CS 316: Pipelining Hazards

Kavita Bala
Fall 2007
Computer Science
Cornell University

Basic Pipelining

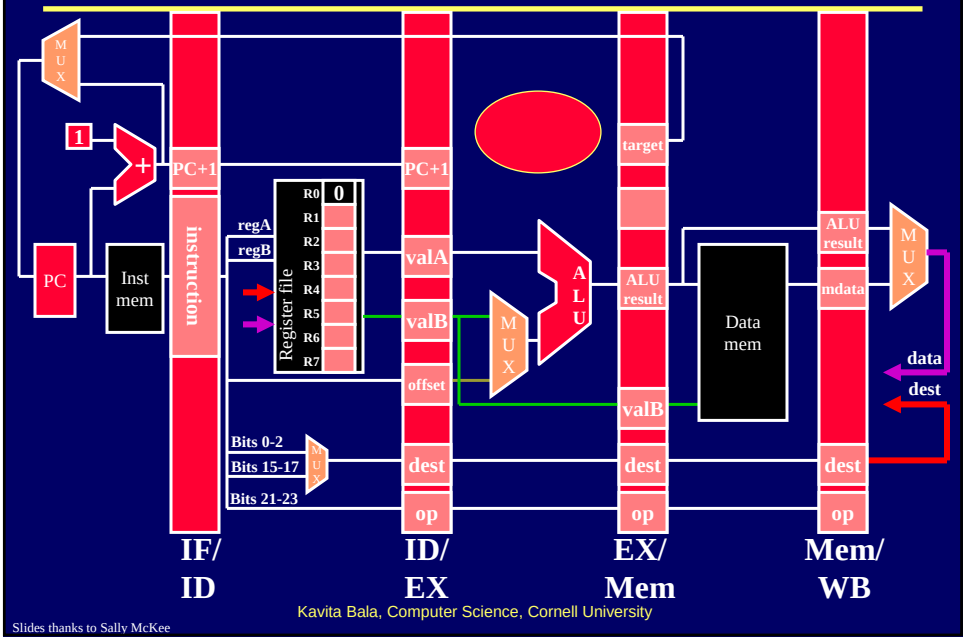
Five stage "RISC" load-store architecture

1. Instruction fetch (IF)
   - get instruction from memory
2. Instruction Decode (ID)
   - translate opcode into control signals and read regs
3. Execute (EX)
   - perform ALU operation
4. Memory (MEM)
   - Access memory if load/store
5. Writeback (WB)
   - update register file

Following slides thanks to Sally McKee

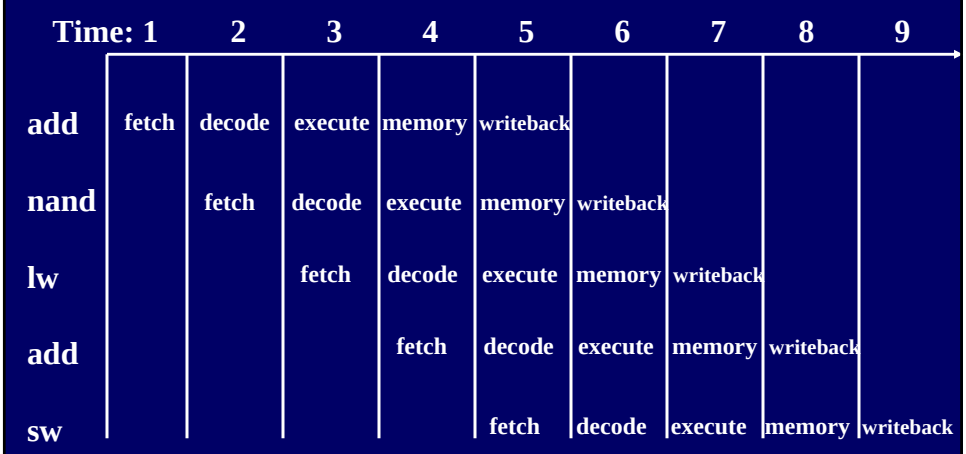
Pipelining Recap

- Powerful technique for masking latencies
  - Logically, instructions execute one at a time
  - Physically, instructions execute in parallel
    - Instruction level parallelism
- Decouples the processor model from the implementation
  - Interface vs. implementation
- BUT dependencies between instructions complicate the implementation



Slides thanks to Sally McKee

Time Graphs



Kavita Bala, Computer Science, Cornell University

Slides thanks to Sally McKee

What can go wrong?

- Structural hazards
  - Two instructions in the pipeline try to simultaneously access the same resource
- Data hazards
  - A required operand is not ready
  - Usually because a previous instruction in the pipeline has not committed it to the register file yet
- Control hazards
  - The next instruction to fetch cannot be determined
  - Usually because a jump or branch instruction has not determined the next PC yet

Kavita Bala, Computer Science, Cornell University

Slides thanks to Sally McKee

What Can Go Wrong?

- Data hazards
  - register reads occur in stage 2
  - register writes occur in stage 5
  - could read the wrong value if is about to be written
- Control hazards
  - branch instruction may change the PC in stage 4
  - what do we fetch before that?

Slides thanks to Sally McKee

Different type of Hazards

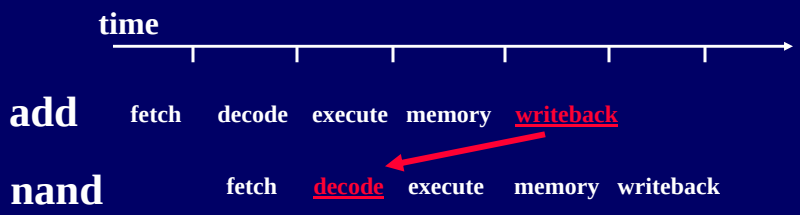
- Use register value in subsequent instruction

```
add   3 1 2
nand  5 3 4
or    6 3 1
sub   6 3 1
```


Slides thanks to Sally McKee

Data Hazards: AL ops

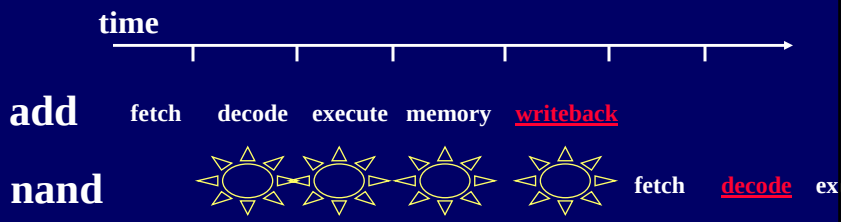
add 3 1 2
nand 5 3 4



If not careful, you read the wrong value of R3

Data Hazards: AL ops

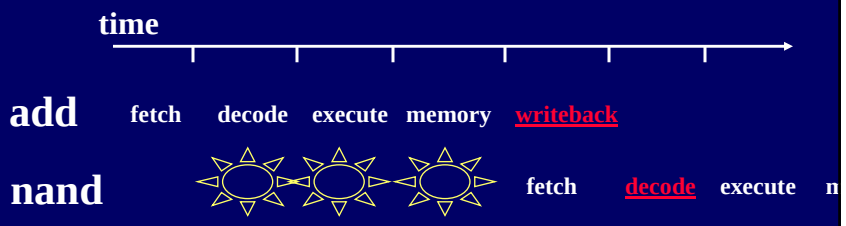
add 3 1 2
nand 5 3 4



If not careful, you read the wrong value of R3

Data Hazards: AL ops

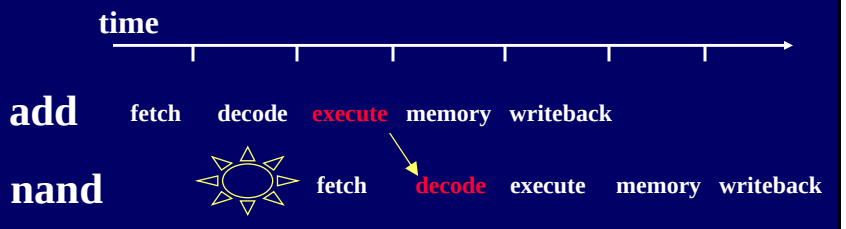
add 3 1 2
nand 5 3 4



If not careful, you read the wrong value of R3

Data Hazards: AL ops

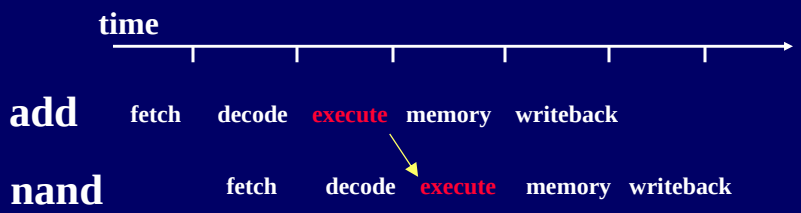
add 3 1 2
nand 5 3 4



If not careful, you read the wrong value of R3

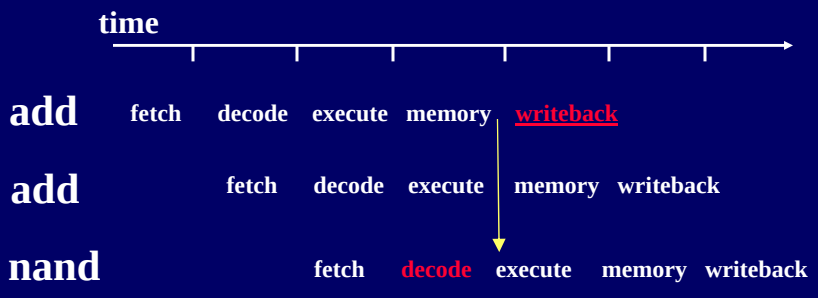
Data Hazards: AL ops

add 3 1 2
nand 5 3 4

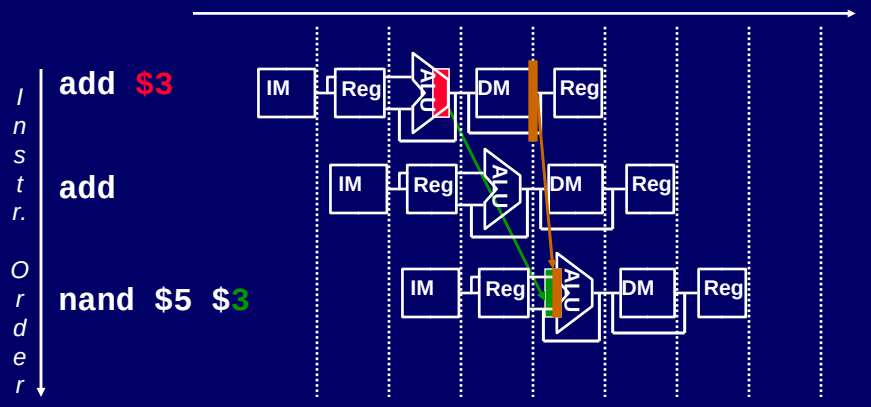


Data Hazards: AL ops

add 3 1 2
and 8 0 2
nand 5 3 4

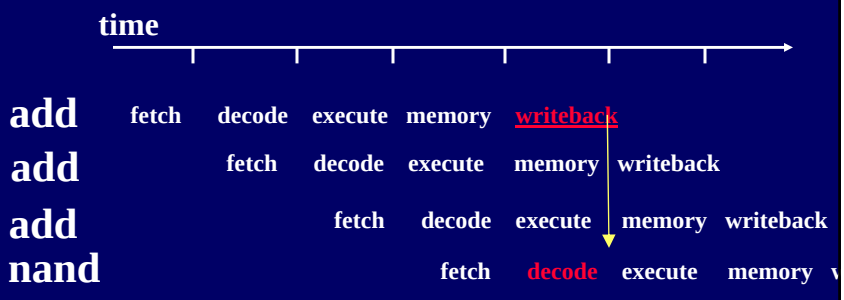


Forwarding Illustration



Data Hazards: AL ops

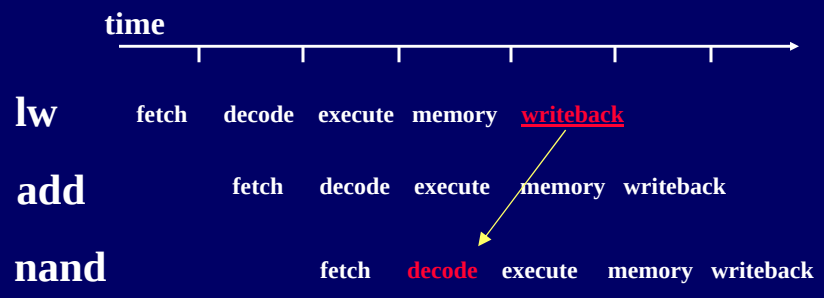
```
add   3 1 2
and   8 0 2
and   8 2 0
nand  5 3 4
```



Write in first half of cycle, read in second half of cycle

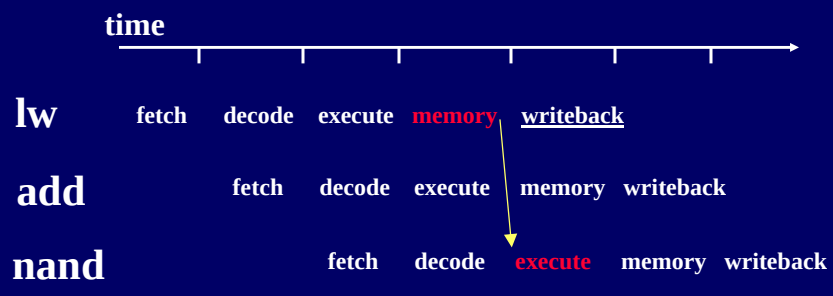
Data Hazards: lw

```
lw     3  12(zero)
and    8   0   2
nand   5   3   4
```



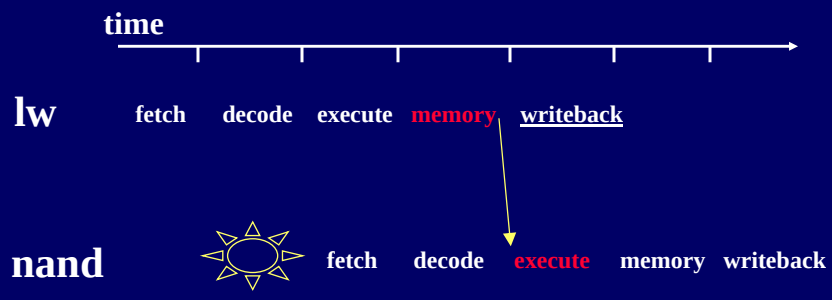
Data Hazards: lw

```
lw     3  12(zero)
and    8   0   2
nand   5   3   4
```



Data Hazards: lw

```
lw     3  12(zero)
and    8  0  2
nand   5  3  4
```



Handling Data Hazards: Summary

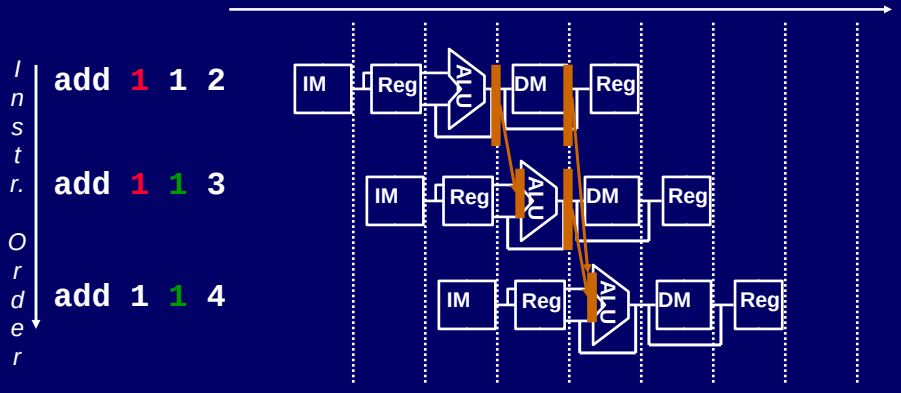
- Forward:
  - New bypass datapaths route computed data to where it is needed
- Beware: Stalling may still be required even in the presence of forwarding

Handling Data Hazards: Detection

- Detection
  - Compare regA with previous DestReg
    (5 bits in MIPS)
  - Compare regB with previous DestReg
    (5 bits in MIPS)

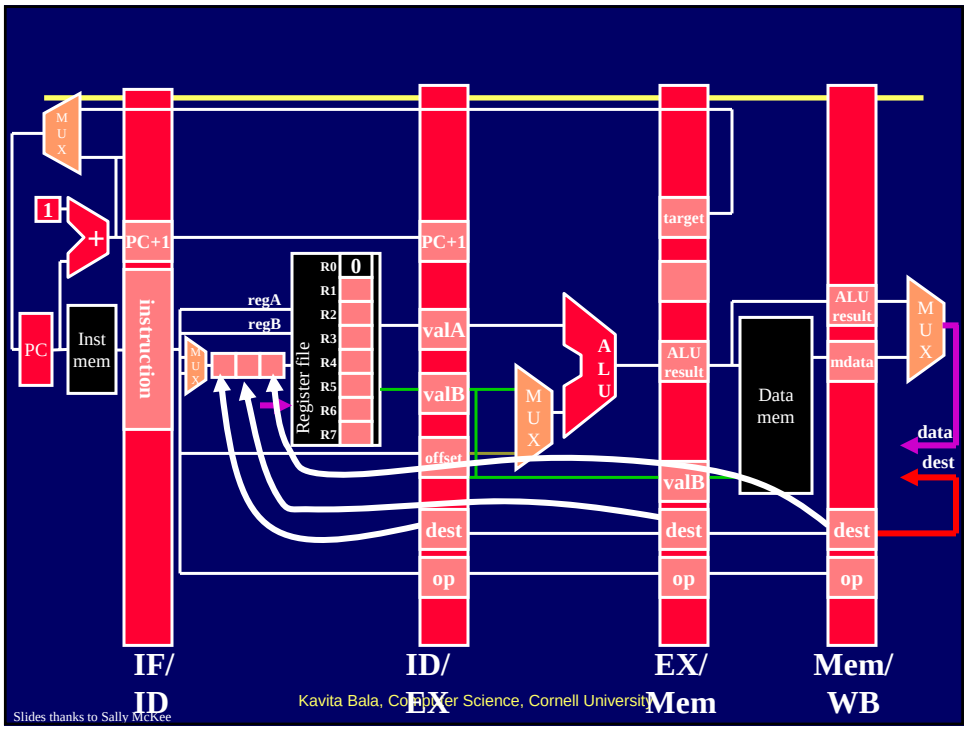
A tricky case

```
add 1 1 2
add 1 1 3
add 1 1 4
```



Another tricky case

```
add   0  4  1
add   3  0  4
```

Slides thanks to Sally McKee

Handling Data Hazards: Forwarding

- No point forwarding to decode
- Forward to EX stage
- From output of ALU and MEM stages

Sample Code

Which data hazards do you see?

```
add  3 1  2
nand 5  3  4
add  7 6  3
lw   6 24(3)
sw   6  12(2)
```

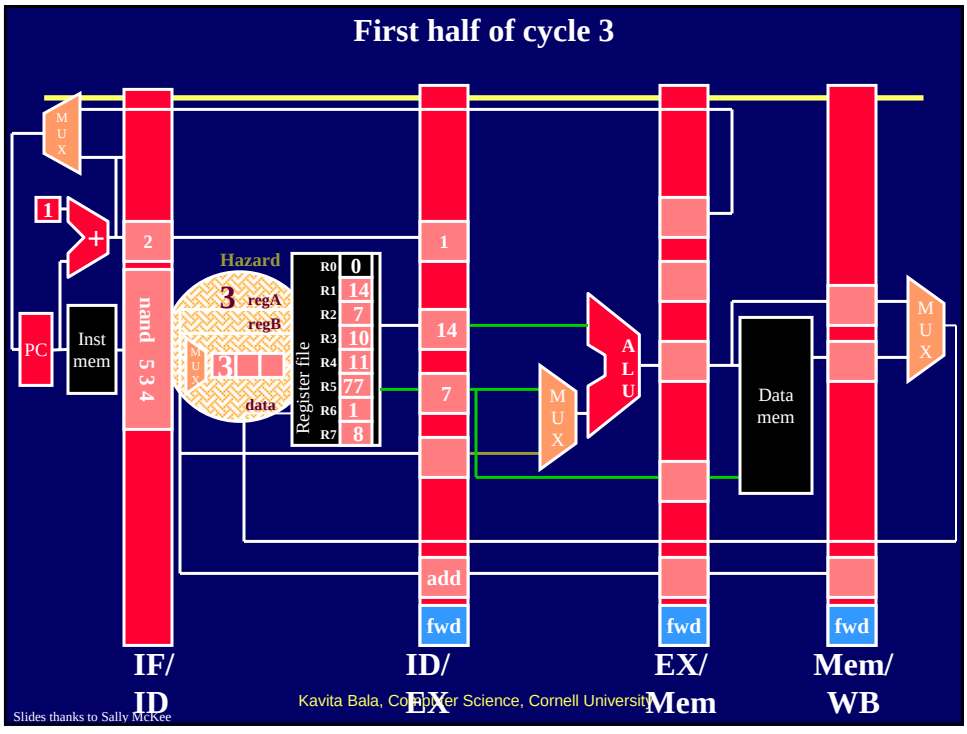

Slides thanks to Sally McKee

Sample Code

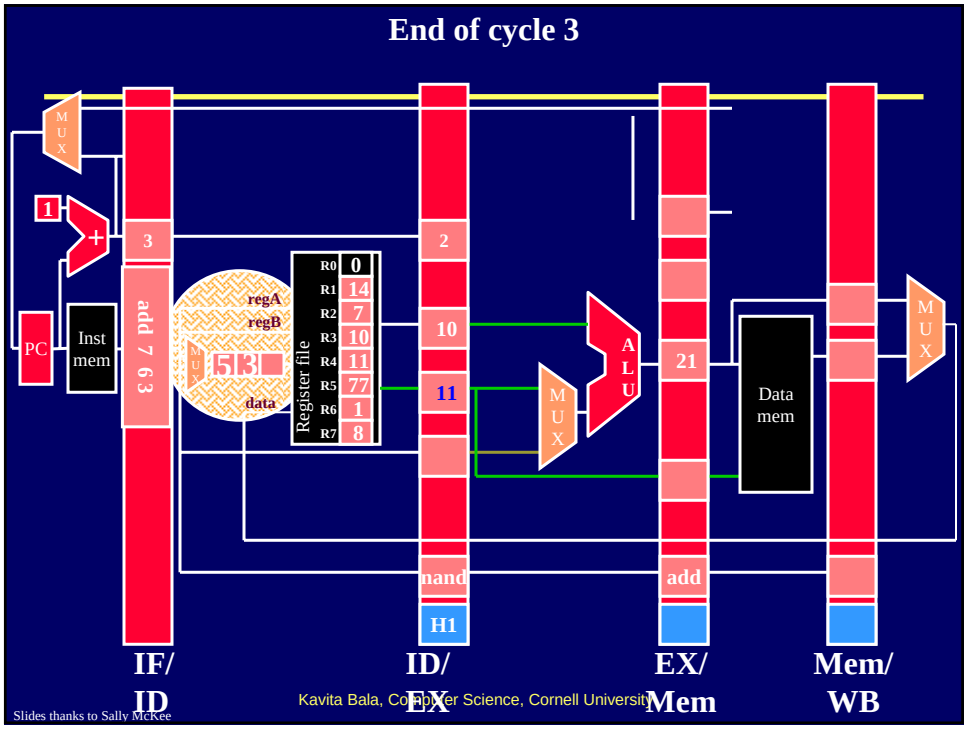
Which data hazards do you see?

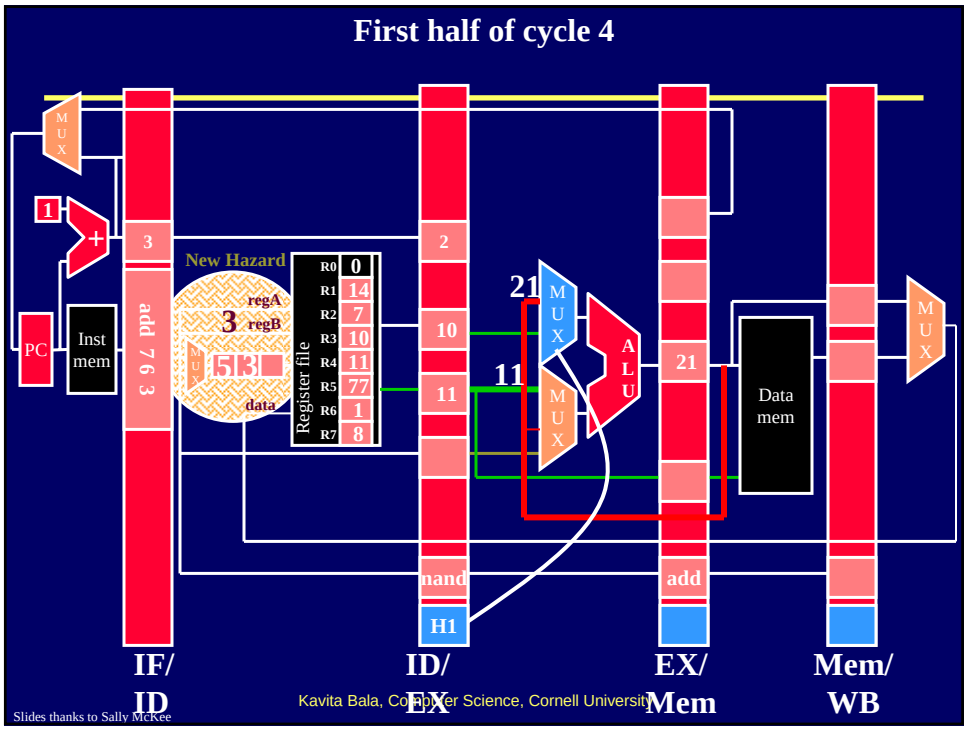
```
add  3 1  2
nand 5 3  4
add  7 6  3
lw   6 24(3)
sw   6 12(2)
```

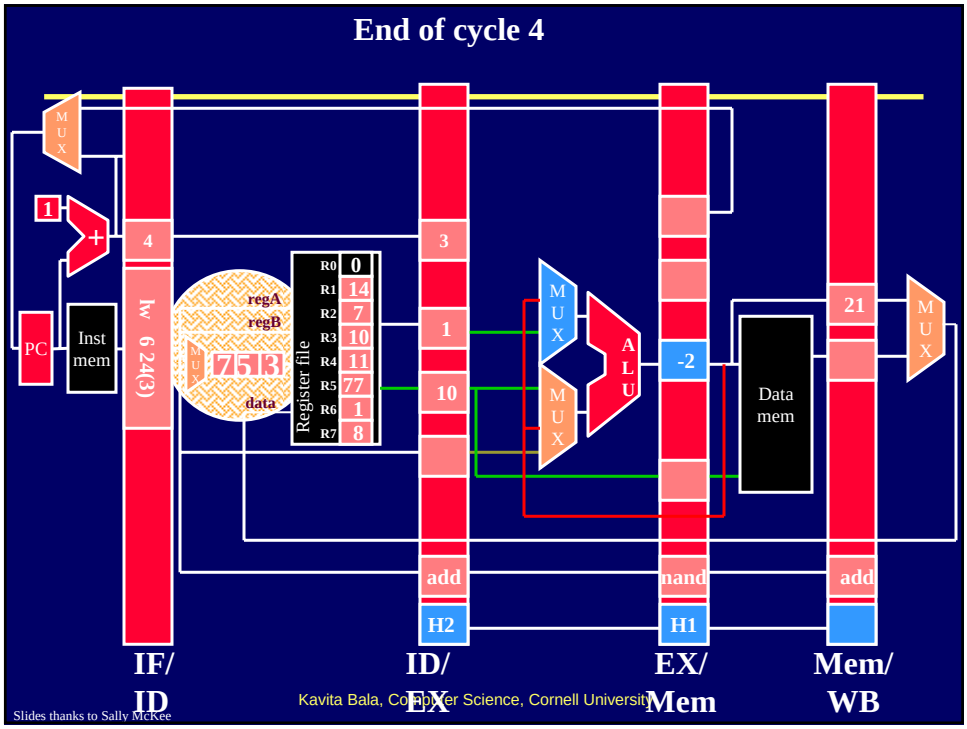
Slides thanks to Sally McKee

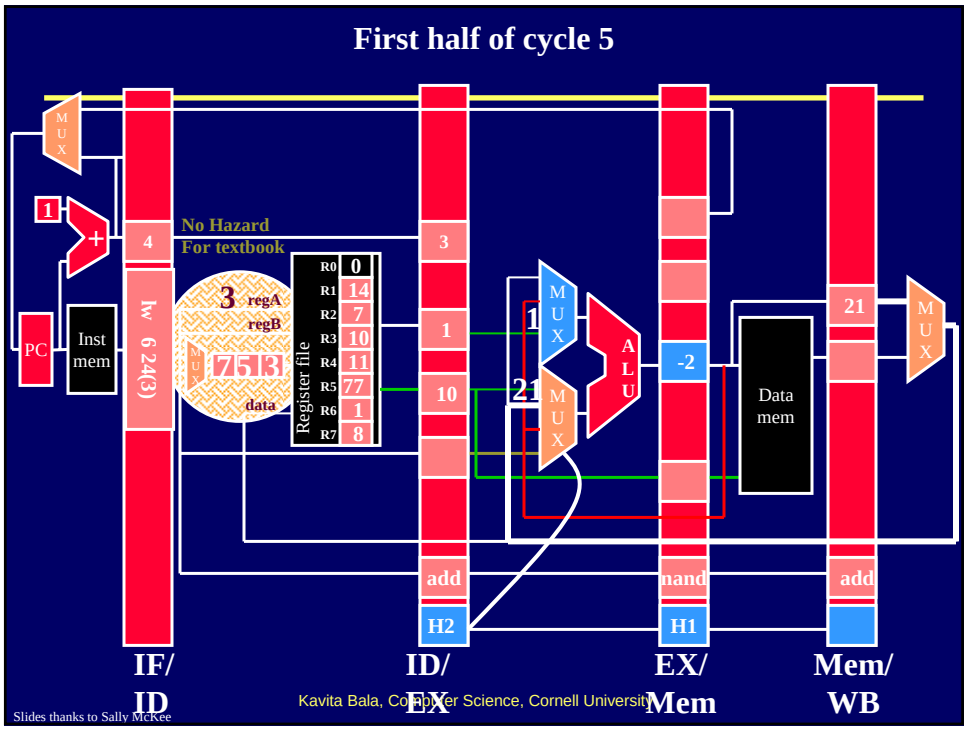


Slides thanks to Sally McKee

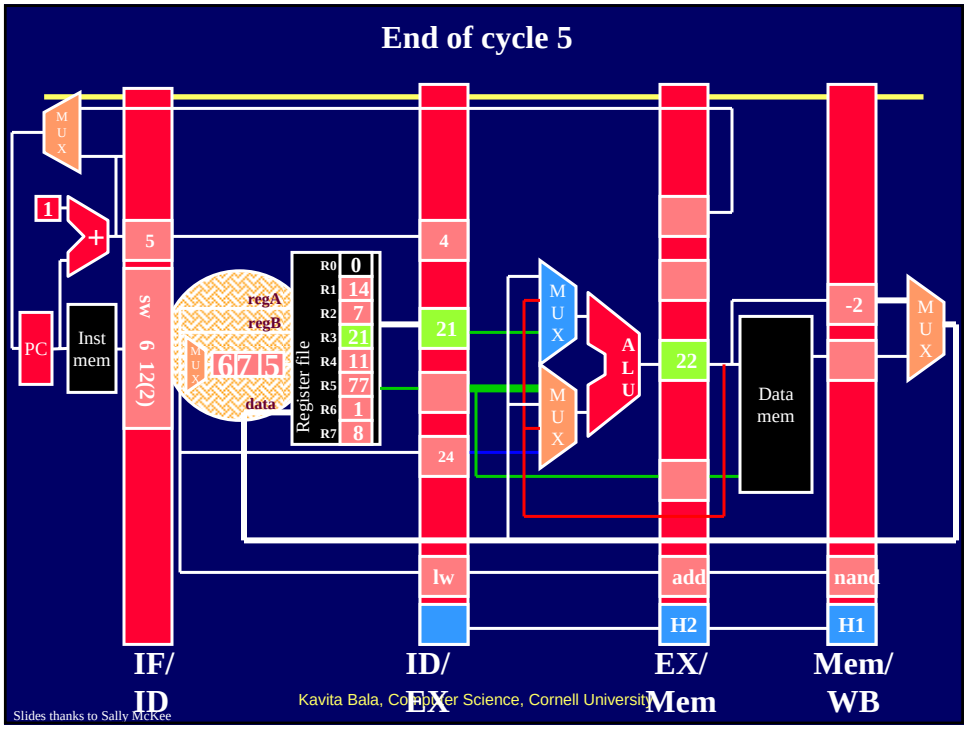

End of cycle 3
MUX
1
+
3
PC
Inst mem
add 7 6 3
regA
regB
MUX
5 3
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
2
10
11
nand
H1
MUX
ALU
21
add
Data mem
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB
Slides thanks to Sally McKee
Kavita Bala, Computer Science, Cornell University


First half of cycle 4
MUX
1
+
3
PC
Inst mem
add 7 6 3
New Hazard
regA
3
regB
MUX
5 3
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
2
10
11
21
MUX
11
MUX
ALU
21
add
Data mem
MUX
nand
H1
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB
Slides thanks to Sally McKee
Kavita Bala, Computer Science, Cornell University

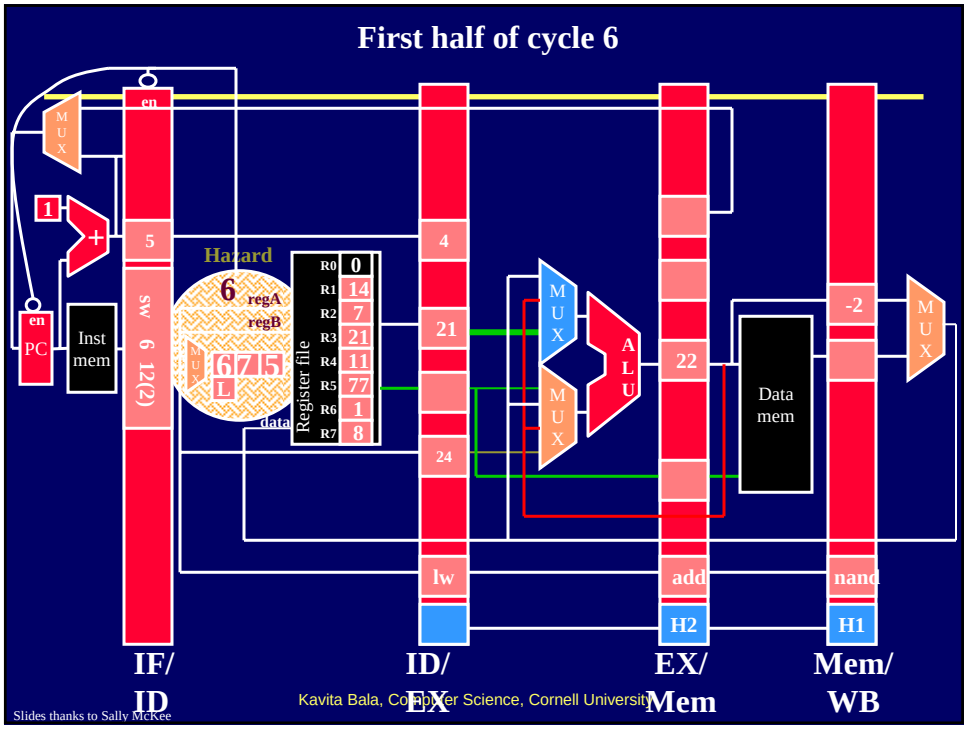

End of cycle 4
MUX
1
+
4
PC
Inst mem
lw 6 24(3)
regA
regB
MUX
75 3
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
3
1
10
add
H2
MUX
MUX
ALU
-2
nand
H1
Data mem
21
add
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB
Slides thanks to Sally McKee
Kavita Bala, Computer Science, Cornell University

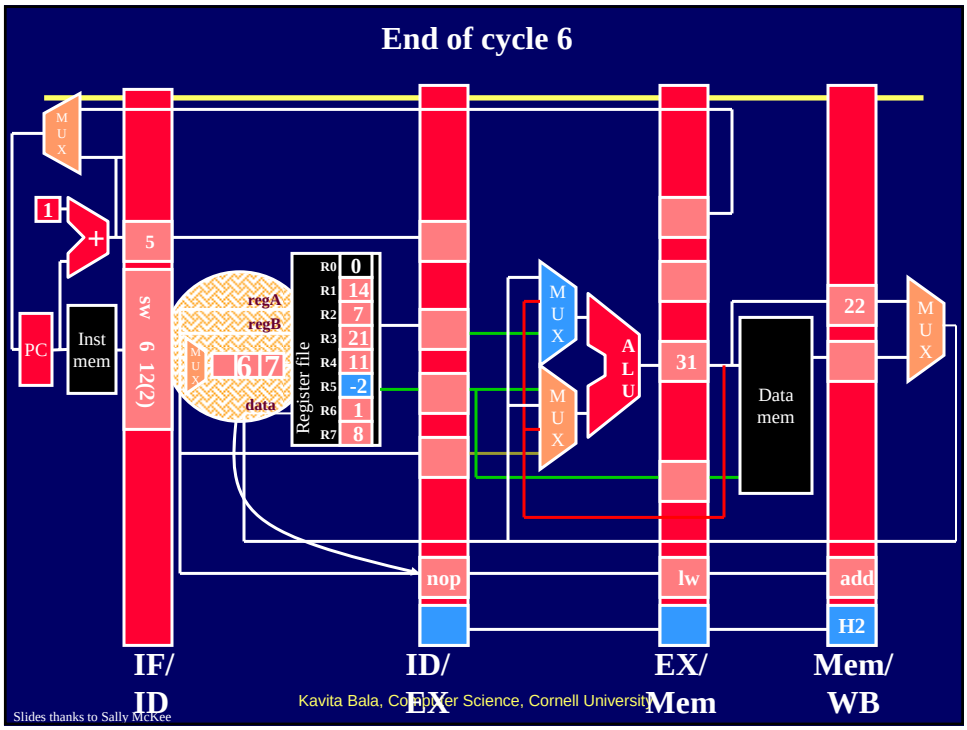

First half of cycle 5
MUX
1
+
4
No Hazard
For textbook
PC
Inst mem
lw 6 24(3)
3
regA
regB
MUX
75 3
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
3
1
10
1
21
MUX
MUX
ALU
-2
nand
H1
Data mem
21
add
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB
Slides thanks to Sally McKee
Kavita Bala, Computer Science, Cornell University

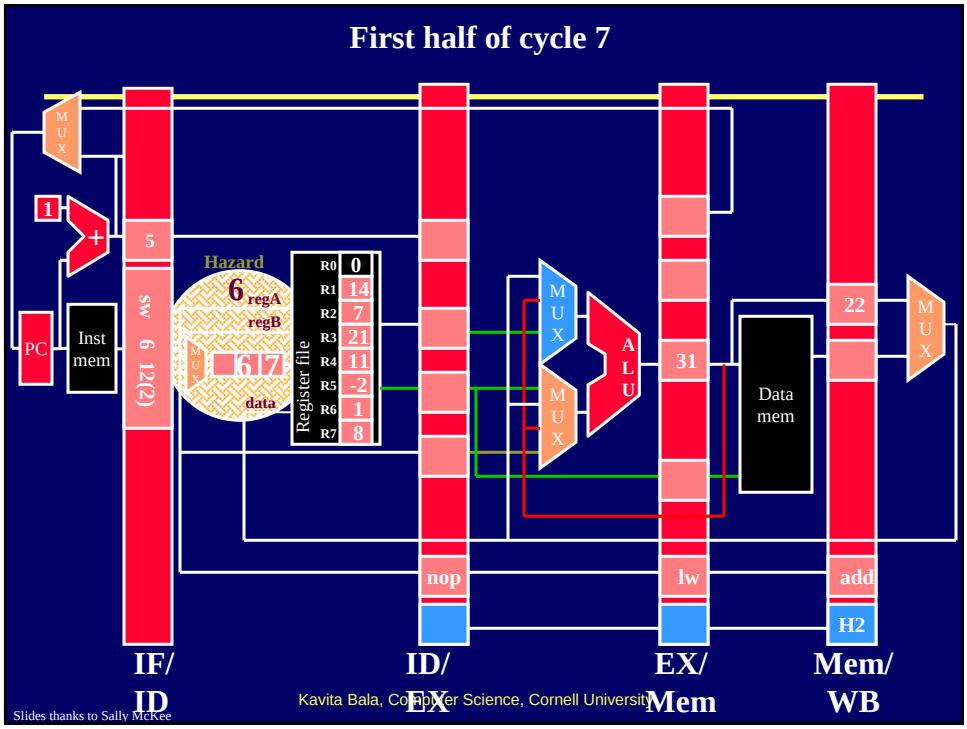
Handling Data Hazards: Stalling

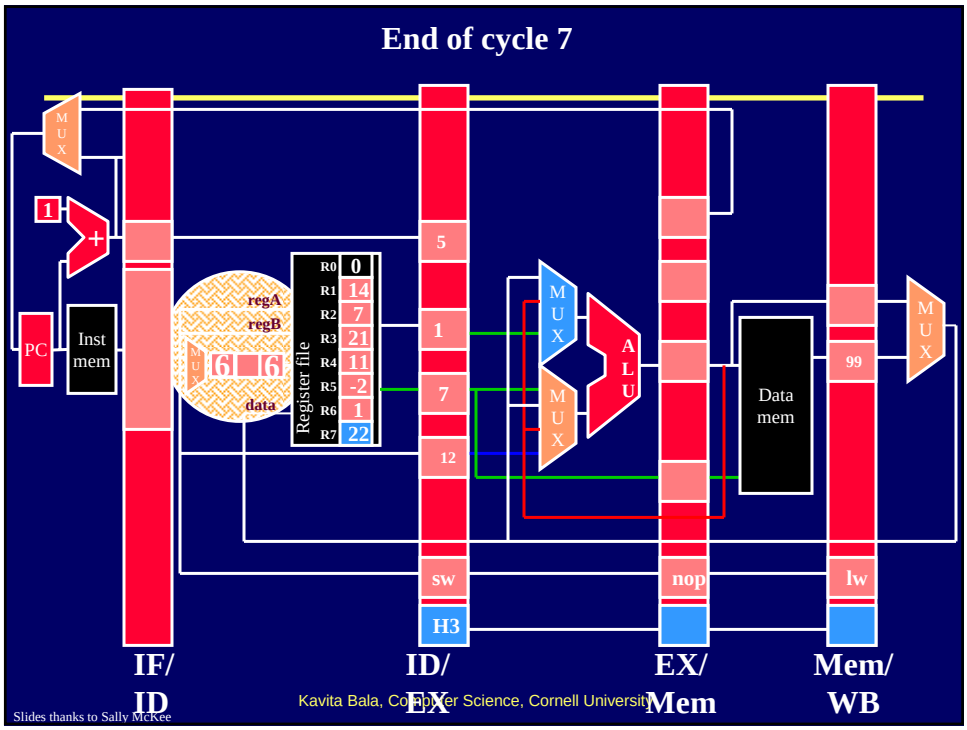


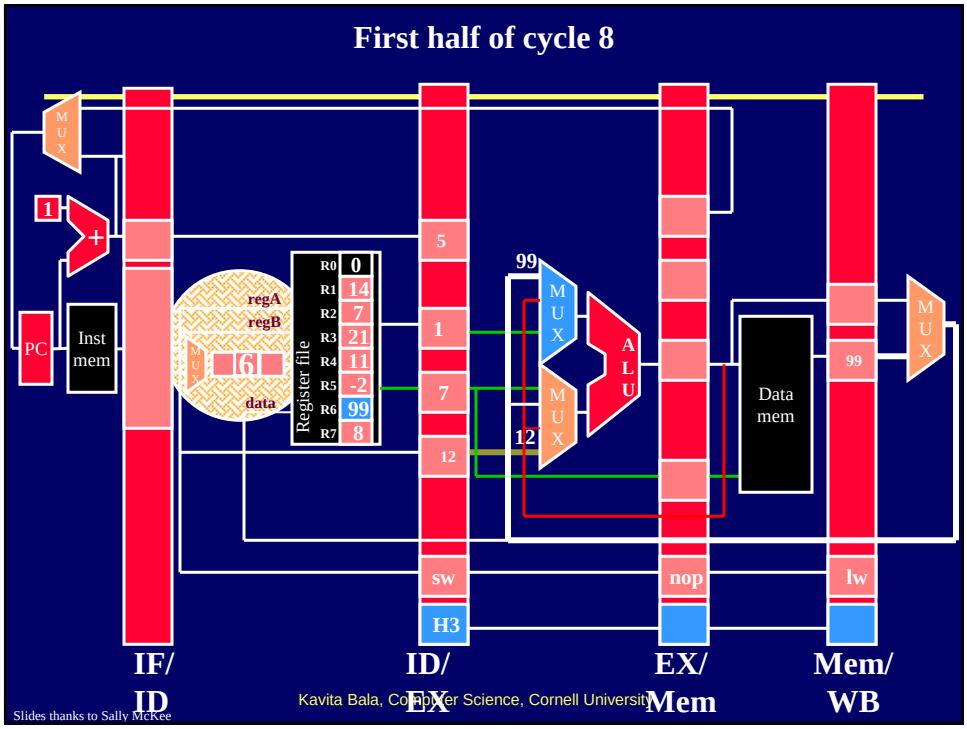
Slides thanks to Sally McKee

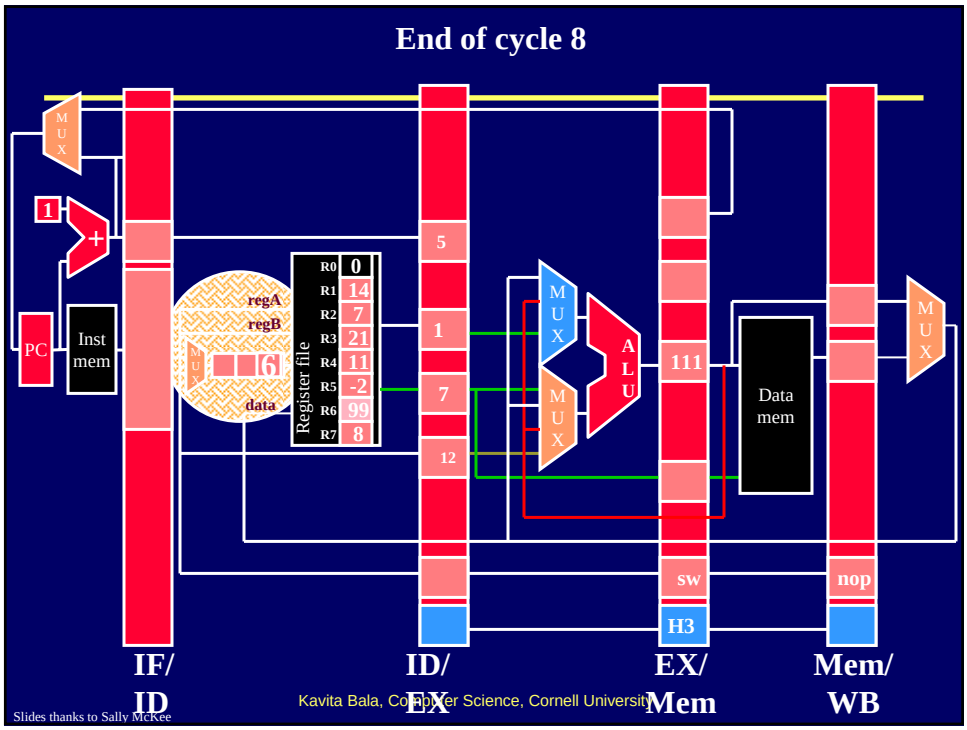

First half of cycle 6
MUX
1
+
en
PC
Inst mem
5
sw 6 12(2)
Hazard
6
regA
regB
6 7 5
L
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 77
R6 1
R7 8
4
21
24
lw
MUX
MUX
ALU
22
add
H2
Data mem
-2
nand
H1
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB
Kavita Bala, Computer Science, Cornell University
Slides thanks to Sally McKee


End of cycle 6
MUX
1
+
PC
Inst mem
5
sw 6 12(2)
regA
regB
6 7
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 1
R7 8
nop
MUX
MUX
ALU
31
lw
Data mem
22
add
H2
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB
Kavita Bala, Computer Science, Cornell University
Slides thanks to Sally McKee


First half of cycle 7
MUX
1
+
PC
Inst mem
5
sw 6 12(2)
Hazard
6
regA
regB
6 7
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 1
R7 8
MUX
MUX
ALU
31
Data mem
22
MUX
nop
lw
add
H2
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB
Slides thanks to Sally McKee
Kavita Bala, Computer Science, Cornell University


End of cycle 7
MUX
1
+
PC
Inst mem
5
1
7
12
sw
H3
regA
regB
6 6
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 1
R7 22
MUX
MUX
ALU
Data mem
99
MUX
nop
lw
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB
Slides thanks to Sally McKee
Kavita Bala, Computer Science, Cornell University


First half of cycle 8
MUX
1
+
PC
Inst mem
regA
regB
MUX
6
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 99
R7 8
5
1
7
12
sw
H3
99
MUX
MUX
ALU
12
Data mem
99
MUX
nop
lw
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB
Slides thanks to Sally McKee
Kavita Bala, Computer Science, Cornell University


End of cycle 8
MUX
1
+
PC
Inst mem
regA
regB
MUX
6
data
Register file
R0 0
R1 14
R2 7
R3 21
R4 11
R5 -2
R6 99
R7 8
5
1
7
12
sw
H3
MUX
MUX
ALU
111
Data mem
MUX
nop
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB
Slides thanks to Sally McKee
Kavita Bala, Computer Science, Cornell University

Control Hazards

- Stall
  - Inject NOPs into the pipeline when the next instruction is not known
  - Pros: simple, clean; Cons: slow
- Delay Slots
  - Tell the programmer that the N instructions after a jump will always be executed, no matter what the outcome of the branch
  - Pros: The compiler may be able to fill the slots with useful instructions; Cons: breaks abstraction boundary
- Speculative Execution
  - Insert instructions into the pipeline
  - Replace instructions with NOPs if the branch comes out opposite of what the processor expected
  - Pros: Clean model, fast; Cons: complex