

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

**Midterm #1
March 4th, 2019
Professor Krste Asanović**

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.

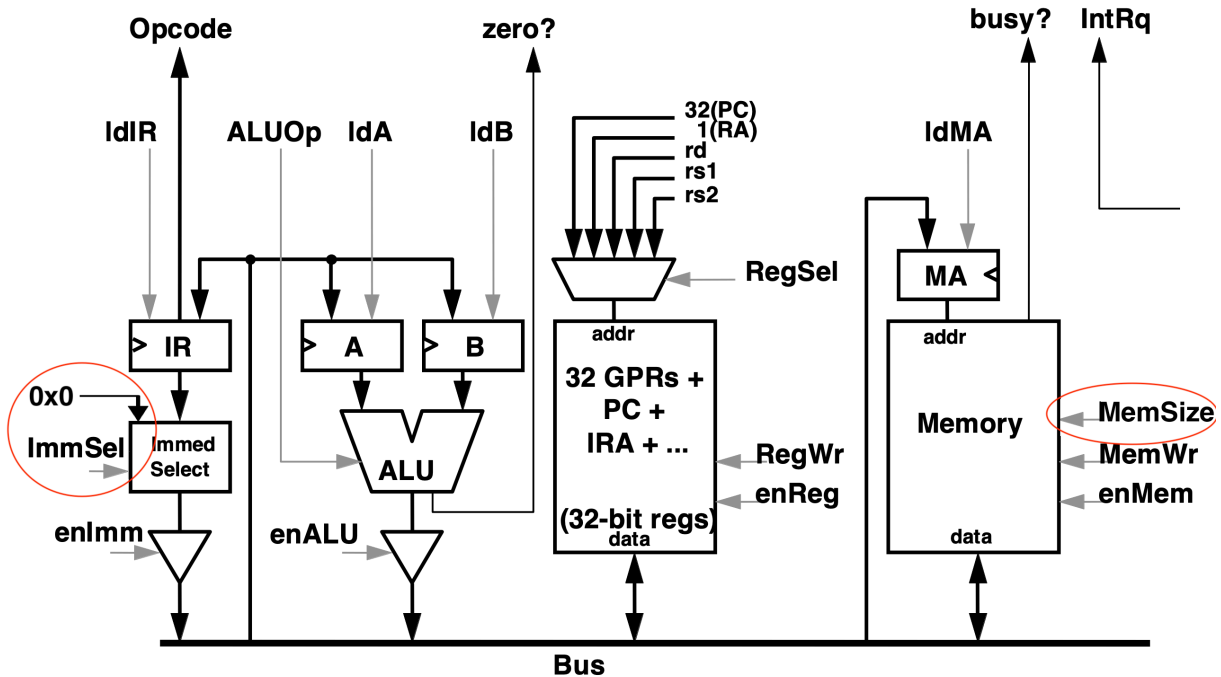
For the final column “Execution Time”, mark whether the following modifications **increase**, **decrease**, have **negligible effect**, or whether the modification will have a potentially significant but **ambiguous** effect. Explain your reasoning. If the modification has an **ambiguous** effect, describe the tradeoff in which it would be a significantly beneficial modification or in which it would be 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.				
b)	Adding a vector (SIMD) extension to the ISA				

c)	Adding an explicit load-delay slot.				
d)	Adding software-prefetching instructions.				
e)	Adding another level in the page-table hierarchy.				

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 μ Br (*microbranch*) column in Table H1-3 represents a 3-bit field with six possible values: N, J, EZ, NZ, D, and S.

- If μ Br 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 ($\neq 1$).
- If μ Br is D (dispatch), then the FSM looks at the opcode and function fields in the IR and goes into the corresponding state.
- If μ Br is S, the μ PC 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 `strchr` 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!

```
strchr 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 `strchr` implementation

How many cycles does your `strchr` 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 `STRCHR0` to the instruction that jumps back to `FETCH0`.

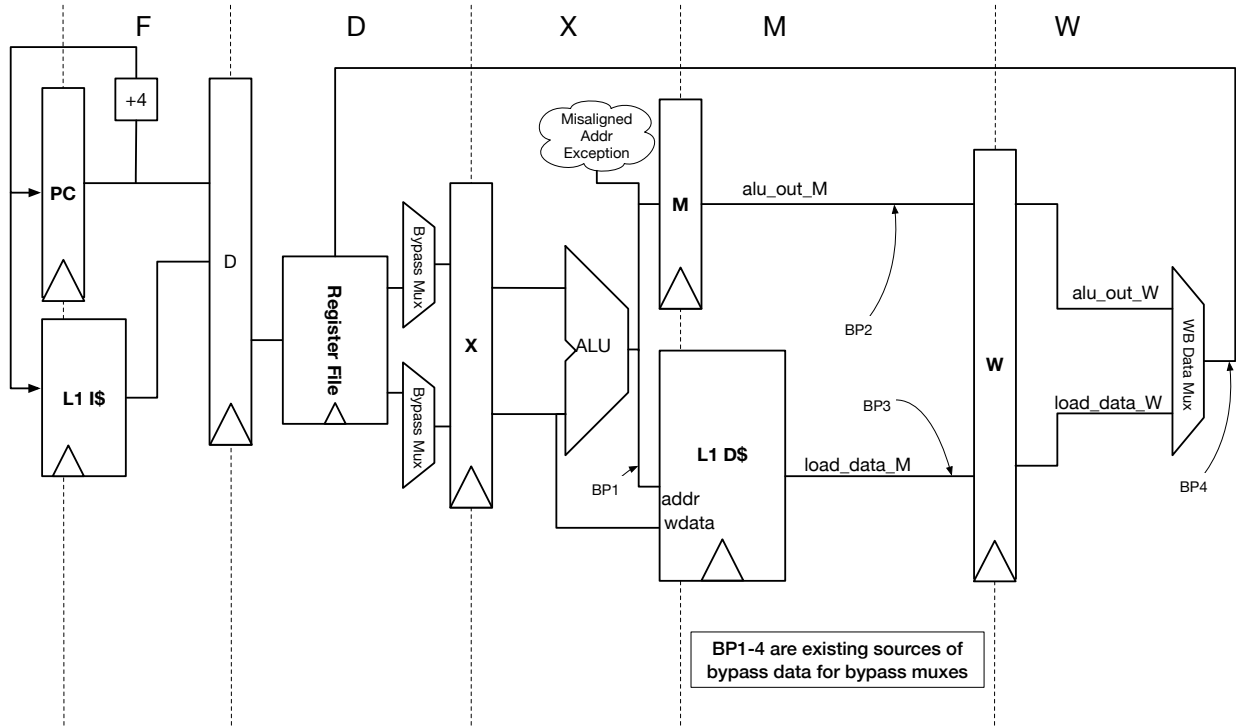
```
char *s1 = "hello world";  
// Case 1  
strchr(s1, 'h');
```

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

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

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																	

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!

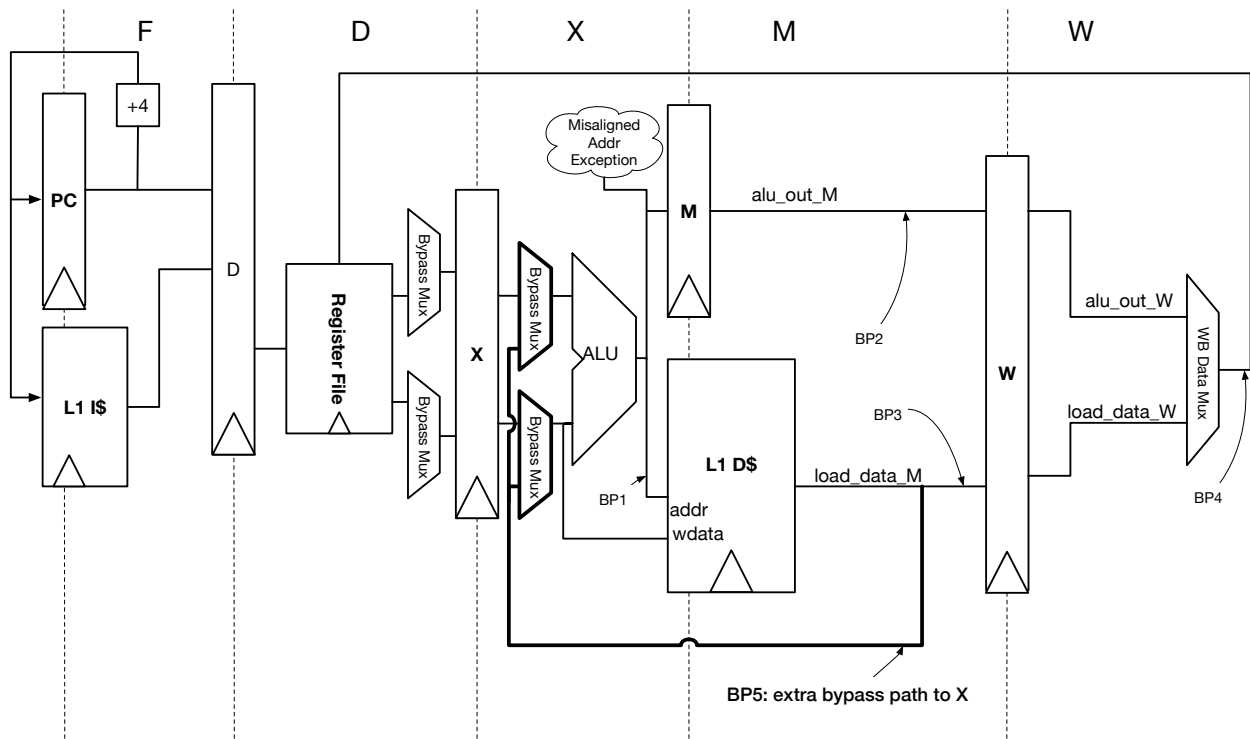
Clock Cycle	0	1	2	3	4	5	6	7	8	9
add x1, x2, x0	F	D	X	M	W					
lw x3, 0(x2)										
add x3, x4, x5										

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.

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?

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

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)										
lw x3, 0(x2)										

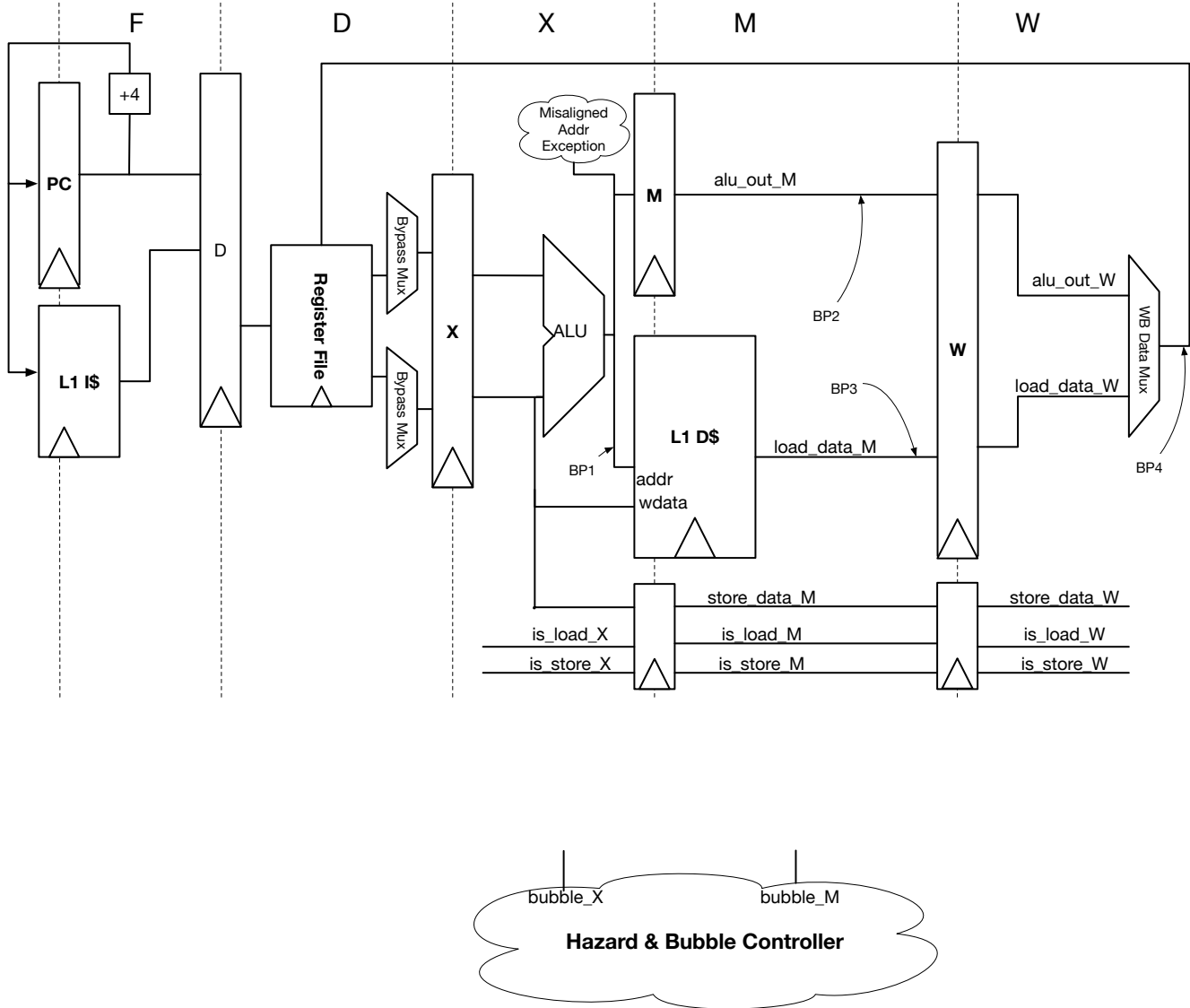
3.E (6 points) Multi-cycle writes

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.

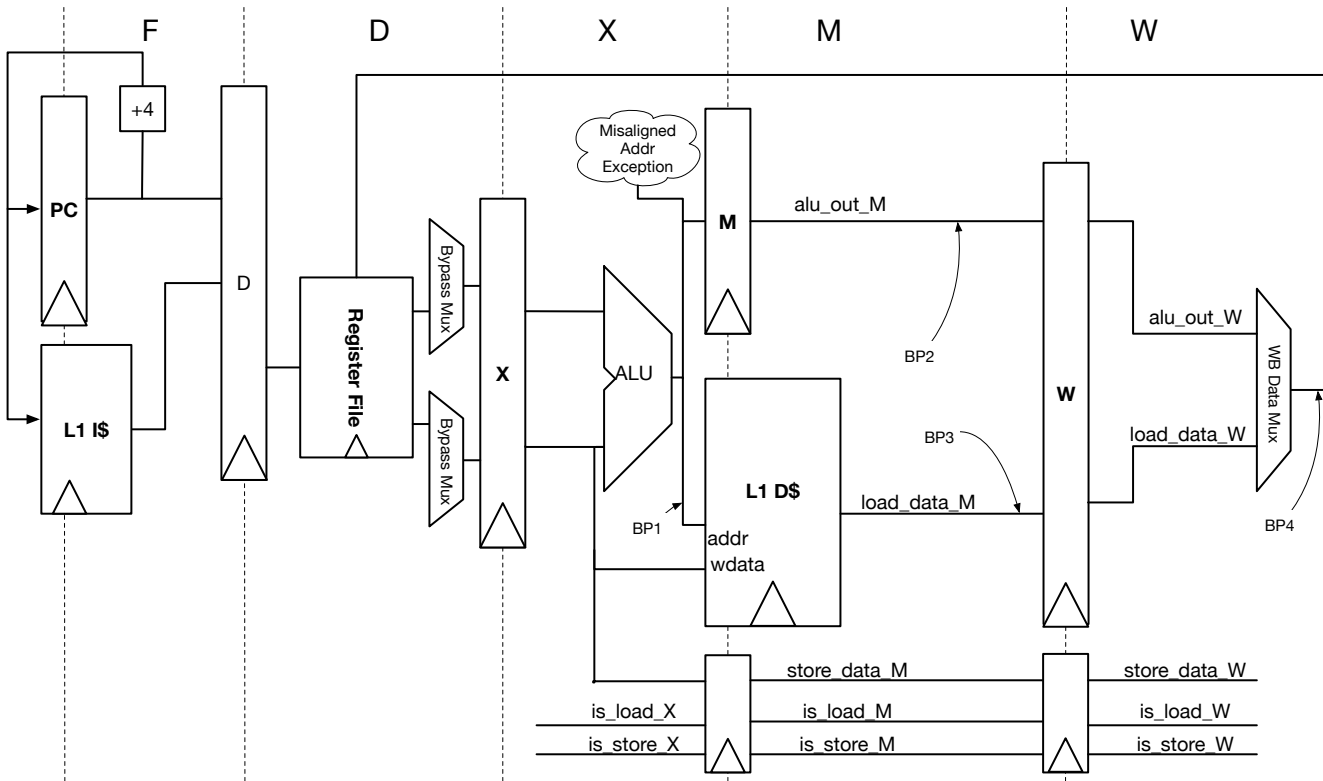
i) Describe a new type of hazard that will need to be addressed in the pipeline and give an example instruction sequence that will cause such a hazard to occur.



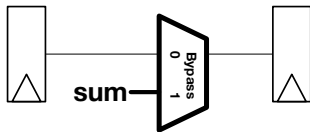
ii) Using the above diagram as a template, draw a new interlock that makes the following instruction sequence execute correctly. You may add new gates and/or basic arithmetic units (adders, comparators, etc). For full credit, minimize the overall impact on CPI. Wire the Boolean interlock signal to one 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)
```



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



iii) Using the above diagram as a template, draw a new bypass path 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)

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.

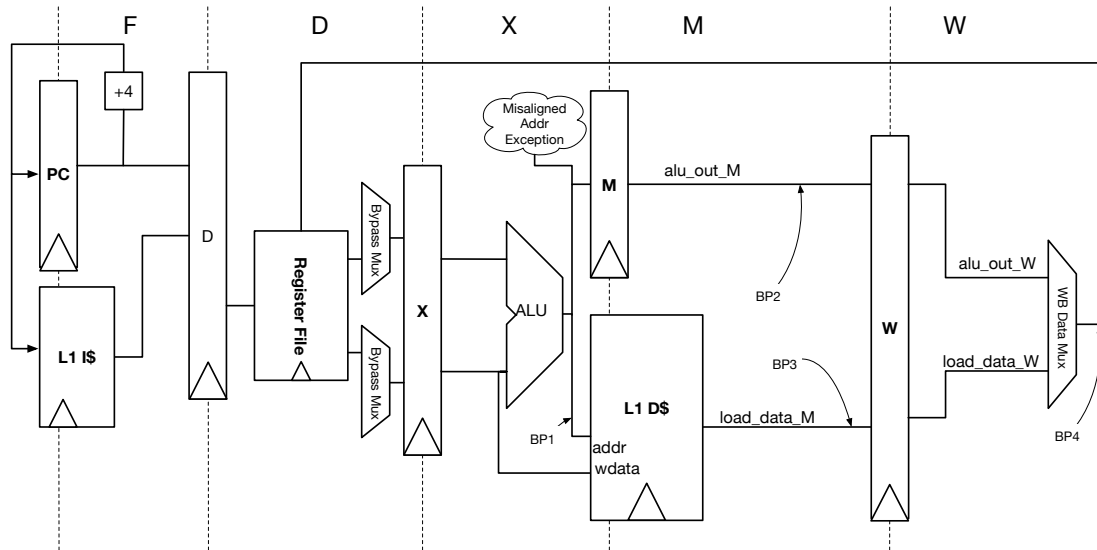
Consider the following RV32IM assembly implementation of this kernel:

```
// x1 holds pointer to x  
// x2 holds pointer to y  
// x3 holds a  
// x4 holds len  
  
LOOP:    add x5, x0, x0  
        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:
```

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

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

C) (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.



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																								
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																								
bge x5, x4, DONE																								

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

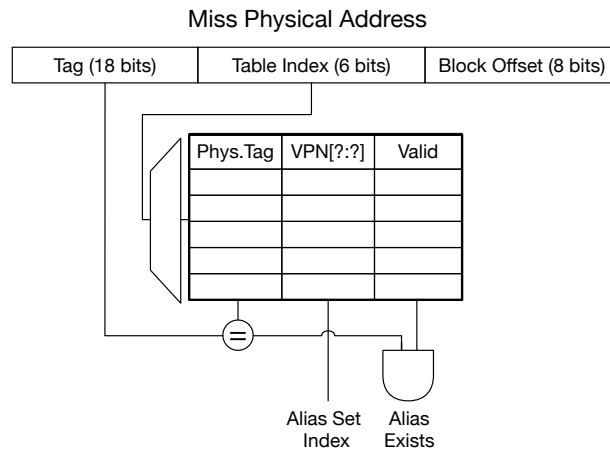
E) **(2 Points)** Name two other optimizations you could employ to improve the CPE (assuming you could completely rewrite the assembly implementation). Explain why they would reduce CPE for the provided kernel. *You do not have to write the code in this part.*

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?

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.

- 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?

- 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.

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.