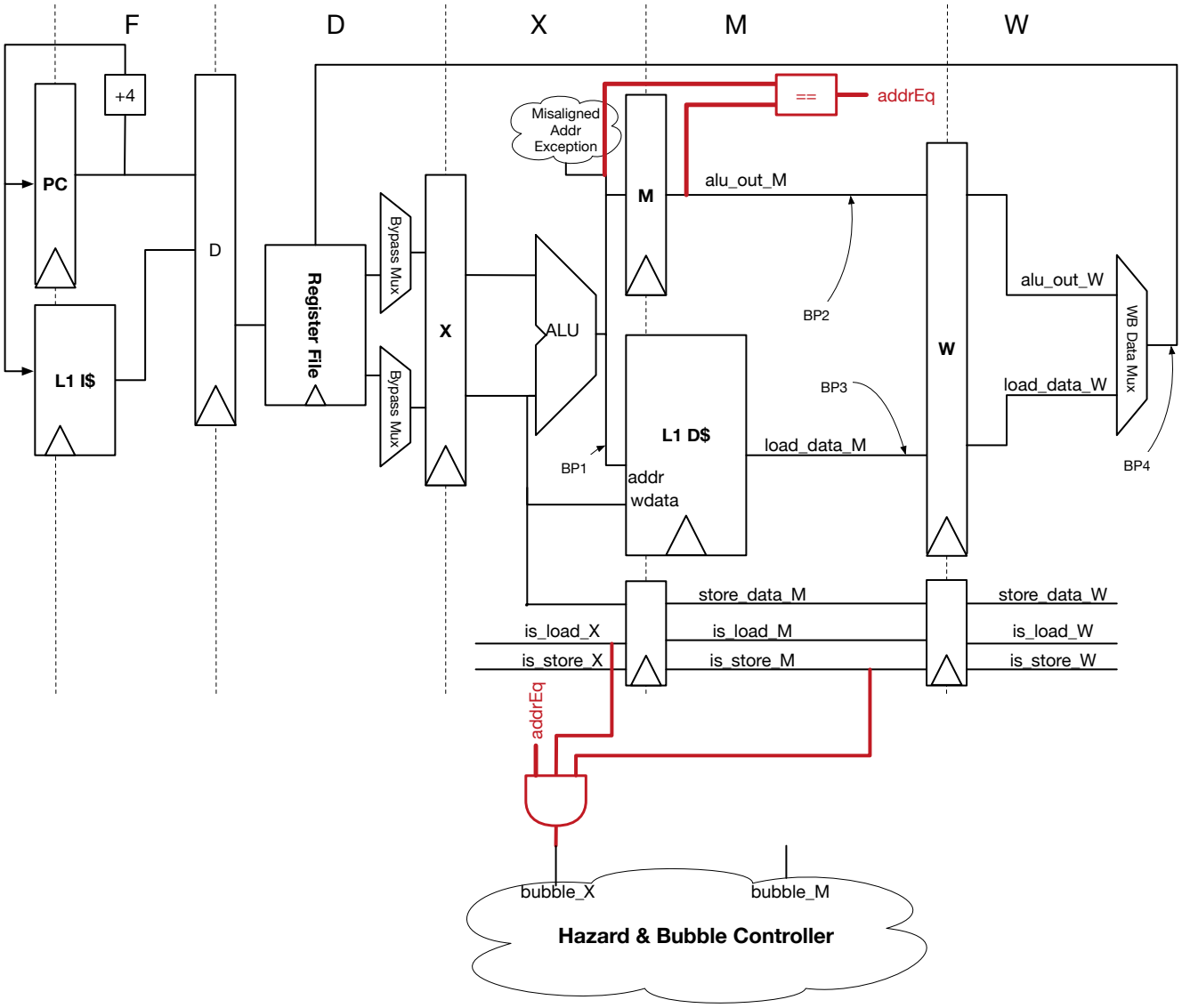
- L1 cache read hits complete in a single cycle
- L1 cache writes have a two cycle *latency* to complete
- L1 Reads and writes still only have a single cycle *occupancy*

When reusing the existing pipelined datapath with this new cache, no new structural hazards are added, as the write *occupancy is still one cycle*. Therefore, this baseline datapath can accommodate the two-cycle write with no extra pipeline stages.

i) Describe a new type of hazard that will need to addressed in the pipeline and give an example instruction sequence that will cause such a hazard to occur.

The RAW hazard through memory from a load following a store to the same address will need to be explicitly detected by the pipeline.

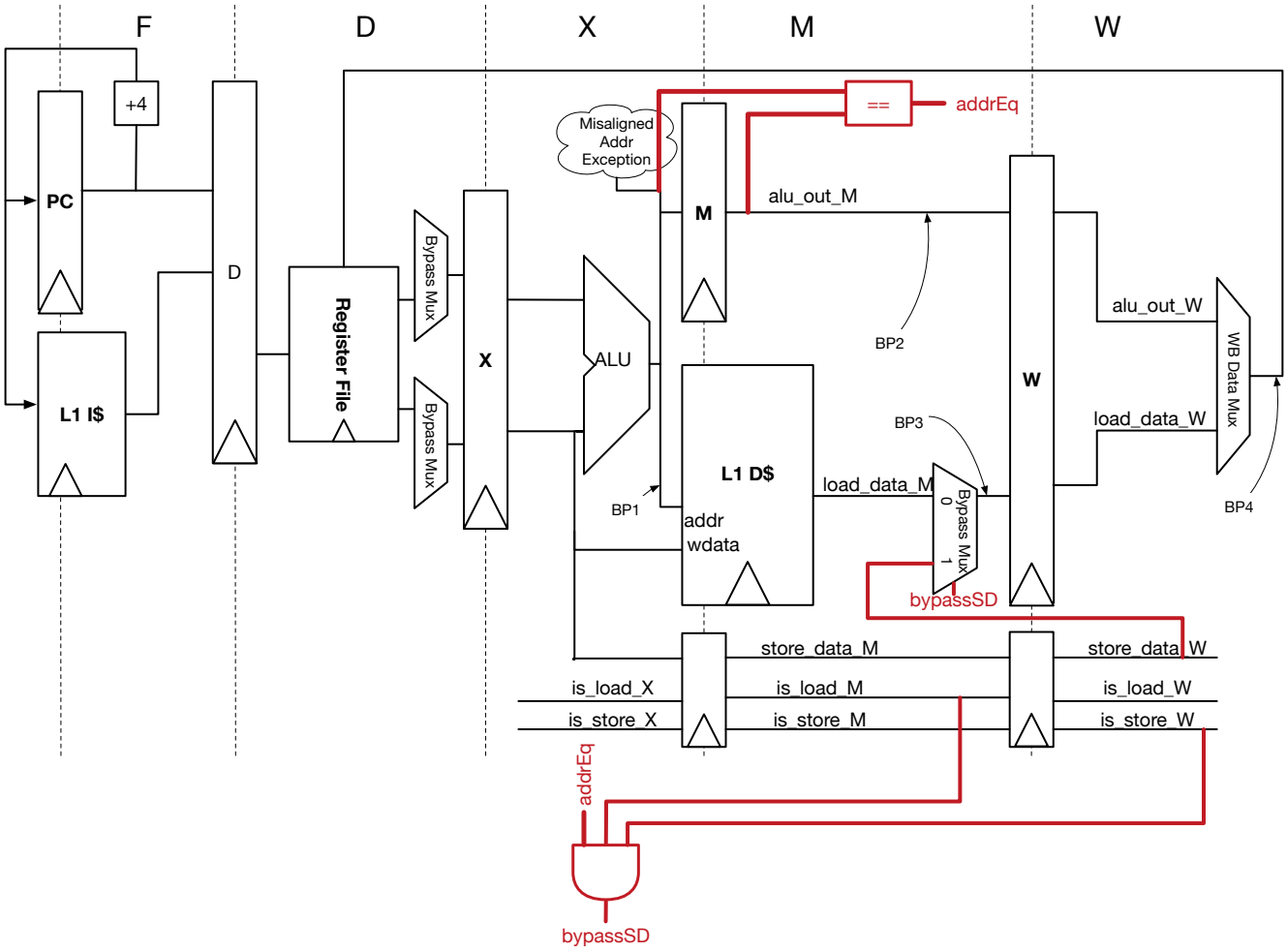
```
sw x1, 0(x2)
lw x3, 0(x2)
```



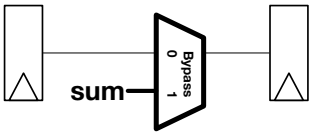
ii) Using the above diagram as a template, draw a new <u>interlock</u> that makes the following instruction sequence execute correctly. You may add new gates and/or basic arithmetic units (adders, comparators, etc). For full credit, minimize the overall impact on CPI. Wire the Boolean interlock signal to one or more of the `bubble_<stage>` signals, which insert a bubble in that stage on the current cycle; this bubble ends up ahead of the instruction that was in that stage. You may use labeled endpoints as "tunnels" to neatly connect wires without clutter. How many bubbles does it add for the following sequence?

```
sw x1, 0(x2)
lw x3, 0(x2)
```

It adds one bubble.



Example: muxing existing wire with tunneled, named signal. New items in bold.



iii) Using the above diagram as a template, draw a new <u>bypass path</u> that makes the following instruction sequence execute correctly with zero bubbles. You may add new gates and/or basic arithmetic units (adders, comparators, etc). You may also add muxes on an existing wire by drawing the mux over the wire. You may use labeled endpoints as "tunnels" to neatly connect wires without clutter.

```
sw x1, 0(x2)
lw x3, 0(x2)
```

There are multiple solutions. The one above is the simplest given the pre-supplied pipeline registers from the diagram.

Problem 4: (15 Points) Software Optimization *(CS152 ONLY)*

In this problem, we'll consider SAXPY kernel operating on 32-bit integer values:

```
for (i = 0; i < len; i++) {
     y[i] = a * x[i]+ y[i]
}
```

In this question, we'll study the performance of this kernel on two different microarchitectures. Specifically, we're interested in both the CPI and how many cycles-per-element (CPE) the kernel takes to execute.

Consider the following RV32IM assembly implementation of this kernel:

```
// x1 holds pointer to x
// x2 holds pointer to y
// x3 holds a
// x4 holds len

          add x5, x0, x0
LOOP:     bge x5, x4, DONE
          lw x6, 0(x1)
          mul x6, x6, x3
          lw x7, 0(x2)
          add x6, x6, x7
          sw x6, 0(x2)
          addi x1, x1, #4
          addi x2, x2, #4
          addi x5, x5, #1
          j LOOP
DONE:
```

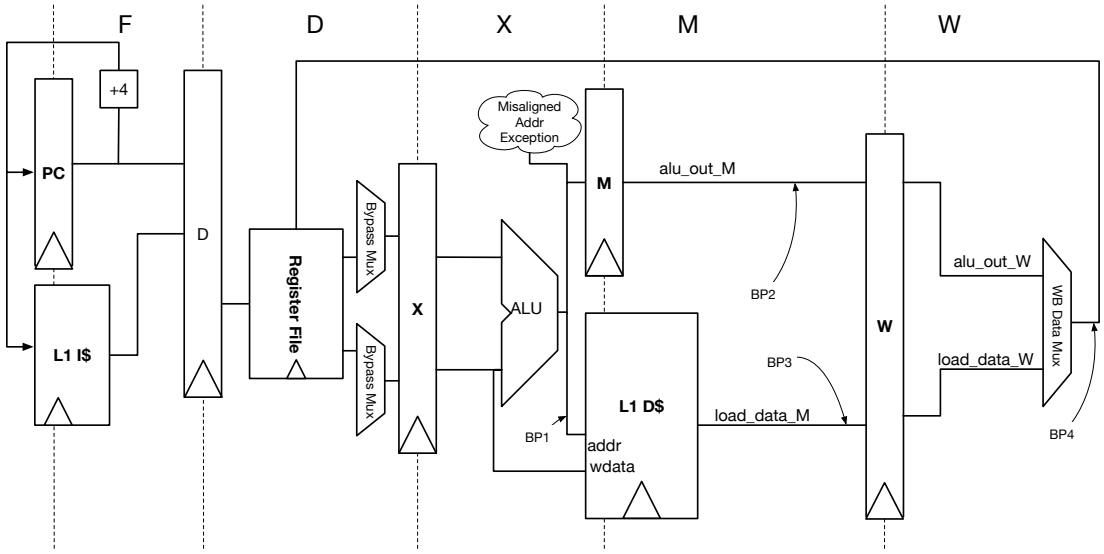
- **(1 Point)** How many instruction bytes are fetched per loop iteration?

  (10 instructions / iteration) * (4B / instruction) = 40 bytes

- **(1 Point)** How many data bytes are loaded per loop iteration?

  2 `lw` instructions = 8 bytes loaded

- **(8 Points)** Fill out the provided pipeline diagram on page 15 for a classic 5-stage in-order pipeline with full-bypassing, for the first 12 dynamic instructions. Assume no cache misses, and no branch prediction, and `len` > 2. Note that **unconditional** branches are resolved in D. What does CPI converge to as we increase the number of iterations executed? CPE? *You may need to wrap instructions back around to cycle 0 in the pipeline diagram.*



CPI converges to 1.3, which is equivalent to CPE = 13, as there are 10 instructions per iteration.

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
add x5, x0, x0	F	D	X	M	W																			
bge x5, x4, DONE		F	D	X	M	W																		
lw x6, 0(x1)			F	D	X	M	W																	
mul x6, x6, x3				F	D	D	X	M	W															
lw x7, 0(x2)					F	F	D	X	M	W														
add x6, x6, x7							F	D	D	X	M	W												
sw x6, 0(x2)								F	F	D	X	M	W											
addi x1, x1, #4										F	D	X	M	W										
addi x2, x2, #4											F	D	X	M	W									
addi x5, x5, #1												F	D	X	M	W								
j LOOP													F	D	X	M	W							
bge x5, x4, DONE															F	D	X	M	W					

- **(3 Points)** Give a reordering of the assembly that achieves a lower CPE, without adding new instructions. What is the CPE of the reordering?

```
            add x5, x0, x0
LOOP:       bge x5, x4, DONE
            lw x6, 0(x1)
            lw x7, 0(x2)
            mul x6, x6, x3
            add x6, x6, x7
            sw x6, 0(x2)
            addi x1, x1, #4
            addi x2, x2, #4
            addi x5, x5, #1
            j LOOP
DONE:
```

By moving the consumer of each load to be at least two instructions after the load, we avoid taking bubbles on load-use delays. However, we still take one bubble from the control hazard on the unconditional jump at the end of the loop, so the iteration takes 11 cycles.

CPE = 11
CPI = 1.1

- **(2 Points)** Name two other optimizations you could employ to improve the CPE (assuming you could completely rewrite the assembly implementation). Explain why they would reduce CPE for the provided kernel. *You do not have to write the code in this part.*

There are several potential answers. Two good answers are listed below.

a. Unrolling the loop: this would reduce the number of bubbles taken by the control hazard from the unconditional jump, but more importantly, it would allow the pointers and index to be incremented half as frequently, cutting the number of instructions (and therefore cycles) per iteration.
b. Calculate the pointer of `x[len]` and put it in `x4` before starting the loop, and use this as a loop bound by changing the branch to `bge x1, x4, done`. This allows us to remove the `addi x5, x5, #1` instruction.

One intuitive but incorrect answer is to delete the existing branch & jump and replace them single conditional backwards branch. This cuts the instruction count, but actually incurs a one-cycle longer branch penalty at the end of the loop due to the later resolution of conditional branches. Therefore, this change doesn't actually affect the value that CPE converges to as the vector gets infinitely long. Now, if we did have branch prediction this would be a very sensible optimization.

Problem 5: (15 Points) Virtual Memory and Aliasing

A) **(2 Points)** You are asked to design a virtually indexed, physically tagged cache. A page is 4096 bytes. The cache must have 64 lines of 256 bytes each. What associativity must the cache have in order for there to be no aliasing?

$(64 = 2^6) * (256 = 2^8) = 2^{14}$ bytes in the cache

Each way can be no bigger than the page size to avoid the potential for aliasing in a virtually-indexed, physically-tagged cache. This is due to the fact that all of the index bits must come from the page offset.

Therefore, minimum associativity $= \frac{cache\ size}{page\ size} = \frac{2^{14}}{2^{12}} = 2^2 =$ 4-way set associative.

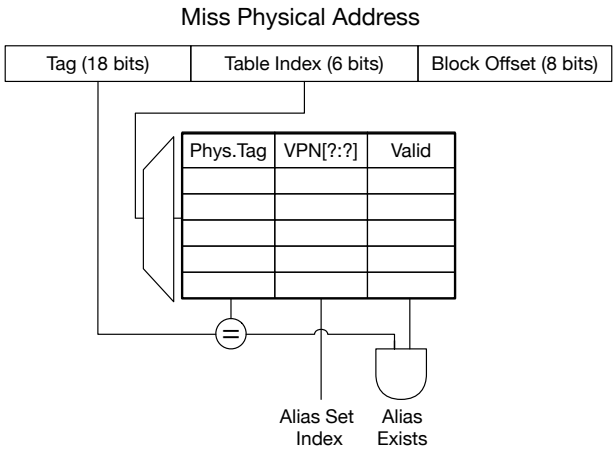
B) **(3 Points)** Assume the cache is direct-mapped, and suppose an alias exists for the physical address 0x80007100. Which sets in the L1 could contain the aliased entry? The sets are indexed starting from zero.

```
Tag    = addr[31:14]          VPN, PPN    = addr[31:12]
Index  = addr[13:8]           Page Offset = addr[11:0]
Offset = addr[7:0]
```

Therefore, bits [13:12] of the address represent the part of the index that comes from the virtual page number, which means that they are the only bits that will differ among aliases of the same line in the cache. These are the top two bits of the index.

Address 0x80007100 lives in index 0x31 in the cache. Since the top two bits of the index are from the VPN, not the PPN, the same physical line could map to indices 0x01, 0x11, 0x21, or 0x31.

C) **(2 Points)** To detect aliases, suppose we implement a table which has a single row per L1 cache line. Its structure is given below:



On a miss in the L1 cache, the table is indexed using the physical address bits in positions corresponding to the cache's index bits. If the physical tag matches and the entry is valid, the aliased line is moved into the new set, and the VPN bits in the table are updated. Otherwise, the L1 line pointed to by the table entry is evicted, and the table entry is updated with the physical tag and VPN bits of the missing line.

For the cache organization of part B, which bits, if any, of the VPN must be stored in the table to resolve aliases? Why?

In order to determine which set the pre-existing alias lives in, you need to be able to complete its index. The bottom four bits of the index (`addr[11:8]`) come from the page offset, so they match those of the newly-accessed virtual address. Therefore, we only need the top two bits of the virtual index, which come from the bottom two bits of the VPN.

`VPN[1:0]`, which is `VAddr[13:12]`

D) **(3 Points)** Suppose we want to load from two 4KiB arrays. First, we load every entry from `foo`, which is stored at virtual address 0x8000_0000, and then every entry from `bar`, which is stored at virtual address 0x8000_1000. If virtual addresses 0x8000_0000 and 0x8000_1000 map to physical addresses 0x3000 and 0x7000 respectively, how many bytes of `foo` will reside in L1 cache once we've finished loading from `bar`? Explain.

Since the physical address ranges of `foo` and `bar` do not overlap, there will be no aliases. However, the problem is that our alias table is direct-mapped based off the six physical address bits (`PAddr[13:8]`) in positions corresponding to the cache's index bits. Element zero of `foo` and element zero of `bar` have physical addresses have matching values of those six bits: 0x30. Therefore, they will collide in the alias resolution table, which does not have the capacity to consistently maintain both entries, even though

they do not have matching physical tags and therefore are not aliases. This would require one to be evicted, even though the lines' virtual indices do not conflict!

The effect is that every first access to a line in `bar` evicts the corresponding line in `foo`, even though they occupy different sets in the cache. **This means that zero bytes of `foo` will reside in the cache once we've finished loading from `bar`.**

E) **(5 Points)** To fix this, we could make the table associative. How many ways would you add and how many rows would you need to ensure you can always resolve aliases while completely removing the behavior of part D? Explain.

The goal is to allow multiple distinct lines (non-matching physical tags) to be tracked at the same physical index in the alias resolution table. Since there are four sets that could hold these lines per Q5.B, we only need to hold 4 entries to ensure that we do not create artificial conflicts. Therefore, we would need four ways.

Secondly, if we kept the row count the same, we'd have 4x more entries (rows * associativity) than lines in our L1 cache. And since the sets that would alias are the only four sets the share their block offset, we can just use the block offset to index into the table → we can use ¼ as many rows.

Comments:

One way to think about this solution is that we've built an auxiliary, 4-way set-associative, L1 cache to "simulate" how a well-behaved VIPT L1 cache would behave. It has the same dimensions as the alias-free VIPT cache in part A, it just doesn't store the data.

Any larger table with the same associativity would work just fine. If you just suggested increasing associativity without decreasing the number of lines, you'd have something that would look very similar to the tag array of inclusive, 64KiB L2 cache. This was the scheme mentioned in class for resolving aliases.