Announcements

- Lab 7 meets Fri / Mon
- P2
  - part L is due Thursday next week
- Homework 3 Due Mon 11/6

Pipelining

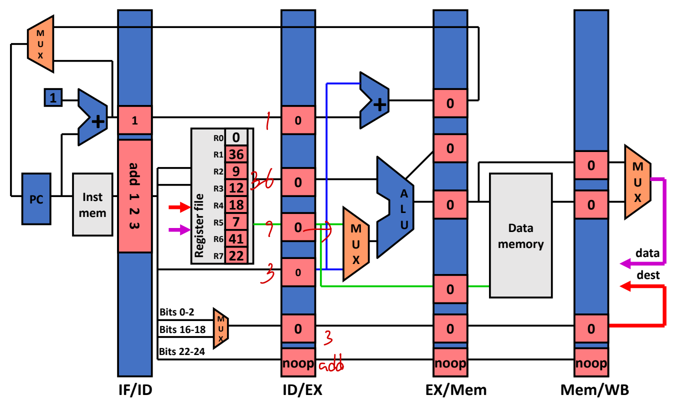
- Want to execute an instruction?
  - Build a processor (multi-cycle)
  - Find instructions
  - Line up instructions (1, 2, 3, …)
  - Overlap execution
    - Cycle #1: Fetch 1
    - Cycle #2: Decode 1 Fetch 2
    - Cycle #3: ALU 1 Decode 2 Fetch 3
    - . . . . . .
  - This is called pipelining instruction execution.
  - Used extensively for the first time on IBM 360 (1960s).
  - CPI approaches 1.

Sample Code (Simple)

Let's run the following code on pipelined LC2K:

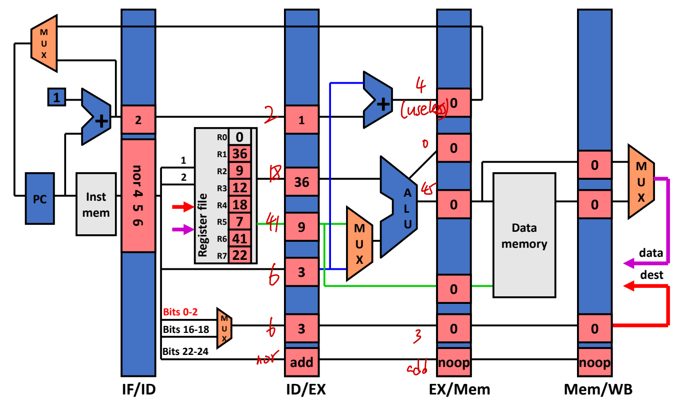
add	1	2	3	; reg 3 = reg 1 + reg 2
nor	4	5	6	; reg 6 = reg 4 nor reg 5
lw	2	4	20	; reg 4 = Mem[reg2+20]
add	2	5	5	; reg 5 = reg 2 + reg 5
sw	3	7	10	; Mem[reg3+10] =reg 7

Time 1 - Fetch: add 1 2 3



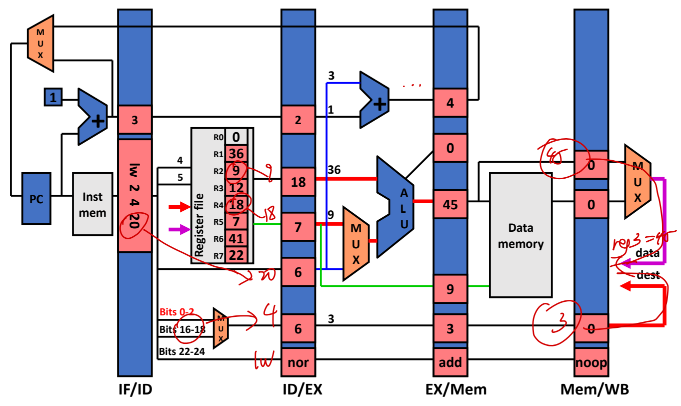
add 1 2 3

Time 2 - Fetch: nor 4 5 6



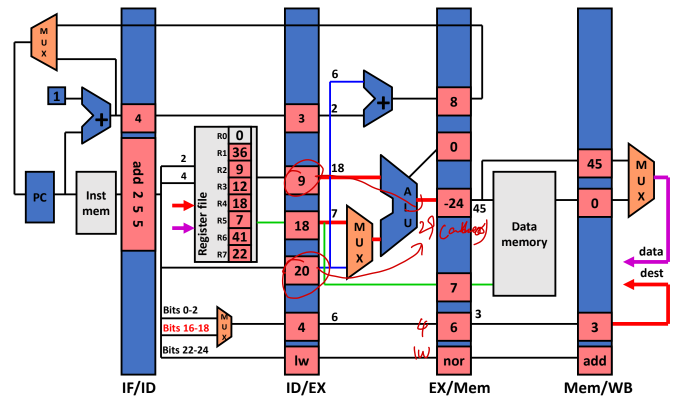
nor 4 5 6 add 1 2 3

Time 3 - Fetch: lw 2 4 20



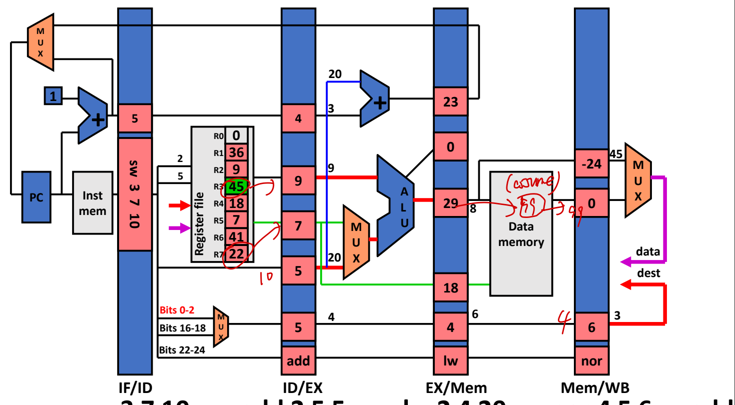
lw 2 4 20 nor 4 5 6 add 1 2 3

Time 4 - Fetch: add 2 5 5



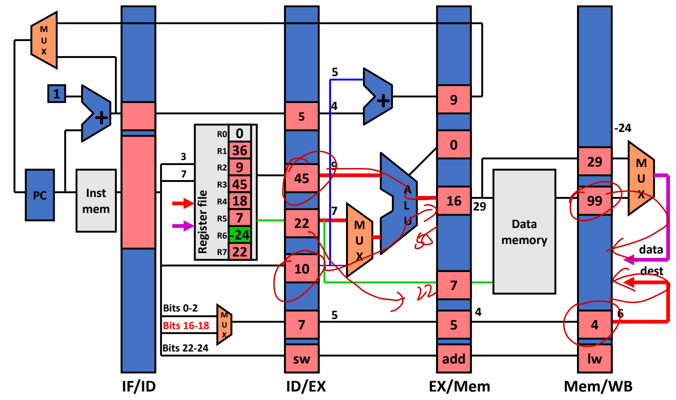
add 2 5 5 lw 2 4 20 nor 4 5 6 add 1 2 3

Time 5 - Fetch: sw 3 7 10



sw 3 7 10 add 2 5 5 lw 2 4 20 nor 4 5 6 add

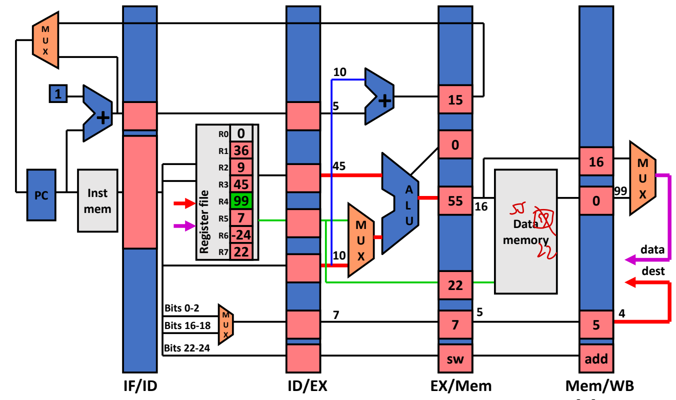
Time 6 – no more instructions



sw 3 7 10 add 2 5 5 lw 2 4 20 nor

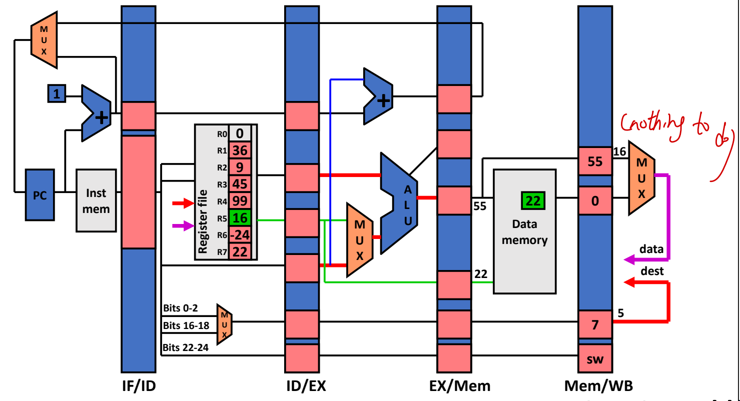
Time 7 – no more instructions

Poll: What all happens on the next clock cycle?



sw 3 7 10 add 2 5 5 lw

Time 8 – no more instructions



sw 3 7 10 add

Pipelining - What can go wrong?

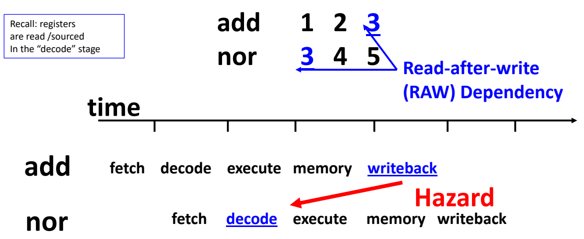
- Data hazards: since register reads occur in stage 2 and register writes occur in stage 5 it is possible to read an old / stale value before the correct value is written back.
- Control hazards: A branch instruction may change the PC, but not until stage 4. What do we fetch before that?
- Exceptions: Sometimes we need to pause execution, switch to another task (maybe the OS), and then resume execution… how to we make sure we resume at the right spot
- Now - Data hazards
  - What are they?
  - How do you detect them?
  - How do you deal with them?

Pipeline function for ADD

- Fetch: read instruction from memory
- Decode: <u>read source operands from reg</u>
- Execute: calculate sum
- Memory: pass results to next stage
- Writeback: <u>write sum into register file</u>

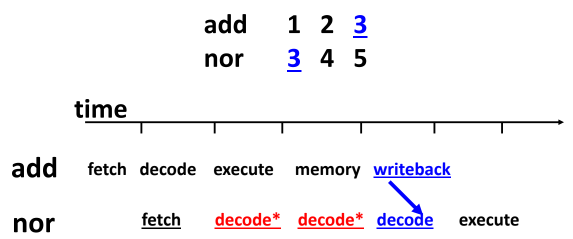
Data Hazards

Recall: registers are read /sourced In the "decode" stage


add 1 2 3
nor 3 4 5
Read-after-write (RAW) Dependency
time
add fetch decode execute memory writeback
Hazard
nor fetch decode execute memory writeback

If not careful, nor will read a stale value of register 3

Data Hazards


add 1 2 3
nor 3 4 5
time
add fetch decode execute memory writeback
nor fetch decode* decode* decode execute

Assume Register File gives the right value of register 3 when read/written during <u>same</u> cycle. This is consistent with most processors (ARM/x86), <u>but not Project 3</u>.

Definitions

- Data Dependency: *one instruction uses the result of a previous one*
  - Doesn't necessarily cause a problem
- Data Hazard: *one instruction has a data dependency that will cause a problem if we don't "deal with it"*

Class Problem 1

Poll: Which of these instructions has a data dependency on an earlier one? Which of those are data hazards in our 5-stage pipeline?

1. add 1 2 3
2. nor 3 4 5
3. add 6 3 7
4. lw 3 6 10
5. sw 6 2 12

What about here?

1. add 1 2 3
2. beq 3 4 1
3. add 3 5 6
4. add 3 6 7

Class Problem 1

Which read-after-write (RAW) dependences do you see?

Which of those are data hazards?

1. add 1 2 3
2. nor 3 4 5
3. add 6 3 7
4. lw 3 6 10
5. sw 6 2 12

What about here?

1. add 1 2 3
2. beq 3 4 1
3. add 3 5 6
4. add 3 6 7

Three approaches to handling data hazards

- Avoid
  - Make sure there are no hazards in the code
- Detect and Stall
  - If hazards exist, stall the processor until they go away.
- Detect and Forward
  - If hazards exist, fix up the pipeline to get the correct value (if possible)

Handling data hazards I: Avoid all hazards

- Assume the programmer (or the compiler) knows about the processor implementation.
  - Make sure no hazards exist.
    - Put noops between any dependent instructions.

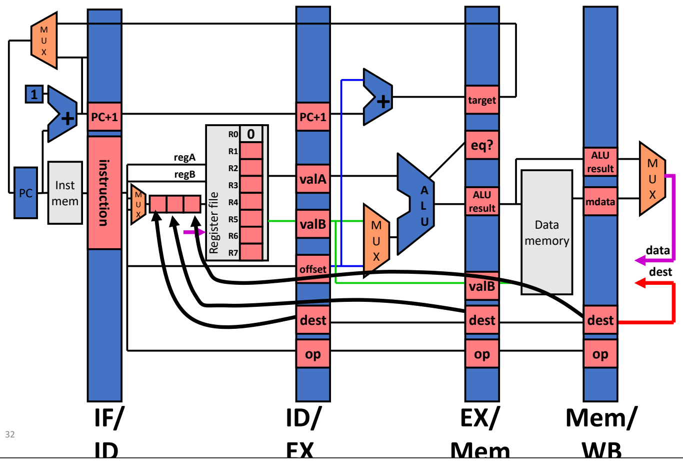
add 1 2 3 ← **write register 3 in cycle 5**
noop
noop
nor 3 4 5 ← **read register 3 in cycle 5**

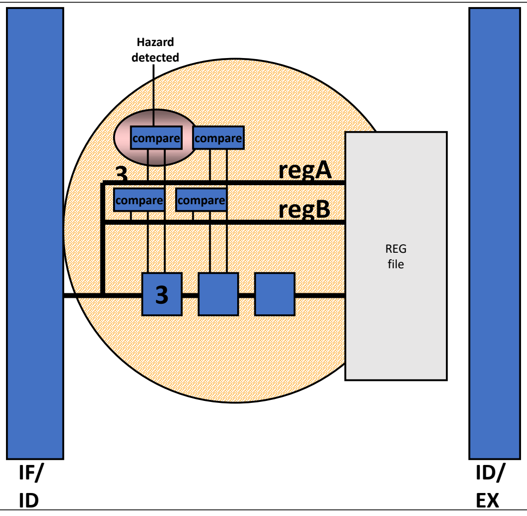
Problems with this solution

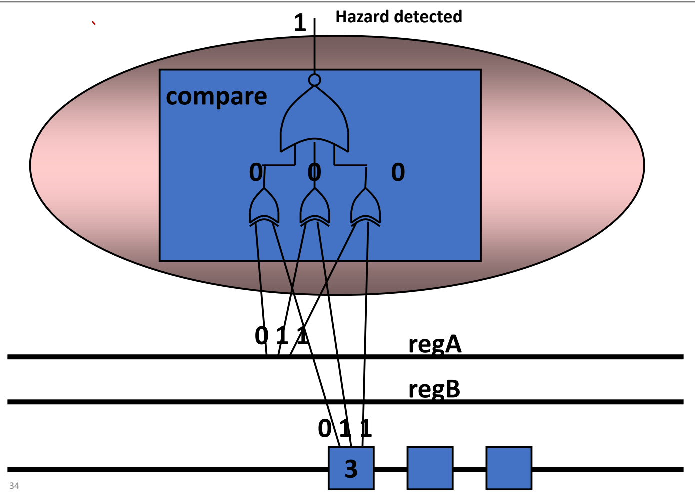
- Old programs (legacy code) may not run correctly on new implementations
  - Longer pipelines need more noops
- Programs get larger as noops are included
  - Especially a problem for machines that try to execute more than one instruction every cycle
  - Intel EPIC: Often 25% - 40% of instructions are noops
- Program execution is slower
  - CPI is 1, but some instructions are noops

Handling data hazards II: Detect and stall until ready

- Detect:
  - Compare regA with previous DestRegs
    - 3 bit operand fields
  - Compare regB with previous DestRegs
    - 3 bit operand fields
- Stall:
  - Keep current instructions in fetch and decode
  - Pass a noop to execute
- How do we modify the pipeline to do this?







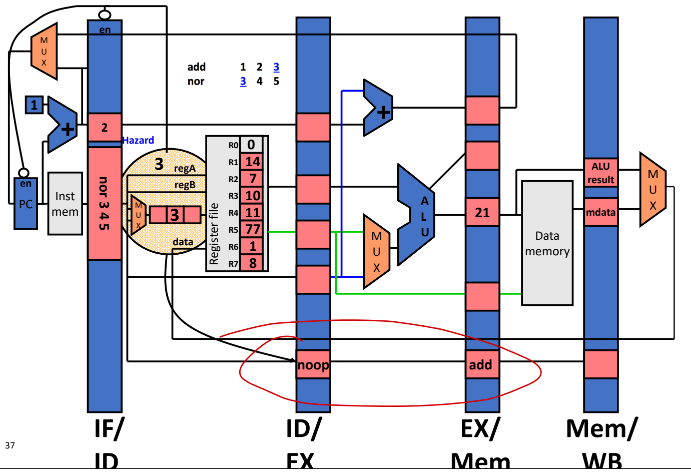
Example

- Let's run this program with a data hazard through our 5-stage pipeline
  add 1 2 3
  nor 3 4 5
- We will start at the beginning of cycle 3, where add is in the EX stage, and nor is in the ID stage, about to read a register value

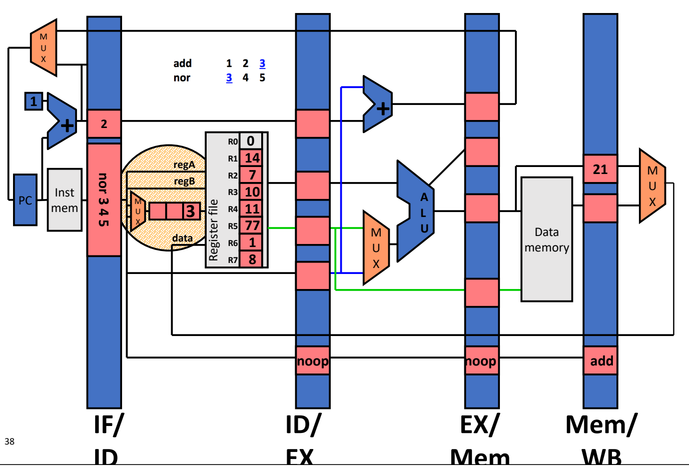
Time:	1	2	3
add 1 2 3	IF	ID	EX
nor 3 4 5		IF	ID

Hazard!

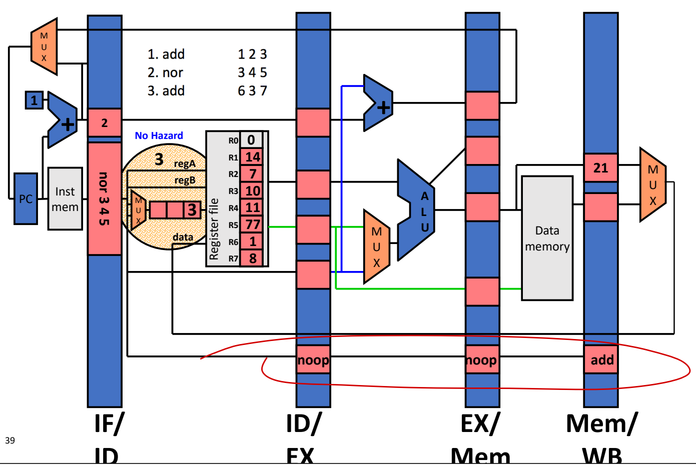
First half of cycle 4



End of cycle 4

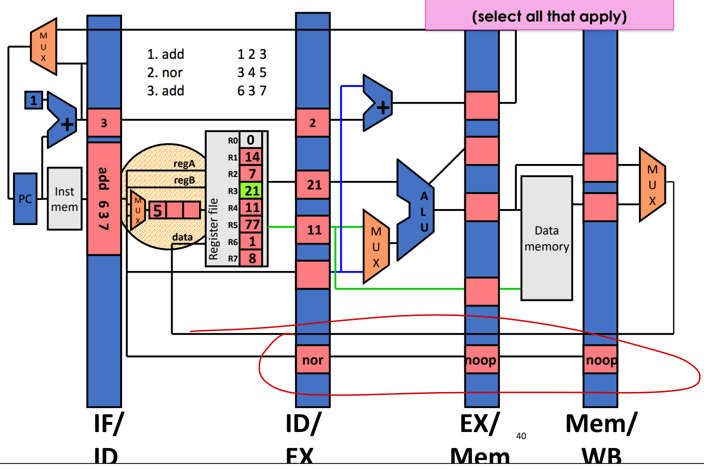


First half of cycle 5



End of cycle 5

Poll: How many cycles of delay can each hazard introduce? (select all that apply)



Time Graph

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add 1 2 3	IF	ID	EX	ME	WB								
nor 3 4 5		IF	ID*	ID*	ID	EX	ME	WB					
add 6 3 7													
lw 3 6 10													
sw 6 2 12													

Solution

Poll: Which problems does "detect and stall" fix over "avoid hazards"? (select all)
1. Breaking backwards compatibility
2. Larger programs
3. Slower programs

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add 1 2 3	IF	ID	EX	ME	WB								
nor 3 4 5		IF	ID*	ID*	ID	EX	ME	WB					
add 6 3 7					IF	ID	EX	ME	WB				
lw 3 6 10						IF	ID	EX	ME	WB			
sw 6 2 12							IF	ID*	ID*	ID	EX	ME	WB

Live Poll + Q&A: slido.com #eecs370