

# EECS 151/251A

## Discussion 6

Zhaokai Liu  
10/5, 10/6 and 10/11

# Agenda

- RISC-V Pipeline and Hazards
- FPGA

# Pipeline

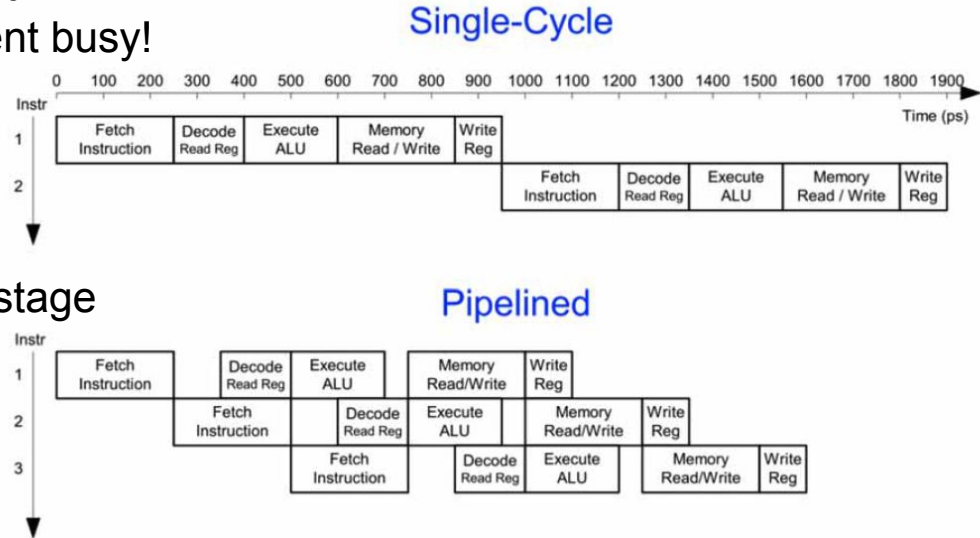
# Race Conditions: Synthesis vs. Simulation

Why pipeline?

- In digital design, if clock speed doubles, the performance doubles
- Time constraints must be satisfied
- We want to keep every component busy!

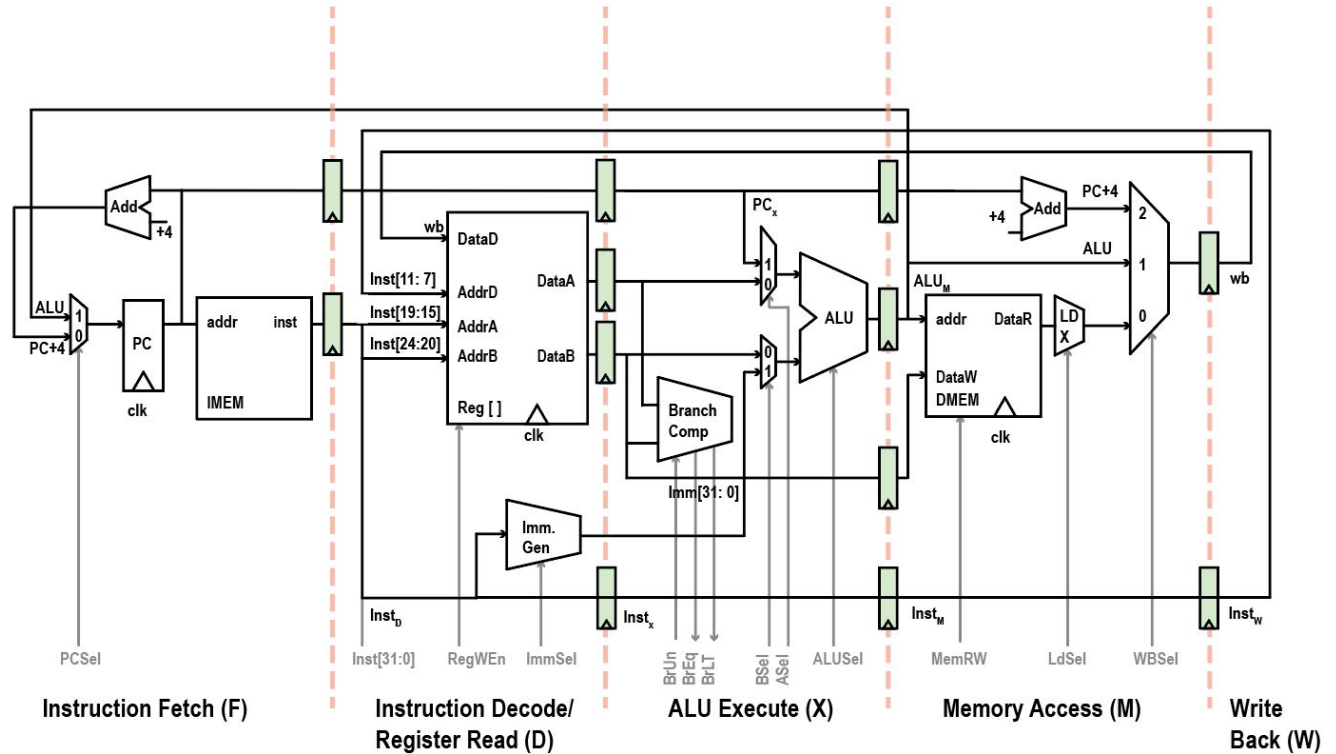
More challenges

- Design complexity
- Hazard
- clk is determined by the longest stage



# Complete datapath - pipeline

- Structural hazard
- Data hazard
- Control hazard



Hazards

# Data Hazard - Stall

Consider a 5-stage pipeline

```
add x3, x1, x2
sub x4, x1, x2
xor x5, x1, x3
or  x6, x2, x5
```

#	IF	D	EX	M	WB
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					

# Data Hazard - Forwarding

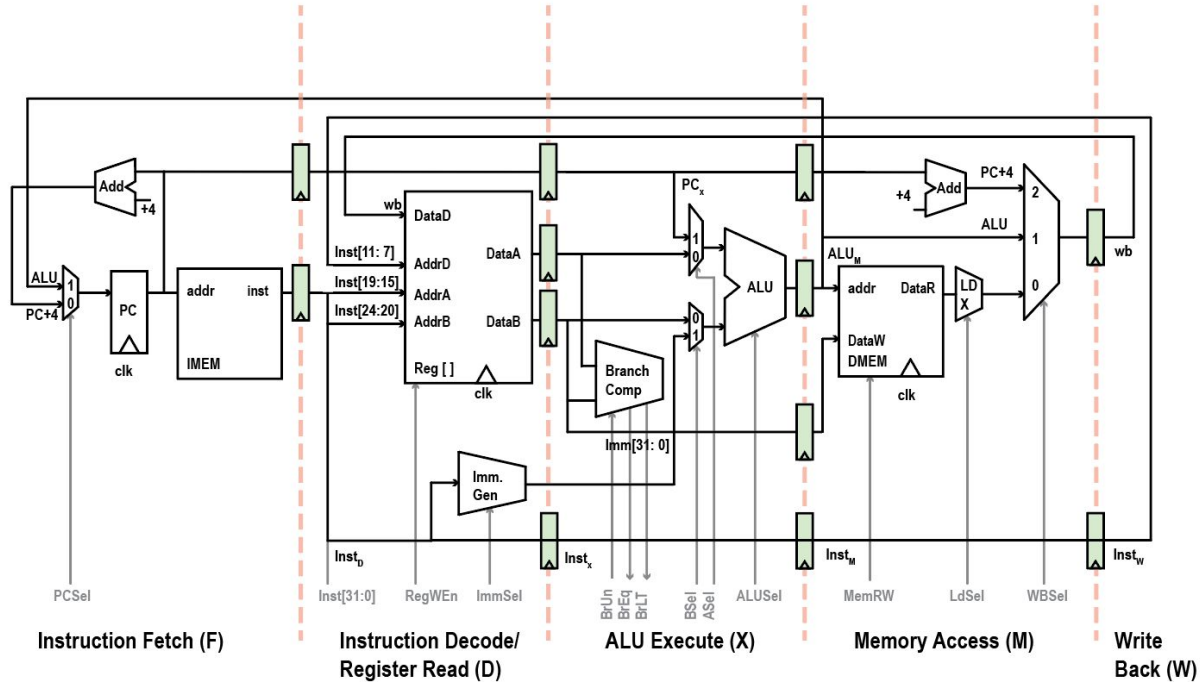
Consider a 5-stage pipeline

```
add x3, x1, x2  
sub x5, x4, x3
```



# Control Hazard

For this datapath, how many **extra** cycles are taken if a wrong prediction is made?



# Control Hazard

Consider a 5-stage pipeline

- Branches are not taken by default
  - `x1=x2`

```
beq x1, x2, imm
```

```
add x3, x1, x2
```

```
sub x4, x1, x2
```

```
xor x5, x1, x2
```

```
or x6, x1, x2
```

```
...
```

```
imm: and x3, x1, x2
```

```
nop
```

#	IF	D	EX	M	WB
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

# Control Hazard

Consider a 5-stage pipeline

- Branches are not taken by default
  - w/ forwarding hardware
  - **x1=x2**

```
beq x1, x2, imm  
add x3, x1, x2  
sub x4, x1, x2  
xor x5, x1, x2  
or  x6, x1, x2  
...
```

```
imm:  and x3, x1, x2  
      nop
```

#	IF	D	EX	M	WB
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

# Control Hazard - branch prediction

Base on last choice is a simple but useful strategy  
Consider the following code:

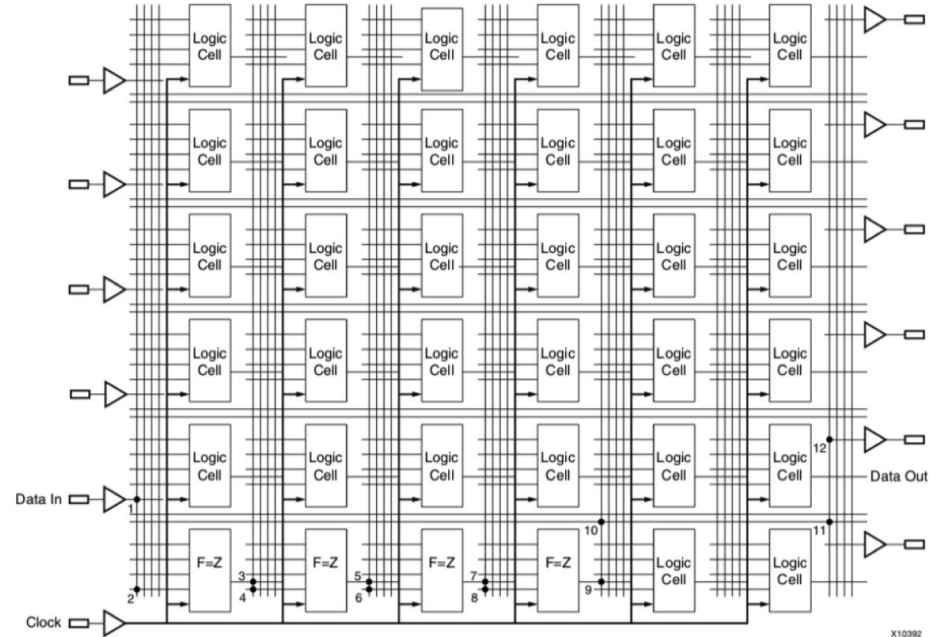
```
addi x1, x0, 0
addi x2, x0, 1
addi x10, x0, 101
add x1, x1, x2
addi x2, x2, 1
blt x2, x10, -8
nop
```

```
// equivalent to
s = 0;
for (i=1; i<101; i++){
    s += i;
}
```

FPGA

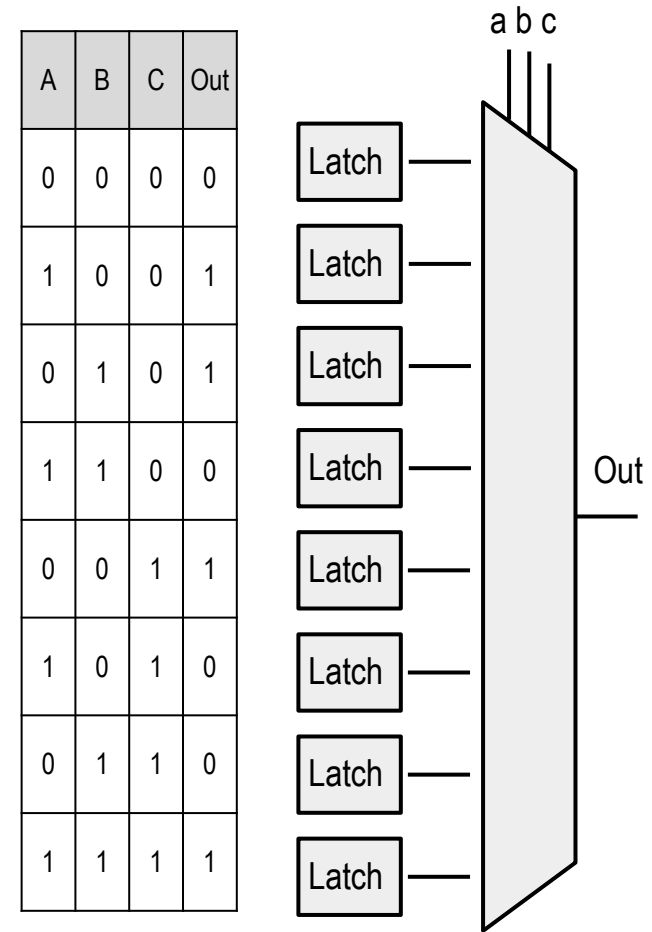
# FPGA Structure

- Array of "Logic Cells" and interconnect
- What are "Logic Cells" exactly?
  - How to implement every possible logic function in finite space?
  - How to adapt to any N-bit wide input?



# Implementing Functions with LUTs

- Like a hardware truth table
- Map each input to corresponding output
- Easy to implement
  - Use mux with programmable latches on each input
  - Program latch to correspond to expected output
  - Select output with inputs to LUT, timing is independent of function



# What function is this?

A	B	C	Out
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	0
0	0	1	0
1	0	1	0
0	1	1	0
1	1	1	1

Out =

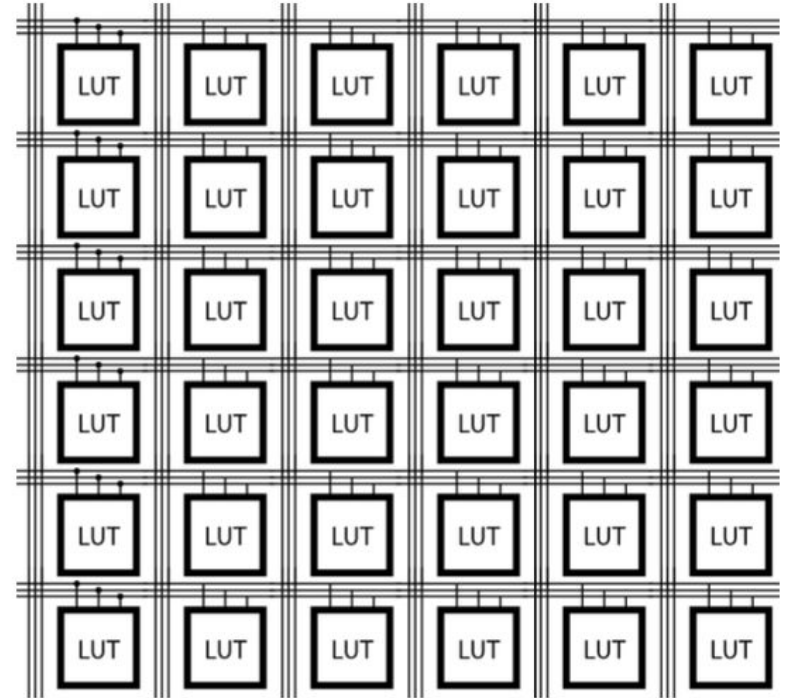
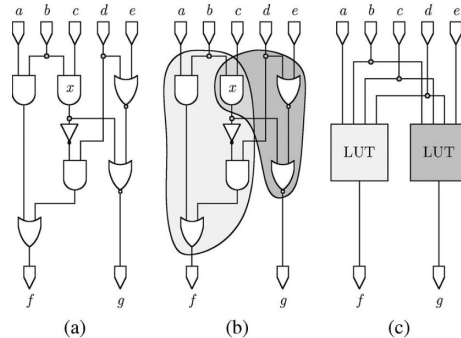
A	B	C	Out
0	0	0	0
1	0	0	1
0	1	0	1
1	1	0	0
0	0	1	0
1	0	1	1
0	1	1	0
1	1	1	0

Out =



# Implementing Functions with LUTs

- Array of LUTs and interconnect
  - Here's a proto-FPGA of 3-input LUTs
  - Can perform any combination of 3- input logic functions!
- What if we want to have a 4-input function?



# Implementing Functions with LUTs

If you can write a truth table for it (which you can with any combinational block), you can implement it with a single LUT!

- This does not necessarily mean you should implement all combinational functions with a single LUT:
- A combinational block with 64 inputs would require a LUT of  $2^{64} \approx 1.84 \times 10^{19}$  entries! Just storing all of the output bits would require 2305843 TB of data!

When would you want a 64 input combinational block? How about a 32 bit adder (32 bits for each input operand)

- There is likely a more efficient way of implementing a 64 input combinational block

# Building Larger LUTs

- With smaller LUTs
- Let's say we have 3 input LUTs, is there a way we could create a 4 input LUT?

D	C	B	A	Out
0	0	0	0	o1
0	0	0	1	o2
0	0	1	0	o3
0	0	1	1	o4
0	1	0	0	o5
0	1	0	1	o6
0	1	1	0	o7
0	1	1	1	o8
D	C	B	A	Out
1	0	0	0	o9
1	0	0	1	o10
1	0	1	0	o11
1	0	1	1	o12
1	1	0	0	o13
1	1	0	1	o14
1	1	1	0	o15
1	1	1	1	o16

# Building Larger LUTs

- Consider a 4-input LUT
  - What the function is this?
- How to build this out of 3 input LUTs?
- Notice how the LUT depends on d
  - Can split into  $d = 0$  and  $d = 1$  halves – abc inputs look identical!

d=0	D	C	B	A	Out
	0	0	0	0	0
	0	0	0	1	1
	0	0	1	0	1
	0	0	1	1	0
	0	1	0	0	1
	0	1	0	1	0
	0	1	1	0	0
	0	1	1	1	1
d=1	D	C	B	A	Out
	1	0	0	0	1
	1	0	0	1	0
	1	0	1	0	0
	1	0	1	1	1
	1	1	0	0	0
	1	1	0	1	1
	1	1	1	0	1
	1	1	1	1	0

# Building Larger LUTs

