Discussion 7

Pipelining, hazards and FPGA

Pipeline hazards

- Structural Hazard
  - An instruction in the pipeline may need a resource being used by another instruction in the pipeline
  - Solution: Stalling newer instruction, or adding more hardware
- Data Hazard:
  - An instruction may depend on a data value produced by an earlier instruction
- Control Hazard (branches, exceptions)
  - An instruction may depend on the next instruction’s address

Structural hazard

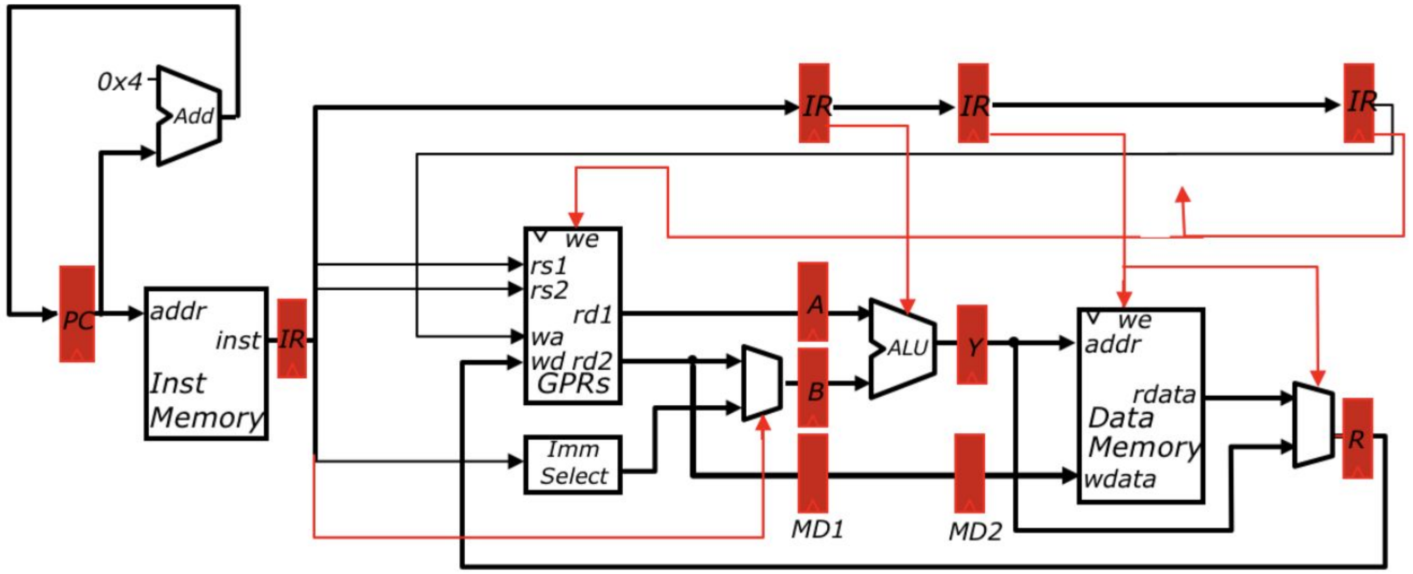
- Memory access
    - Register files have multiple ports (2 read, 1 write)
    - Separate instruction memory and data memory

Data Hazard

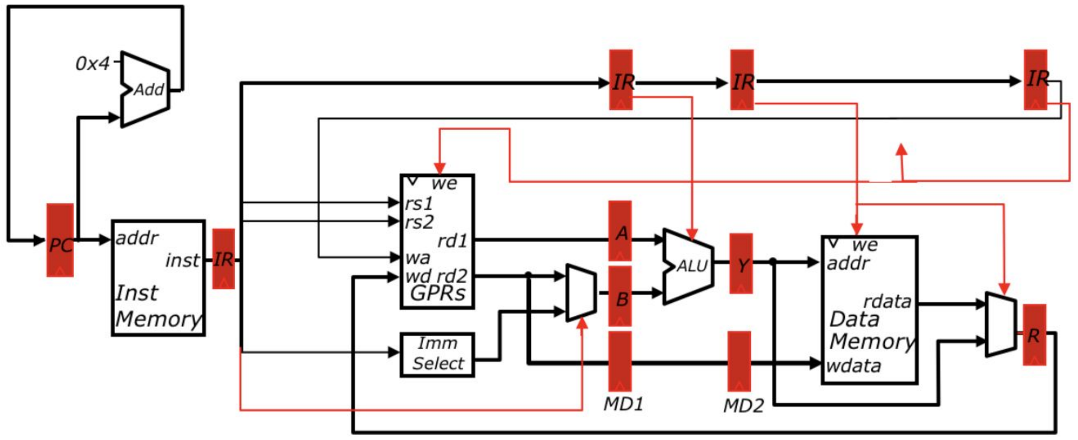
x1 <- x0 + 10

x4 <- x1 + 3

x5 <- x1 + 5

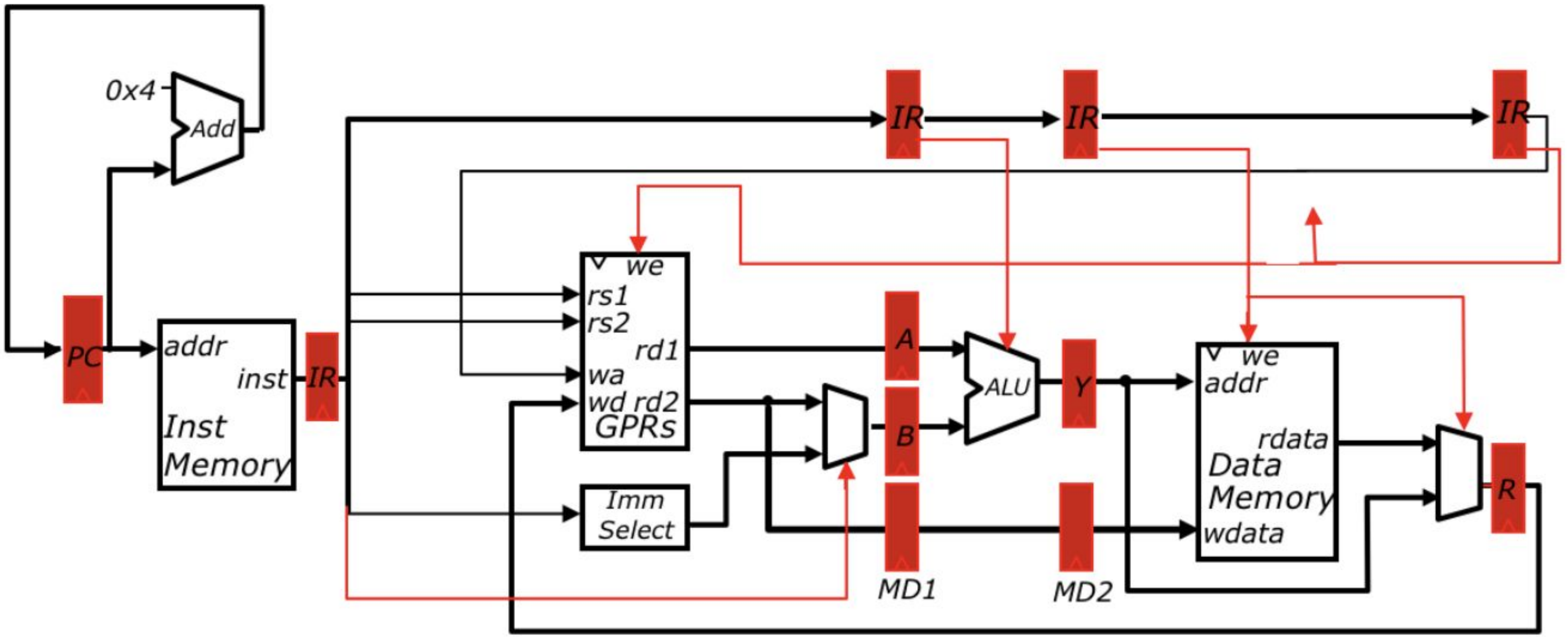


Data Hazard

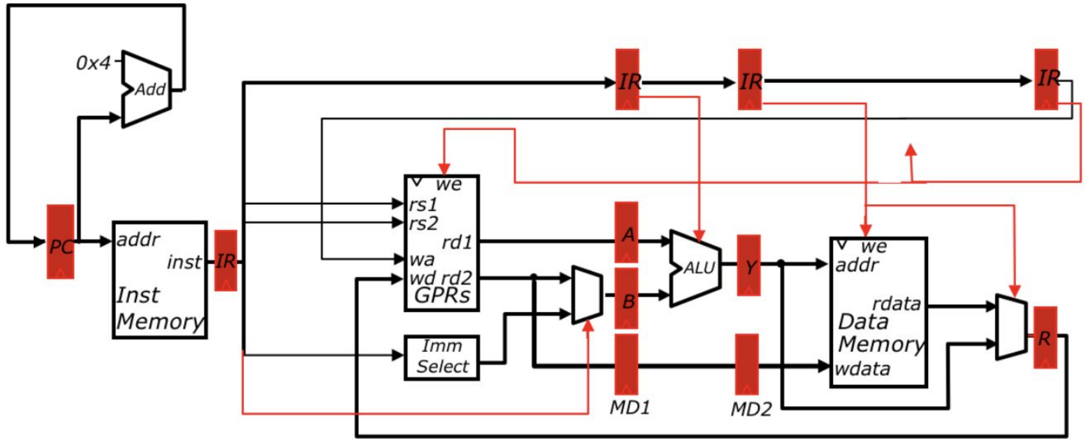


Time	x1 <- x0 + 10	x4 <- x1 + 3	x5 <- x1 + 5
t0	IF		
t1	ID	IF	
t2	EX	ID	IF
t3	MA	?	ID
t4	WB		?
t5			
t6			
t7			

Data Hazard: Stalling



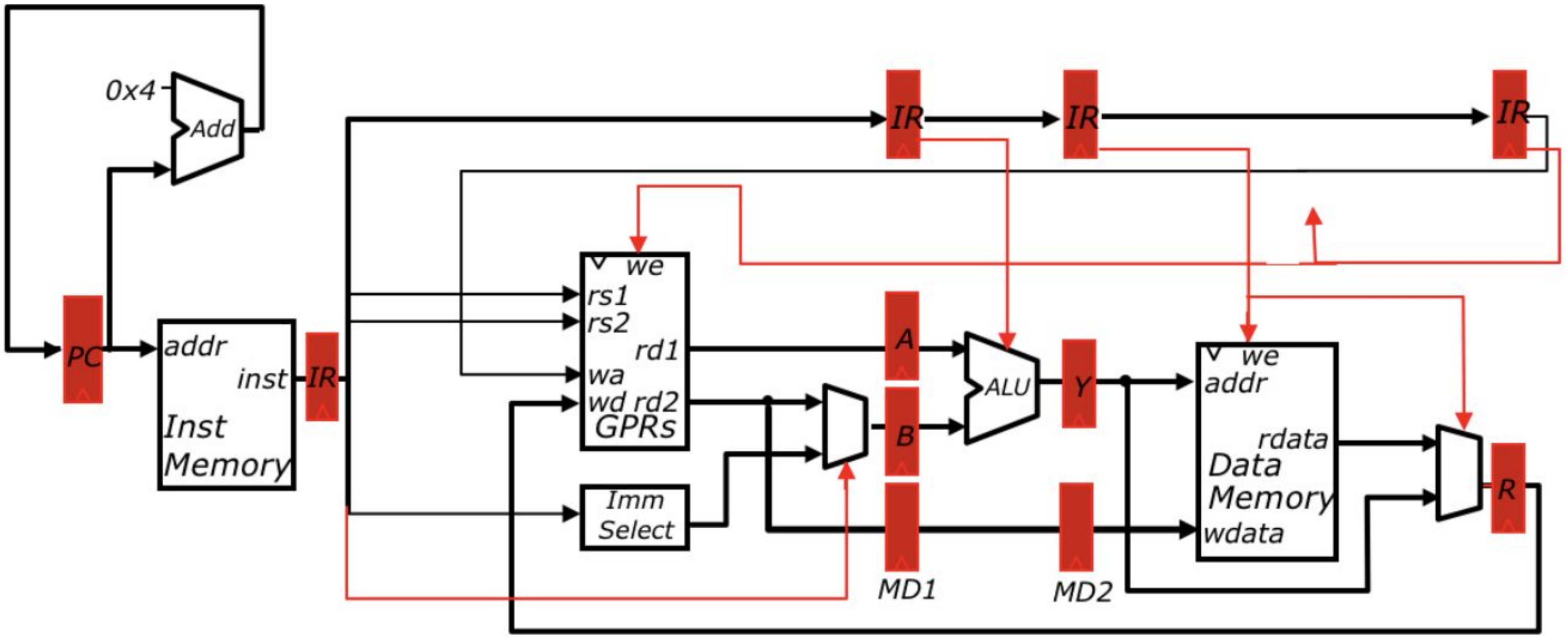
Data hazard: stalling



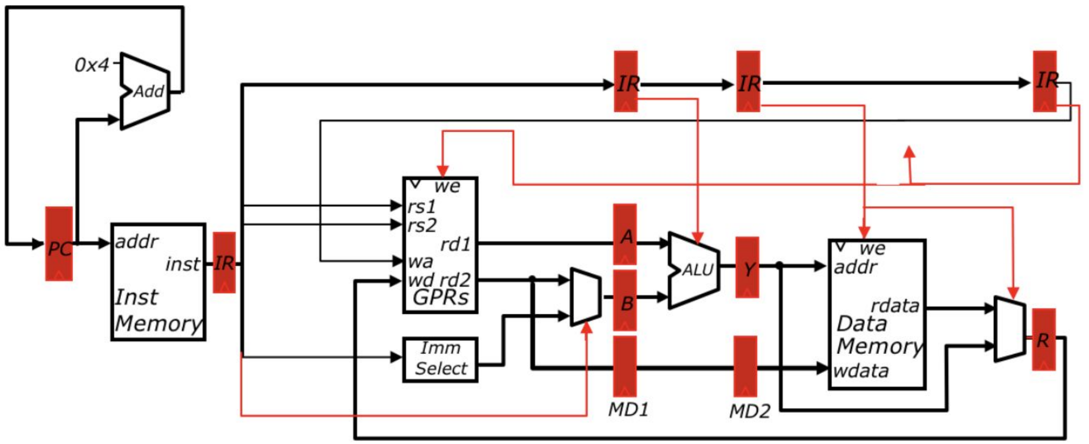
Assume register file can be written and read at the same time

Time	x1 <- x0 + 10	x4 <- x1 + 3	x5 <- x1 + 5
t0	IF		
t1	ID	IF	
t2	EX	ID	IF
t3	MA	ID	IF
t4	WB	ID	IF
t5		EX	ID
t6		MA	EX
t7		WB	MA

Data Hazard: forwarding

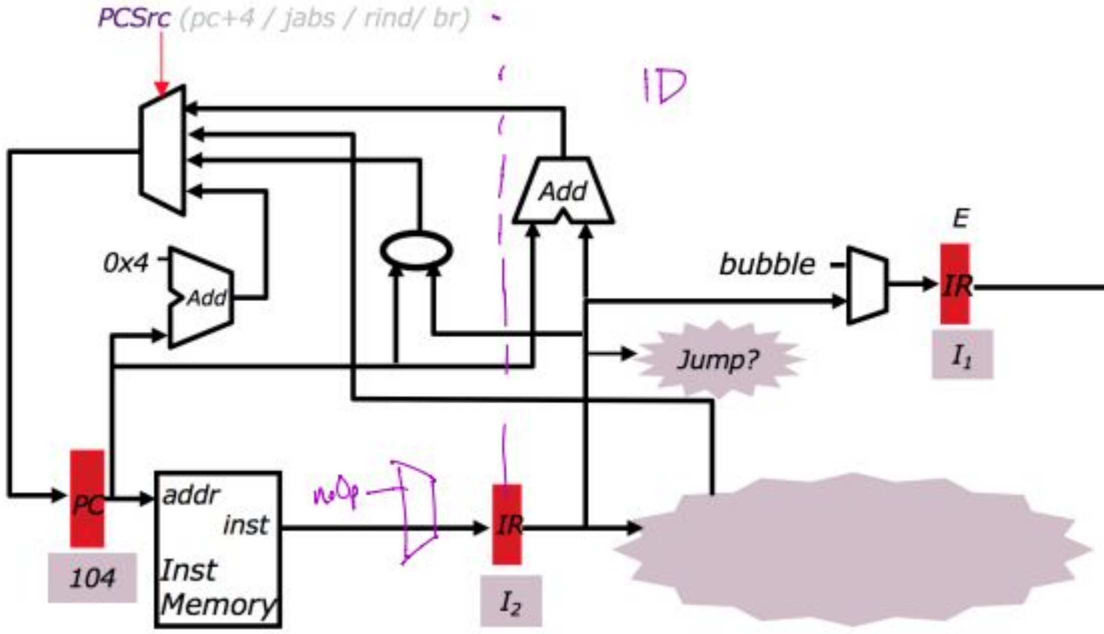


Data Hazard: forwarding



Time	x1 <- x0 + 10	x4 <- x1 + 3	x5 <- x1 + 5
t0	IF		
t1	ID	IF	
t2	EX	ID	IF
t3	MA	EX	ID
t4	WB	MA	EX
t5		WB	MA
t6			WB
t7			

Control Hazard: jump

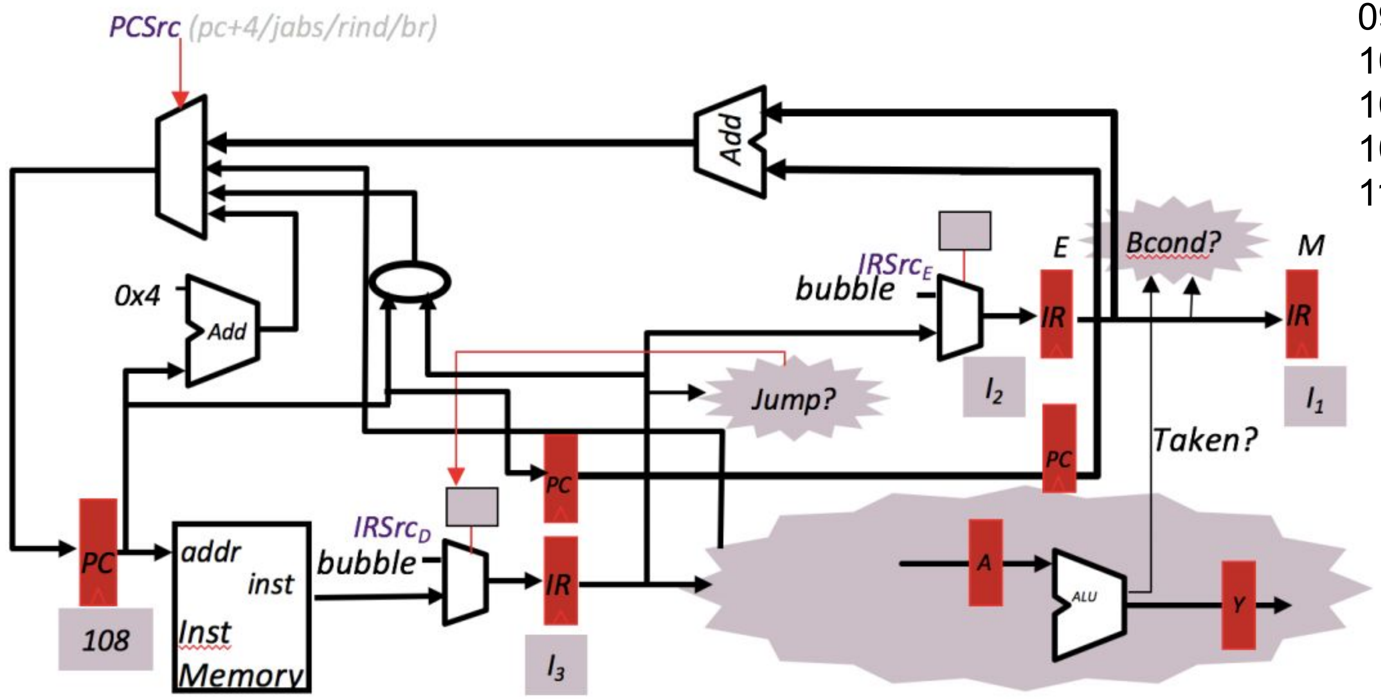


Example

096	ADD
100	J 304
104	ADD
……	
304	ADD

Automatically fetched into pipeline, need to kill it

Control Hazard: conditional branch



Example

096	ADD
100	BEQ x1, x2, 200
104	ADD
108	ADD
112	ADD

Example 1

1. Draw the datapath that would eliminate data hazards for a 3-stage pipeline

Example 1

1. Draw the datapath that would eliminate data hazards for a 3-stage pipeline

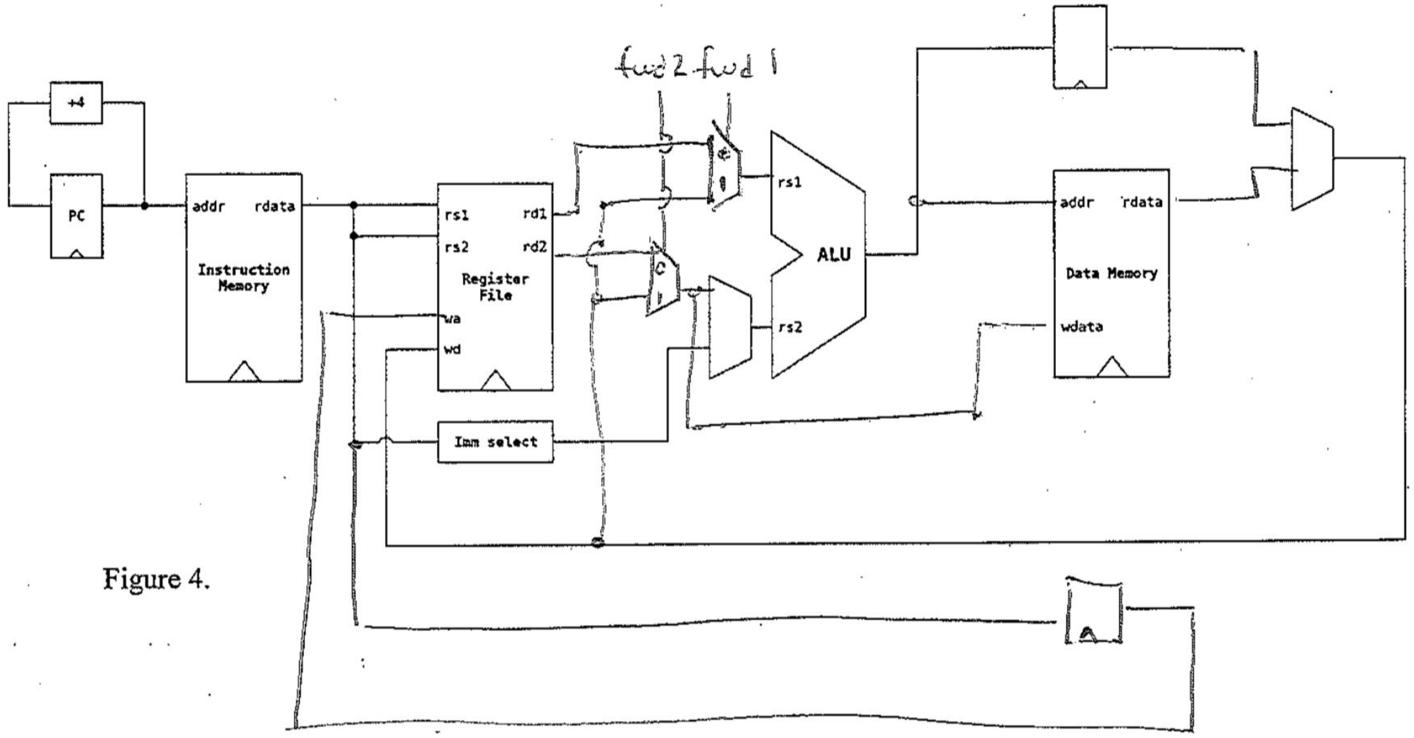


Figure 4.

Example 1

2. Write verilog code to describe the control logic you drew above

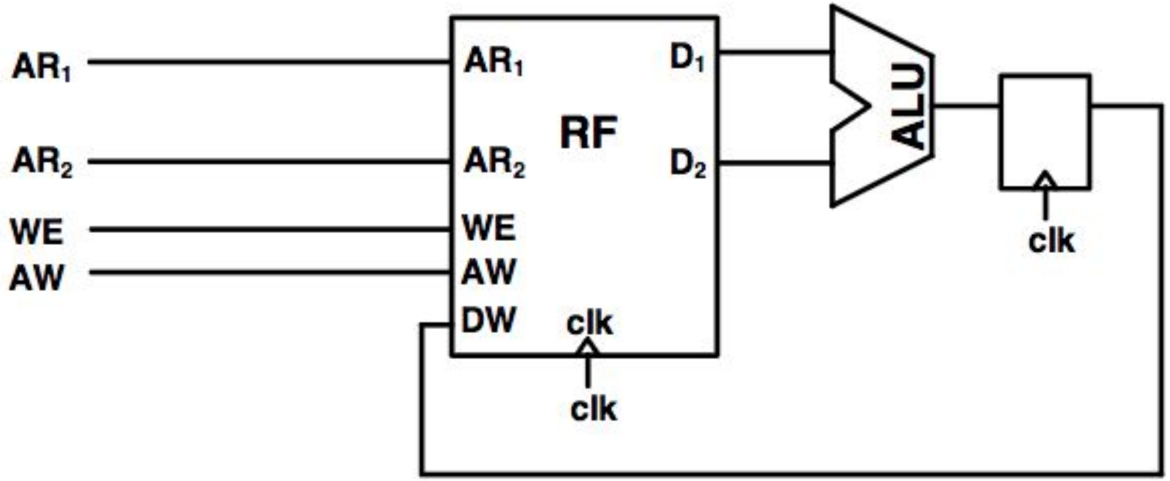
Example 1

2. Write verilog code to describe the control logic you drew above

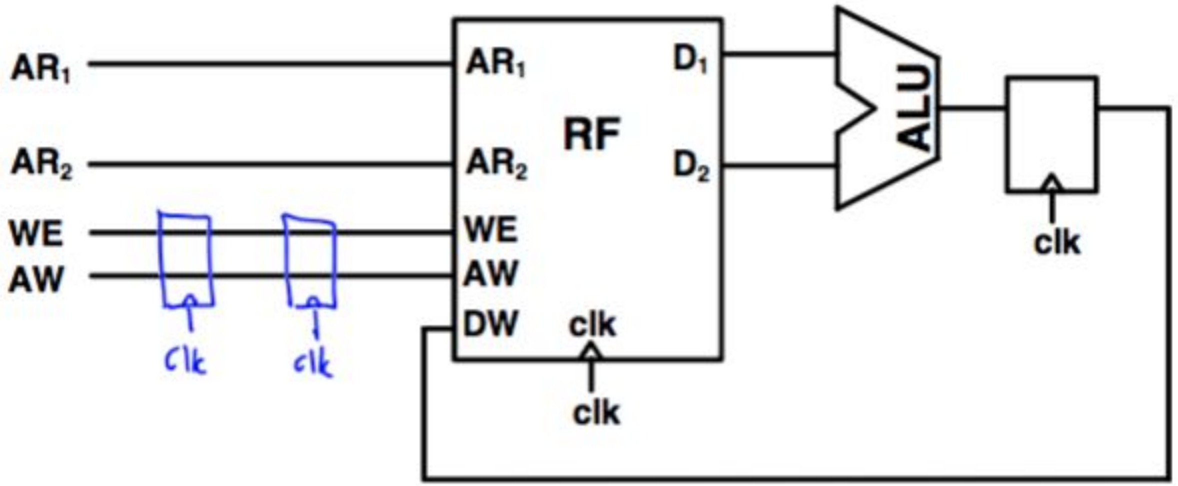
```
assign fwd1=(wa == rs1) && (wa != 0)

assign fwd2=(wa==rs2) && (wa != 0)
```

Example 2

Shown below is a portion of a prototype design for a pipelined CPU's datapath that uses a register file with **synchronous reads and writes**. However, even ignoring any potential data hazards, this design does not function corretly - in particular, register type instructions. Explain what the error is caused by and add any extra components necessary to correct the design.

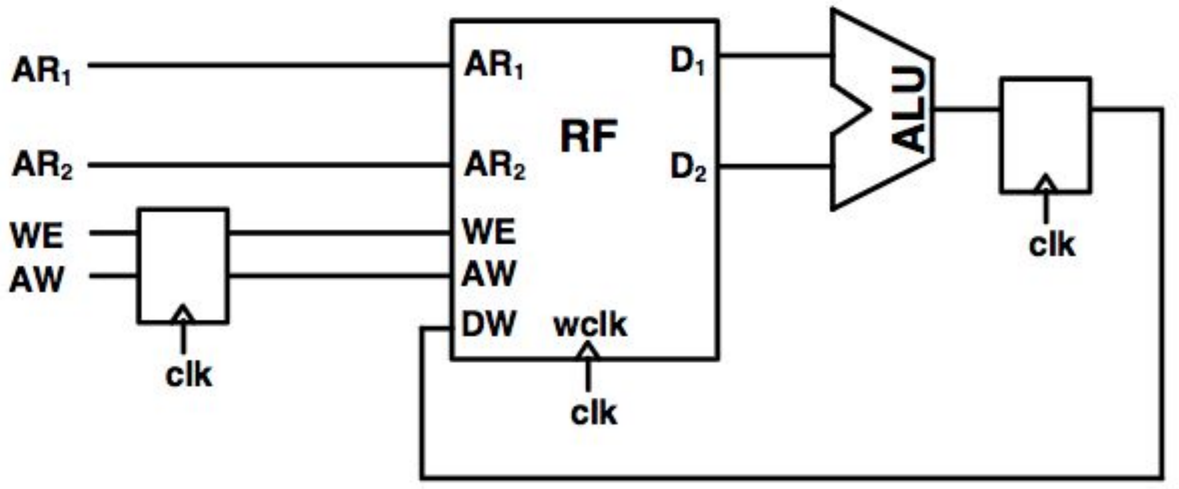


Example 2

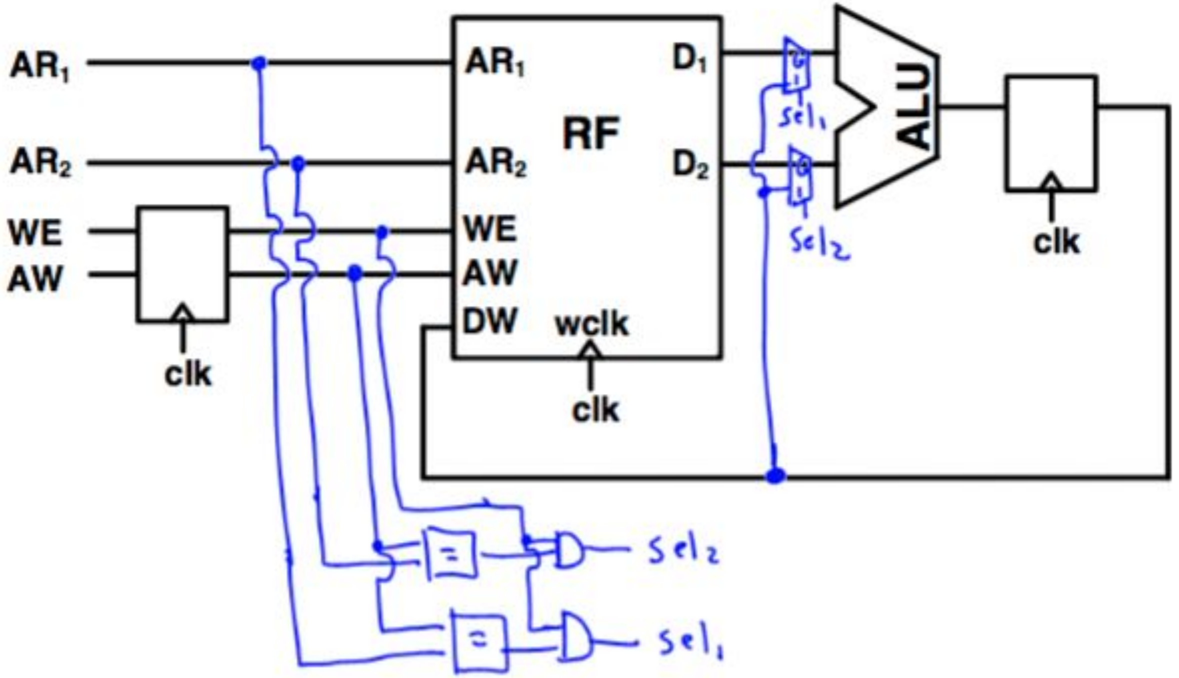


Example 2

Now this register file has synchronous writes with asynchronous reads. Add appropriate forwarding to eliminate all data hazards.



Example 2


AR1
AR2
WE
AW
clk
AR1
AR2
WE
AW
DW
wclk
clk
RF
D1
D2
sel1
sel2
ALU
clk
sel2
sel1
=
=
0
1
0
1

LUT

- LUT is short for “Look Up Table”
- The number of rows in the table is 2^N where N = number of input bits
- There is 1 row for every possible input combination
    - If you view the inputs as a single multiple bit wide wire, you can think of it as specifying an *address* in the LUT
- The designer determines what the output will be for each row of the table

Implementing functions with LUTs

- You can view the entries of an N-input LUT as being entries in a truth table for an N-input combinational logic block
- Since the LUT contains a row for every possible combination of inputs, we can implement any combination function by specifying the output values for each row in the table.

What function is this?

- This outputs 1 only when exactly 1 of A, B and C are true
- A&(~B)&(~C) | (~A)&B&(~C) | (~A)&(~B)&C

	C	B	A	Out
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

Building larger LUTs

- The bottom half of the table looks like a repeat of the top half of the table except
  - The top half of the table is when d is 0, the bottom half is when d is 1
  - The top and bottom halves of the table have different outputs.

d	c	b	a	out
0	0	0	0	o_1
0	0	0	1	o_2
0	0	1	0	o_3
0	0	1	1	o_4
0	1	0	0	o_5
0	1	0	1	o_6
0	1	1	0	o_7
0	1	1	1	o_8
1	0	0	0	o_9
1	0	0	1	o_{10}
1	0	1	0	o_{11}
1	0	1	1	o_{12}
1	1	0	0	o_{13}
1	1	0	1	o_{14}
1	1	1	0	o_{15}
1	1	1	1	o_{16}

Let’s split the table

d	c	b	a	out
0	0	0	0	o_1
0	0	0	1	o_2
0	0	1	0	o_3
0	0	1	1	o_4
0	1	0	0	o_5
0	1	0	1	o_6
0	1	1	0	o_7
0	1	1	1	o_8
1	0	0	0	o_9
1	0	0	1	o_{10}
1	0	1	0	o_{11}
1	0	1	1	o_{12}
1	1	0	0	o_{13}
1	1	0	1	o_{14}
1	1	1	0	o_{15}
1	1	1	1	o_{16}

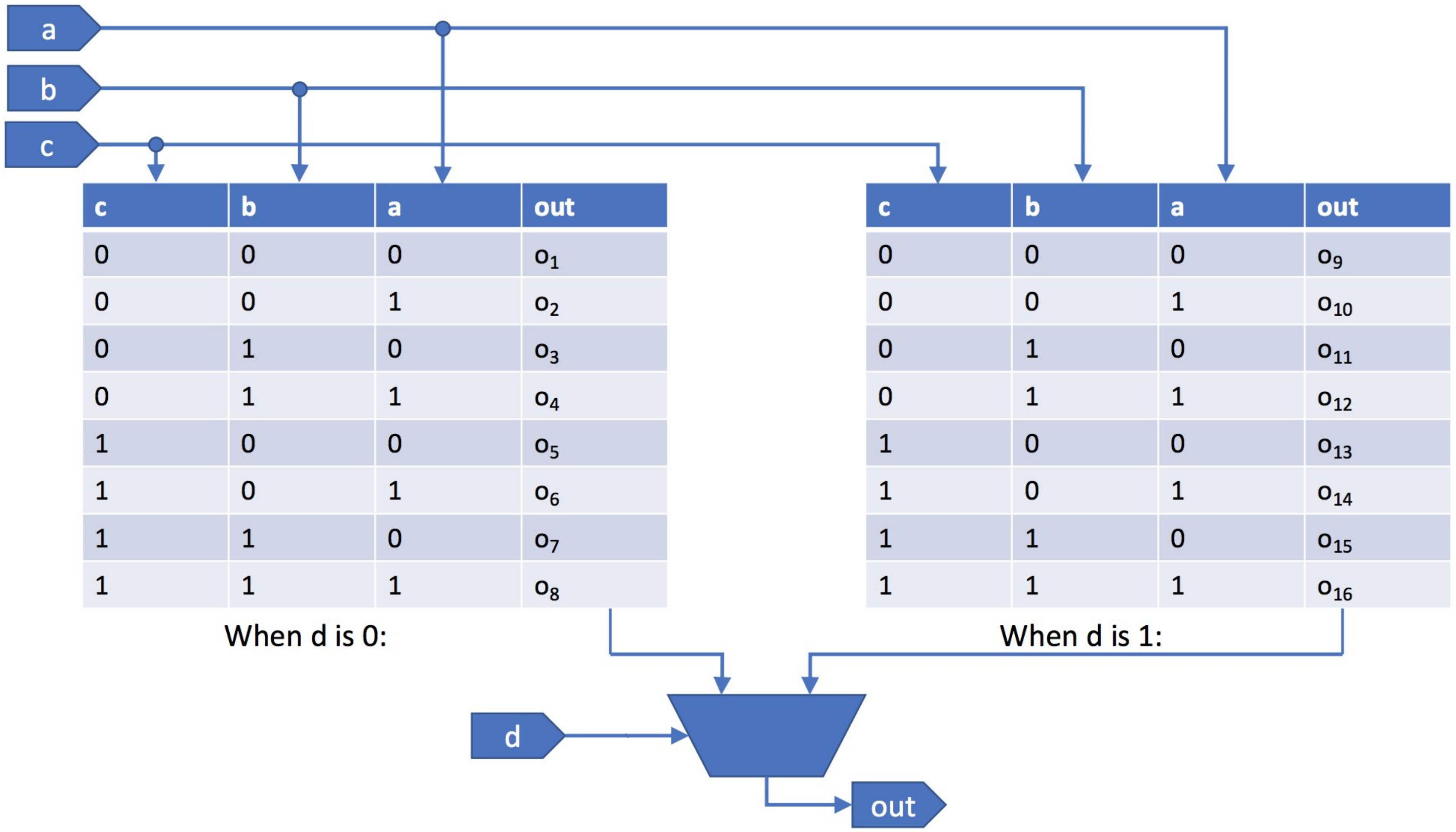
When d is 0:

c	b	a	out
0	0	0	o_1
0	0	1	o_2
0	1	0	o_3
0	1	1	o_4
1	0	0	o_5
1	0	1	o_6
1	1	0	o_7
1	1	1	o_8

When d is 1:

c	b	a	out
0	0	0	o_9
0	0	1	o_{10}
0	1	0	o_{11}
0	1	1	o_{12}
1	0	0	o_{13}
1	0	1	o_{14}
1	1	0	o_{15}
1	1	1	o_{16}

Select which table to use



When d is 0:

c	b	a	out
0	0	0	o_1
0	0	1	o_2
0	1	0	o_3
0	1	1	o_4
1	0	0	o_5
1	0	1	o_6
1	1	0	o_7
1	1	1	o_8

When d is 1:

c	b	a	out
0	0	0	o_9
0	0	1	o_{10}
0	1	0	o_{11}
0	1	1	o_{12}
1	0	0	o_{13}
1	0	1	o_{14}
1	1	0	o_{15}
1	1	1	o_{16}