

Computer System Architecture
6.823 Quiz #1
March 7th, 2014
Professors Daniel Sanchez and Joel Emer

This is a closed book, closed notes exam.

80 Minutes
16 Pages

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Show your work to receive full credit.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz.

Part A	_____	40 Points
Part B	_____	35 Points
Part C	_____	25 Points

TOTAL _____ **100 Points**

Part A: Execute Data Instruction (40 pts)

One day, Ben Bitdiddle started an EDSACjr-based company. Ben wanted to leverage the speed of read-only memory and avoid the inherent hazards of the Princeton architecture, so he went with a Harvard architecture. Unfortunately, Ben's system didn't have any index registers, so he couldn't write self-modifying code. That meant there were a large number of programs he couldn't implement anymore. Ben decided to add an instruction to solve this problem. He called his new instruction `EXD`, for execute data. The `EXD` instruction treats the contents of the accumulator as a new instruction and executes whatever that instruction may be. If the accumulator does not contain a valid instruction, then `EXD` falls back on the processor's fault handling for bad instructions (which you needn't worry about).

For example, from Handout #1 the instruction `ADD 6` (which adds the contents of memory at address six to the accumulator) is encoded as: `0000 1000 0000 0110`.

Therefore if the contents of the accumulator are `0000 1000 0000 0110`, the `EXD` instruction will interpret the accumulator as an `ADD 6` instruction, and add the contents of memory at address six to the accumulator (now interpreted as the *number*: `0000 1000 0000 0110 = 2054`). So if memory at address six holds the value one, then the accumulator will become `0000 1000 0000 0111`. (Which can be interpreted either as the instruction `ADD 7` or the number 2055.)

To simplify writing assembly code, Ben Bitdiddle also augments the EDSACjr's instruction set with a load instruction, `LD n`. This load simply places the value in memory address `n` into the accumulator: $ACC \leftarrow Mem[n]$. `LD` is encoded as `01011 n`; that is, the opcode is `01011`.

Question 1 (5 points):

When Ben shows his idea to Alyssa P. Hacker, she points out that `EXD` could cause an infinite loop. Provide a specific code sequence that illustrates Alyssa's point, using `EXD` to loop forever.

Question 2 (10 points):

Ignoring Alyssa's observation, Ben decided to implement his EXD instruction for the EDSACjr, but he started having trouble figuring out how to use it. Help Ben by writing a series of EDSACjr instructions that will perform an indirect reduced add (that is, the instructions will take a vector of pointers, follow each pointer, and sum up the values stored at the locations in memory that the pointers specify). In C++, this might look something like:

```
int s=0;
for (int i=0; i < 10; i++){
    s += *A[i];
}
```

Fill in the template below with assembly code for this program on the Harvard EDSACjr. You can define memory contents for both the data and instruction memories.

Data Mem	
Addr	Data
A:	120
	107
	122
	130
	151
	112
	132
	109
	140
	117

s: 0
i: 10

107: 40
109: 10
112: 24
117: 50
120: 5
122: 10
130: 20
132: 29
140: 22
151: 12

one: 1

Instr Mem	
Addr	Data
Loop:	LD i
	SUB one
	BLT Done
	STORE i

CLEAR
BGE Loop
Done: HLT

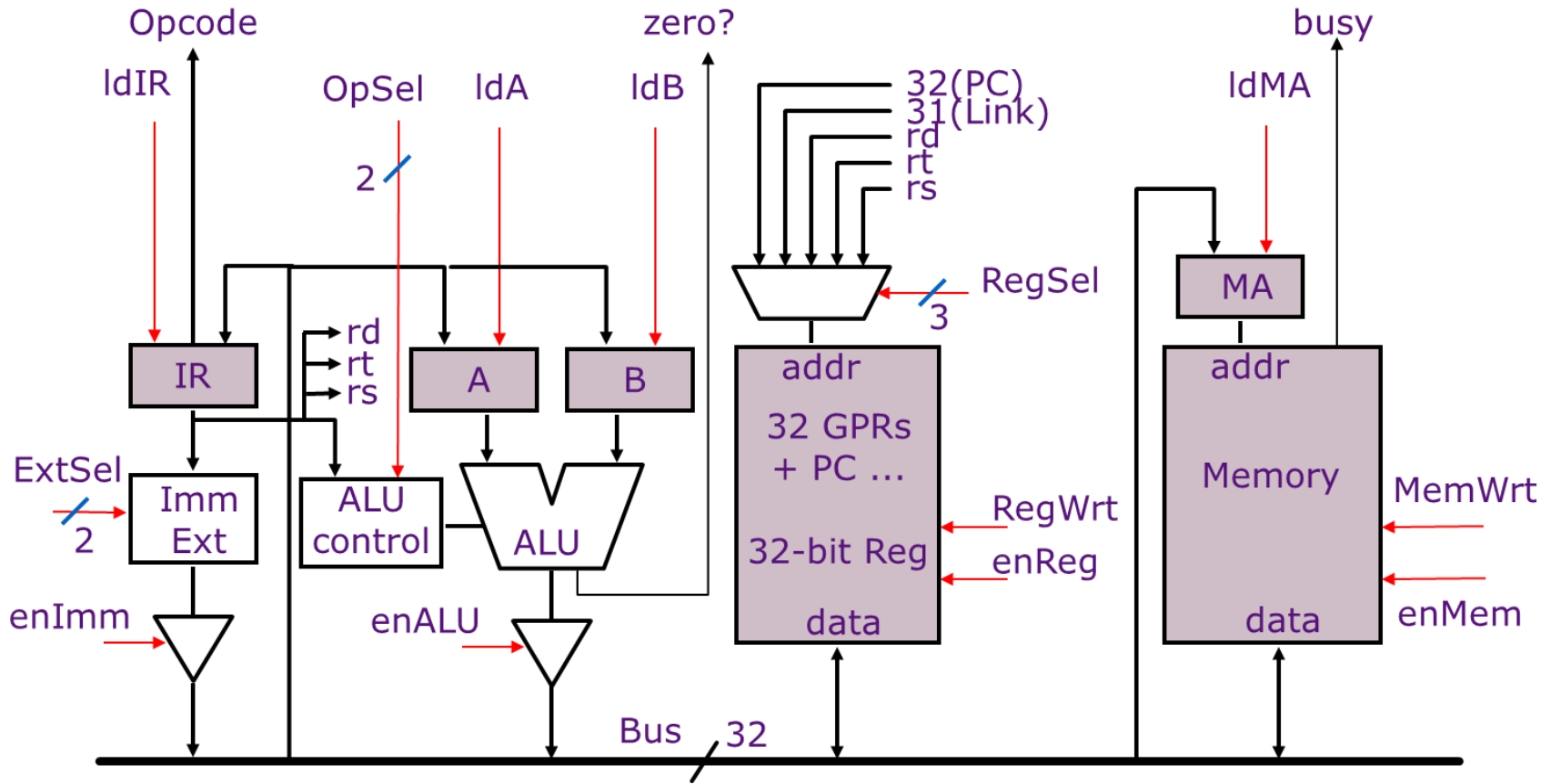
Question 3 (15 points):

Ben is so proud of his EXD instruction that he decides to implement it in MIPS using microprogramming, extending EXD to include a register field `rs` and then executing the contents of `rs`.

First, write register transfer language (i.e. pseudocode like: $A \leftarrow PC$) for Ben's microcoded MIPS implementation of EXD:

Fill in the sheet on the next page with the microcode for EXD. Use don't cares (*) for fields where it is safe to use don't cares. The solution should be elegant and efficient (fewest number of new states needed and hardware added). In order to further simplify this problem, ignore the busy signal and assume that the memory is as fast as the register file. You should try to optimize your implementation for minimum number of cycles necessary and for maximum number of don't-care signals.

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 in the margins as long as you do it neatly. Make sure that your microcode fetches the next instruction.

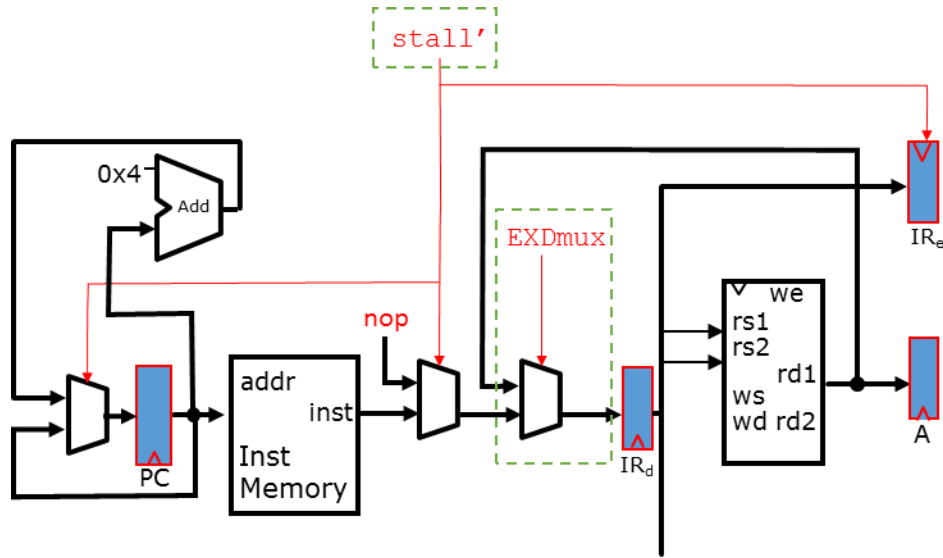


Bus-based MIPS architecture for microcoding.

State	Pseudocode	ld IR	Reg sel	Reg W	En Reg	ld A	ld B	ALU	En ALU	ld MA	Mem W	En Mem	Ex Sel	En Imm	μ Br	Jump target
Fetch0	MA \leftarrow PC A \leftarrow PC	*	PC	0	1	1	*	*	0	1	*	0	*	0	Next	*
	IR \leftarrow Mem	1	*	*	0	0	*	*	0	*	0	1	*	0	Next	*
	PC \leftarrow A+4 B \leftarrow A+4	0	PC	1	1	*	1	A+4	1	*	*	0	*	0	Dis- patch	*
Nop0:	-	*	*	*	0	*	*	*	0	*	*	0	*	0	Jump	Fetch0
EXD:																

Question 4 (10 points):

Grateful for your help, Ben was nonetheless unhappy with the performance of the microcode. So he decided to implement a pipelined version of EXD. Ben realized that the EXD instruction was in and of itself a control hazard. Help Ben safeguard his pipeline. The diagram below shows the front end of the five-stage pipeline we used in class. A new datapath and mux have been added to move `rd1` into the instruction register of the decode stage.



Your task is to write the new stall signal (`stall'`) and fill in the missing signal, `EXDmux`. Write your signal in terms of signals (e.g., `PC` or `rd1` or `IRD`) and feel free to use the old stall signal (`stall`).

`stall'` = _____

`EXDmux` = _____

Part B: Write Effective Address Extensions (35 points)

You've noticed that many programs execute code similar to the following during loops:

```
LD R1, 4(R2)
ADD R2, R2, 4
```

Or:

```
ST R1, 4(R2)
ADD R2, R2, 4
```

You want to optimize your architecture for this common case. You are going to do so by adding “write effective address” variants of the load and store instructions, LDWA and STWA. The semantics of these instructions are that they will perform the normal memory operation (LD or ST) and then write the effective address in the register that indexed into memory (*not* the register whose contents are read/written to memory). Specifically these instructions do the following:

```
LDWA rs, rt, Imm:
    rs ← Memory[(rt) + Imm]
    rt ← (rt) + Imm
```

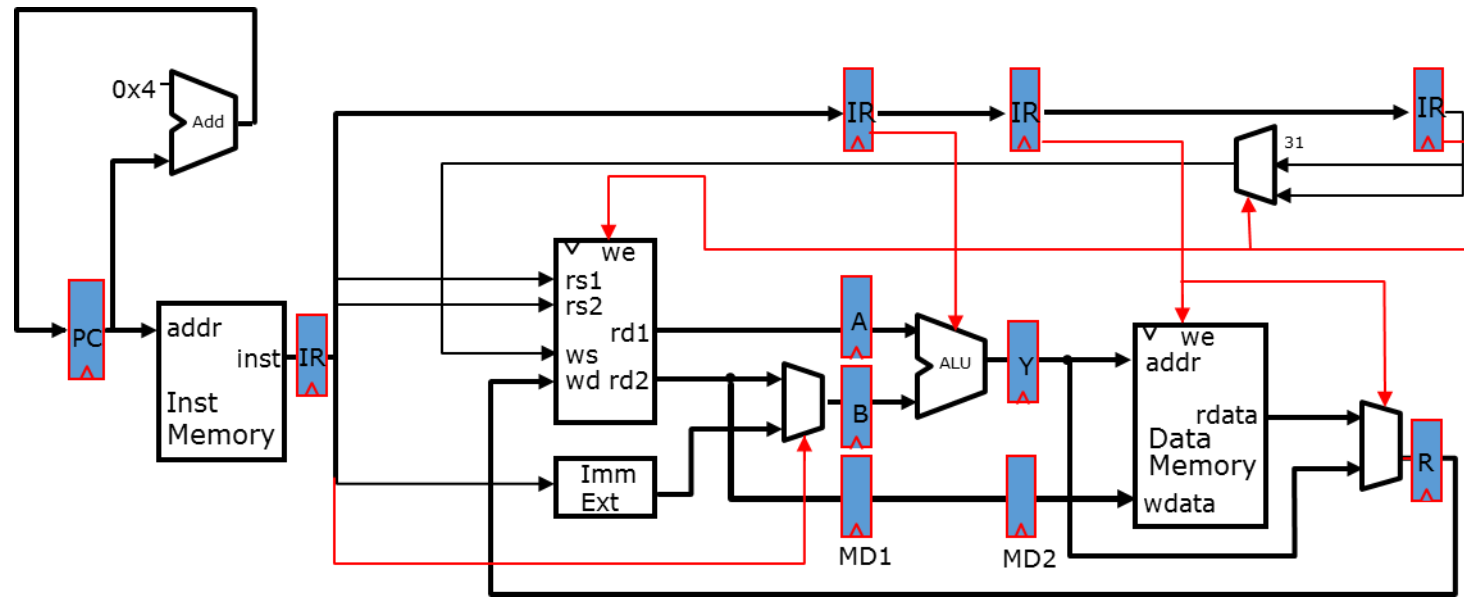
```
STWA rs, rt, Imm:
    Memory[(rt) + Imm] ← (rs)
    rt ← (rt) + Imm
```

These extensions allow us to rewrite the previous examples as:

```
LDWA R1, R2, 4
```

And:

```
STWA R1, R2, 4
```

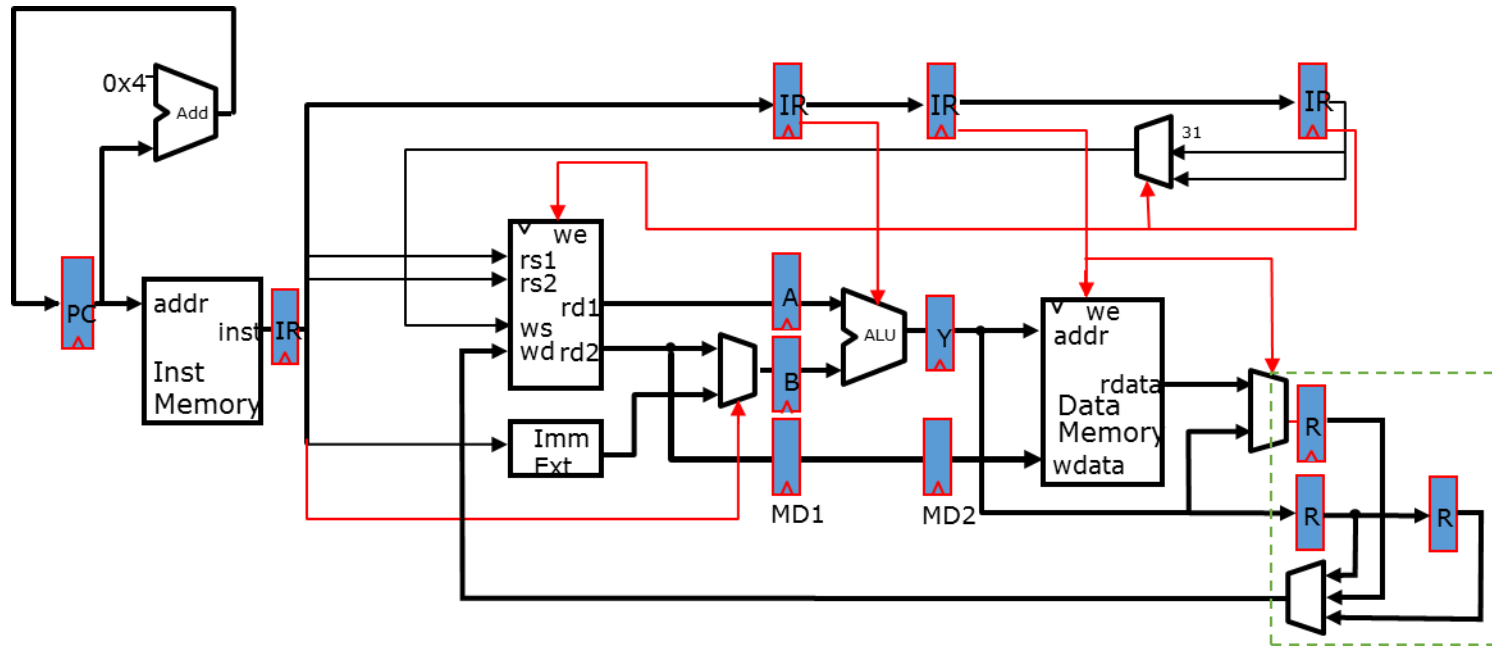



Question 1 (10 points):

You start with implementing STWA. For the following sequence of instructions and the standard five-stage pipeline (shown above), indicate how each instruction will flow through the pipeline on the following page. Assume full bypassing and stall logic are implemented for your architecture. Use arrows to indicate forwarding and dashes for stalls, as illustrated.

Name _____

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12
LD R1, 0(R2)	F	D	E	M	W								
ADD R3, R1, 10		F	D	-	E	M	W						
LD R4, 0(R3)													
STWA, R4, R1, 4													
STWA R4, R1, 4													
ADD R2, R1, R3													



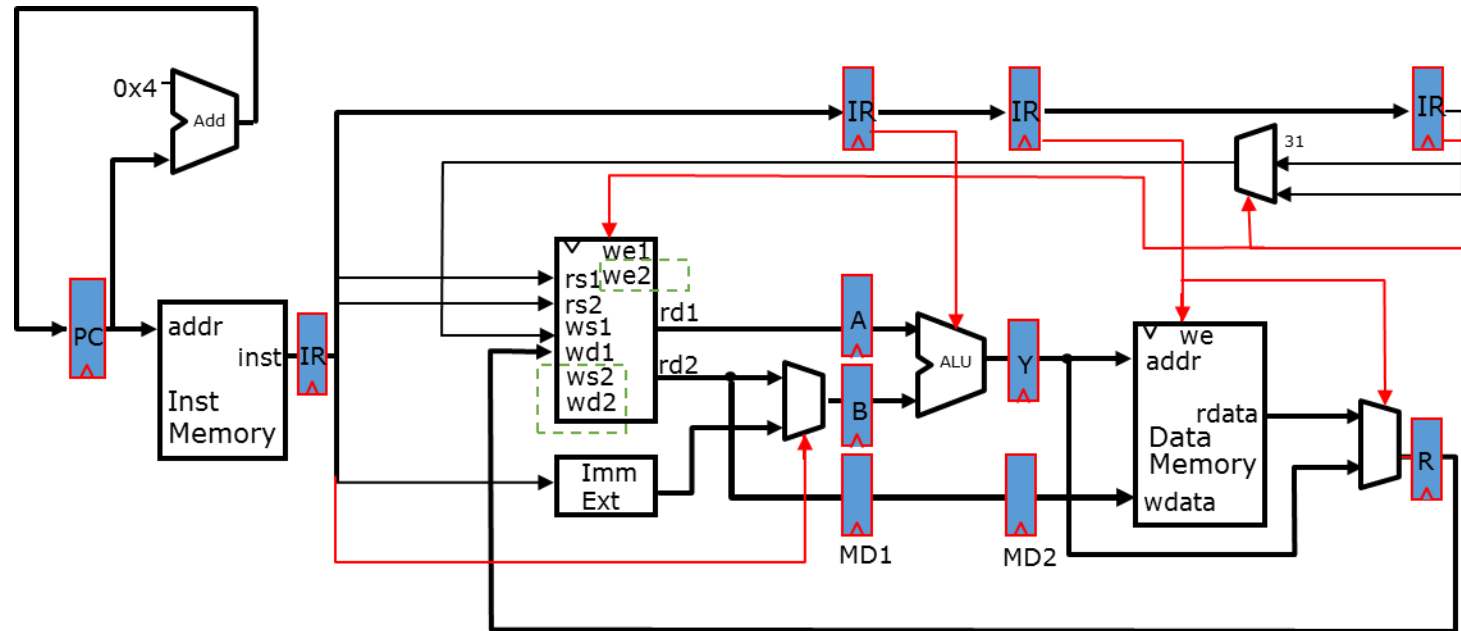
Question 2 (5 points):

You next want to implement LDWA, and quickly realize that LDWA runs into a structural hazard on the register file. You decide to fix this by adding an extra writeback stage (W2) to your pipelined design as shown above. In one or two sentences, explain what the hazard is and why the additional stage fixes it (assume correct stall logic).

Question 3 (10 points):

Assume that the six-stage design above has full bypassing and correct stall logic. Fill in the pipeline for the instructions given below, using arrows and dashes as before.

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12
LD R1, 0(R2)	F	D	E	M	W1	W2							
ADD R3, R1, 10		F	D	-	E	M	W1	W2					
LDWA R5, R3, 0													
ADD R1, R3, R4													
LDWA R5, R3, 0													
ADD R1, R5, R0													



Question 4 (5 points):

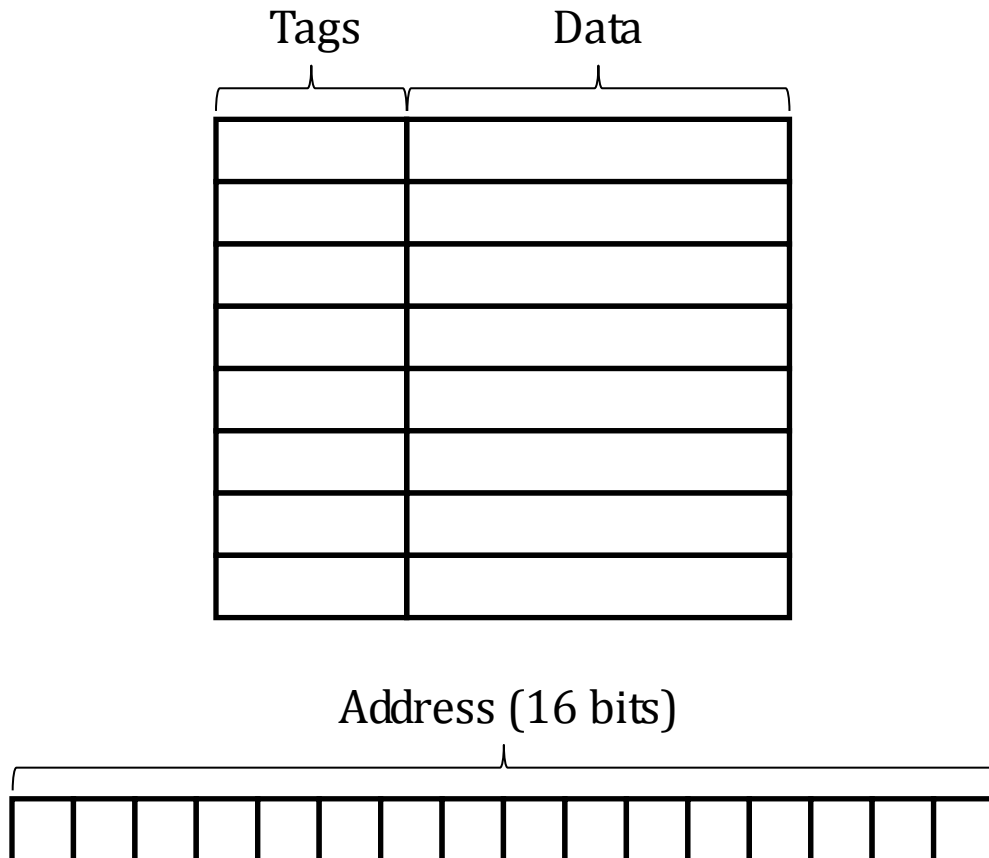
Adding a second writeback stage is only one way to fix this structural hazard. An alternative design is to add a second write port to the register file. Quickly sketch the datapath for this design in the diagram above. You do *not* need to write the stall logic. (Additional signals are: we2, ws2, wd2.)

Question 5 (5 points):

In one or two sentences, explain the tradeoffs between adding an additional pipeline stage vs. adding a write port to the register file. What conditions might favor one or the other design?

Part C: Caches (25 points)

Your processor has an 8-line level 1 data cache as illustrated below. Suppose that cache lines are 32 bytes (256 bits) and memory addresses are 16 bits, with byte-addressable memory. The cache is indexed by low bits without hashing.



Question 1 (10 points):

We're first going to fill in the above diagram with more detail.

Divide the bits of the address according to how they are used to access the cache (tag, index, offset).

What exactly is contained in the cache tags? (Include all bits necessary for correct operation of the cache as discussed in lecture.)

How many bits in total are needed to implement the level 1 data cache?

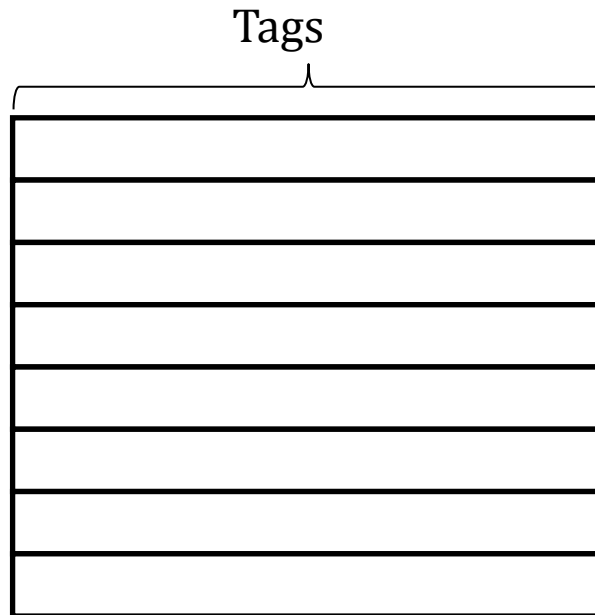
Question 2 (5 points):

Suppose the processor accesses the following data addresses starting with an empty cache:

```

0x0028: 0000 0000 0010 1000
0x102A: 0001 0000 0010 1010
0x9435: 1001 0100 0011 0101
0xEFF4: 1110 1111 1111 0100
0xBEEF: 1011 1110 1110 1111
0x4359: 0100 0011 0101 1001
0x01DE: 0000 0001 1101 1110
0x8075: 1000 0000 0111 0101
0x9427: 1001 0100 0010 0111
    
```

What would the level 1 data cache tags look like after this sequence? How many hits would there be in the level 1 data cache? (*Don't worry about filling in the Data column – we didn't give you the data!*)



Question 3 (10 points):

Suppose that the level 1 data cache has a hit rate of 40% on your application, an access time of a single cycle, and a miss penalty to memory of forty cycles. What is the average memory access time?

You aren't happy with your memory performance, so you decide to add a level two cache. Suppose the level two cache has a hit rate of 50%. What access time must the level two cache have for this to be a good design (ie, reduce AMAT)?