Lecture 9

Pipeline Hazards

Christos Kozyrakis
Stanford University
http://eeclass.stanford.edu/ee108b

Announcements

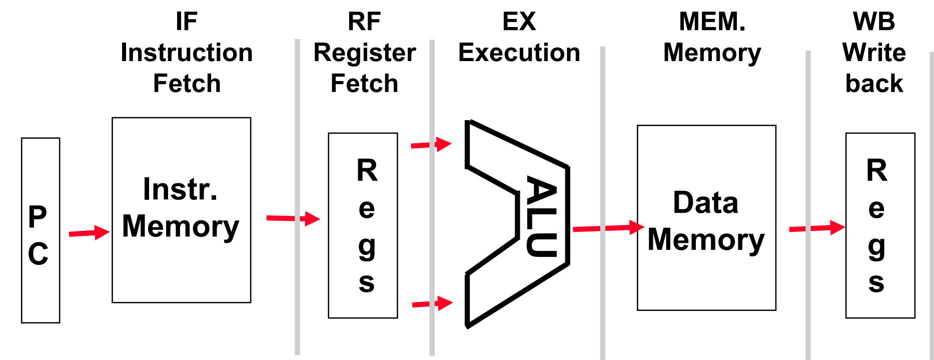
- PA-1 is due today
  - Electronic submission
- Lab2 is due on Tuesday 2/13th
- Quiz1 grades will be available next week
  - Solutions will be posted on line tomorrow
- Tuesday 2/13th lecture will be a video playback
  - Can come to lecture hall or watch from home or offline

Review: Single-cycle Datapath (load instruction)

- IF: Instruction Fetch
  - Fetch the instruction from memory
  - Increment the PC by 4
- RF/ID: Register Fetch and Instruction Decode
  - Fetch base register
- EX: Execute
  - Calculate base + sign-extended immediate
- MEM: Memory
  - Read the data from the data memory
- WB: Write back
  - Write the results back to the register file

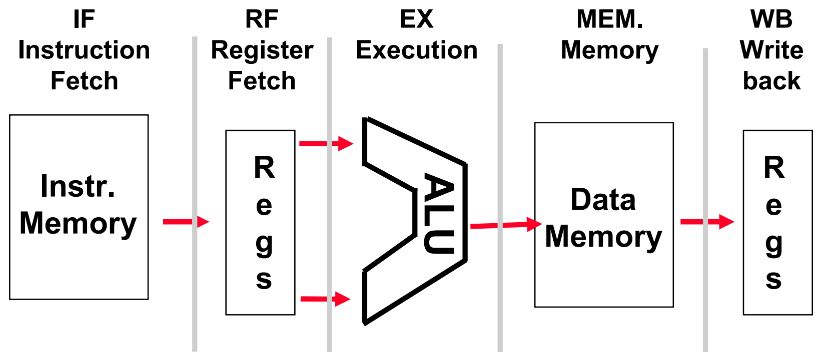
Review: Multi-cycle Datapath

- Divide datapath into steps
  - Each step is one clock cycle
  - Note RF is a short cycle, but waits for clock



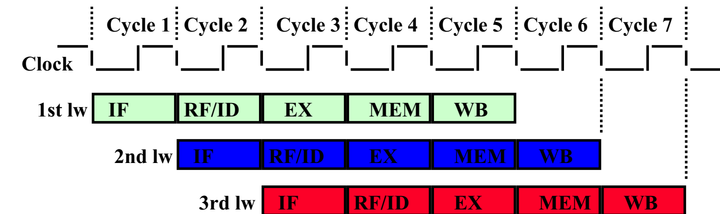
Review: Full Pipeline

- Fetch a new instruction each cycle
  - Each stage of the pipeline is working on a different instruction

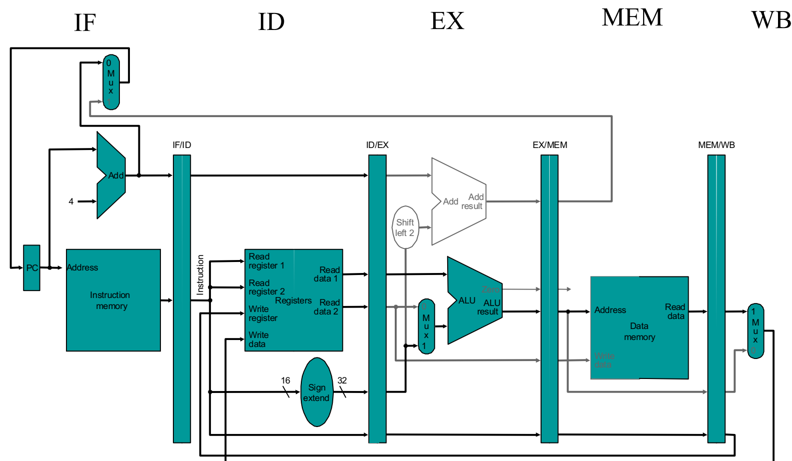


Review: Pipelining Load

- Load instruction takes 5 stages
  - Five independent functional units work on each stage
    - Each functional unit used only once
  - Another load can start as soon as 1st finishes IF stage
  - Each load still takes 5 cycles to complete
  - The *throughput*, however, is much higher



Pipeline Datapath



Important ISA Issues

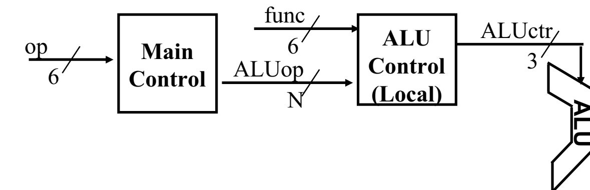
- Instruction length
  - Fixed MIPS instruction length allows easy pipeline even though decode does not happen until the second stage
  - Intel 80x86 has a much more challenging problem where instructions vary from 1-17 bytes
- Instruction formats
  - Register access possible even though decode does not happen until second stage due to regularity of formats
- Limited memory access
  - Since only loads and stores access memory, other instructions do not need to use the ALU to calculate the memory address before the actual computation

Review: Control Signals

func	10 0000	10 0010	Not Important				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	sub	ori	lw	sw	beq	jump
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
Branch	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Sub	Or	Add	Add	Sub	xxx

Review: Multilevel Decoding

- Since only the ALU needs the func field
  - Pass it to the ALU unit, and have a local decoder there



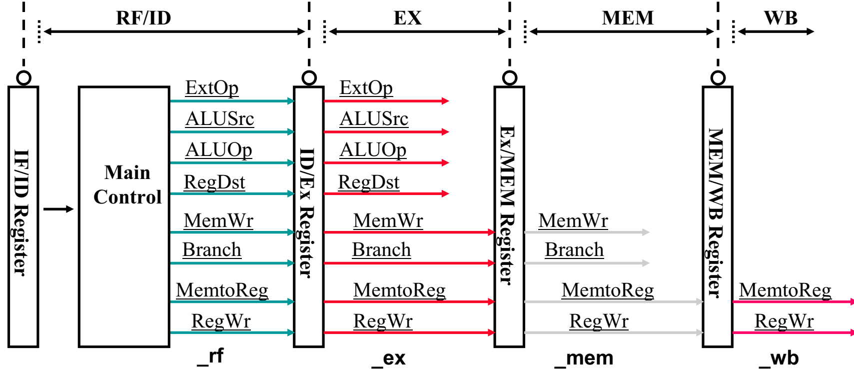
Pipeline Control

- Need to control functional units
  - But they are from working on different instructions!
- Not a problem
  - Just pipeline the control signals along with the data
  - Make sure they line up
- Using labeling conventions often helps
  - Instruction_rf – means this instruction is in RF
  - Every time it gets flopped, changes pipestage
    - Make sure right signals go to the right places

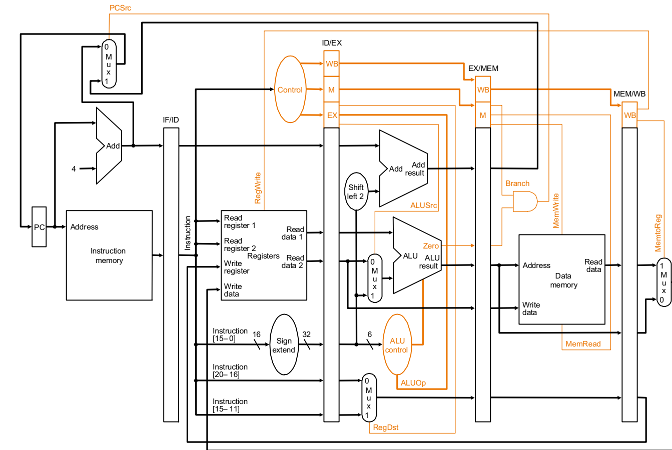
Control Signals

- Use a Main Control unit to generate signals during RF/ID Stage
  - Control signals for EX
    - (ExtOp, ALUSrc, …) used 1 cycle later
  - Control signals for Mem
    - (MemWr, Branch) used 2 cycles later
  - Control signals for WB
    - (MemtoReg, MemWr) used 3 cycles later

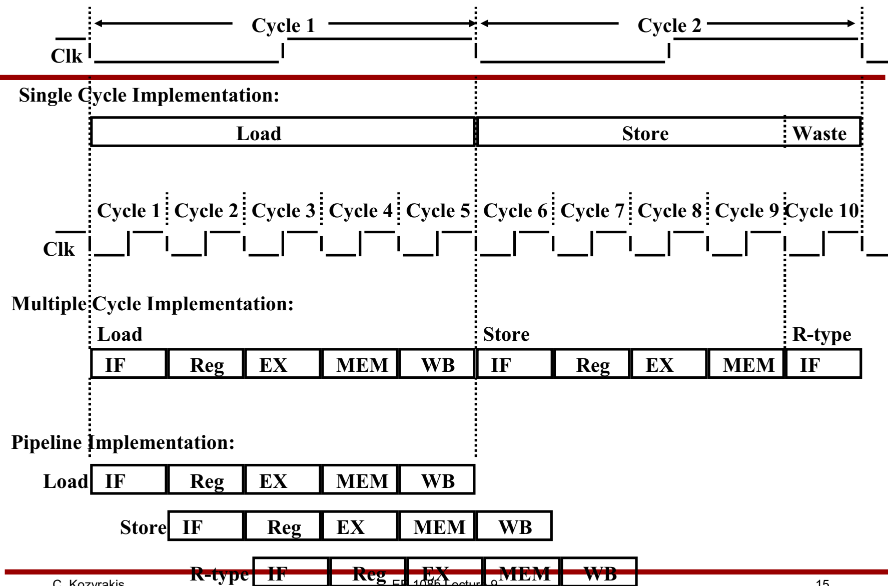
Implementing Control



Putting it All Together



Comparison



But Something Is Fishy Here

- If dividing it into 5 parts made the clock faster
  - And the effective CPI is still one
- Then dividing it into 10 parts would make the clock even faster
  - And wouldn't the CPI still be one?
- Then why not go to twenty cycles?
- Really two issues
  - Some things really have to complete in a cycle
    - Find next PC from current PC
  - CPI is not really one
    - Sometimes you need the results from the previous instruction

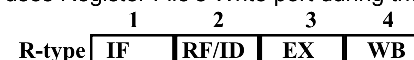
Pipeline Hazards

- These are dependencies between instructions that are exposed by pipelining
  - Causes pipeline to loose efficiency (pipeline stalls, wasted cycles)
  - If all instructions are dependent
    - No advantage of a pipelining (since all must wait)
- These limits to pipelining are known as hazards
  - Structural Hazard (Resource Conflict)
    - Two instructions need to use the same piece of hardware
  - Data Hazard
    - Instruction depends on result of instruction still in the pipeline
  - Control Hazard
    - Instruction fetch depends on the result of instruction in pipeline

Structural Hazards

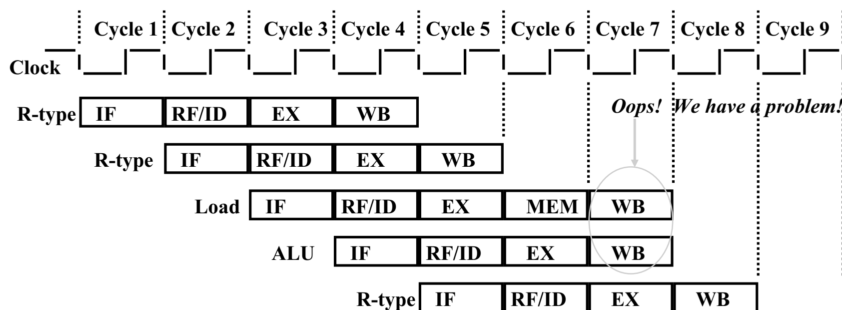
- Resource conflict
  - Occurs when two instructions try to use same hardware
  - Often arise when some functional unit is not fully pipelined
- To avoid structural hazards:
  - Use resource once per instruction
  - Always in the same cycle, so this arrangement is bad:
    - Load uses Register File's Write port during its 5th stage

- R-type uses Register File's Write port during the 4th stage



Structural Hazard Example

- Consider a load followed immediately by an ALU operation
  - Register file only has a single write port
    - But need to write the results of the ALU and the memory back

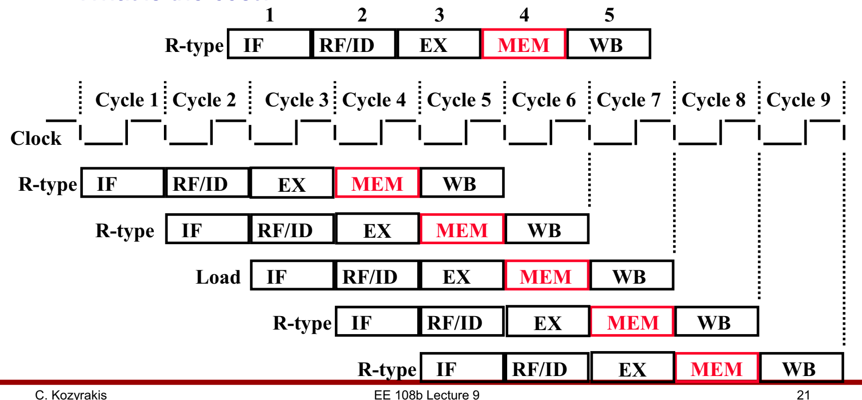


Structural Hazard Solutions

- Follow directions
  - Always use resources at the same time in each instruction
- Build more complex functional units
  - For example support two writes into register file
    - But then you would probably want to make use of this feature in other ways, and you probably still would like to follow directions …
- Delay offending instruction (queue up to use the unit)
  - Used when units are not fully pipelined (mult, div)
  - Hardware inserts a pipeline stall(bubble) that delays offending instruction
  - Increases CPI from the ideal value of 1

Delayed Write Back Solution

- Delay R-type register write by one cycle
  - Does this increase the CPI of instruction?
  - What is the cost?



Aside (kind of): Data Dependencies

- When a compiler is generating code
  - It often tries to reorder code to make pipeline machine faster
- It needs to make sure that the new code sequence
  - Is the same as the old code sequence
- When can it reorder instructions?
  - When the new order does not violate any data dependencies
  - Obvious issue is read after write (true dependence)
    - Need to produce value before reading it
  - But there are others as well
    - WAW
    - WAR

Data Hazard Types

- Data dependencies for instruction *j* following instruction *i*
  - Read after Write (RAW) (true dependence)
    - Instruction *j* tries to read before instruction *i* tries to write it
  - Write after Write (WAW) (output dependence)
    - Instruction *j* tries to write an operand before *i* writes its value
  - Write after Read (WAR) (anti dependence)
    - Instruction *j* tries to write a destination before it is read by *i*
- No such thing as a Read after Read (RAR) hazard since there is never a problem reading twice

Dependency Examples

- True dependency => RAW hazard

```
addu    $t0, $t1, $t2
subu    $t3, $t4, $t0
```

- Output dependency => WAW hazard

```
addu    $t0, $t1, $t2
subu    $t0, $t4, $t5
```

- Anti dependency => WAR hazard

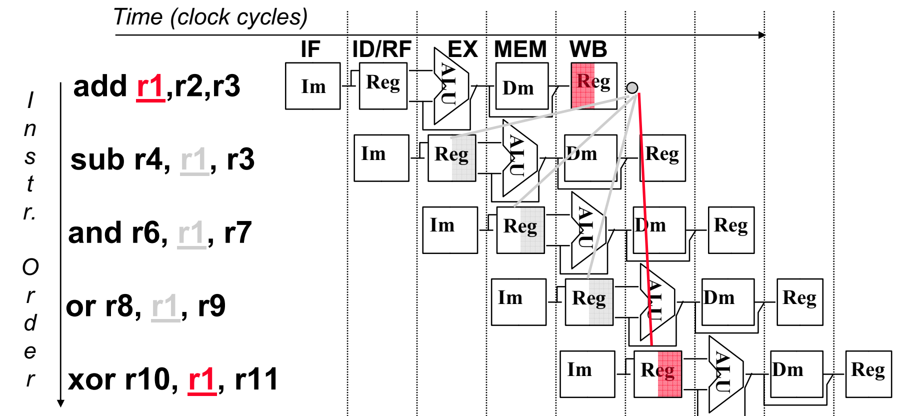
```
addu    $t0, $t1, $t2
subu    $t1, $t4, $t5
```

Data Hazards

- The cost of the additional cycle
  - Programmers think instruction execute in one cycle
  - Need to maintain that illusion even when pipelining
- If the register read and write occur in different pipe stages
  - If reads are early and writes are late
  - Programmer expects the previous instructions have finished
    - They expect the new values
    - But the register file has the old values
    - Violate the RAW dependence
- Can't have WAW or WAR hazards unless:
  - Writes occur in different cycles for different instructions, or reads can occur after writes for an instruction
  - Neither generally happen in five stage pipeline

Data Hazard Example

- Dependencies backwards in time are hazard

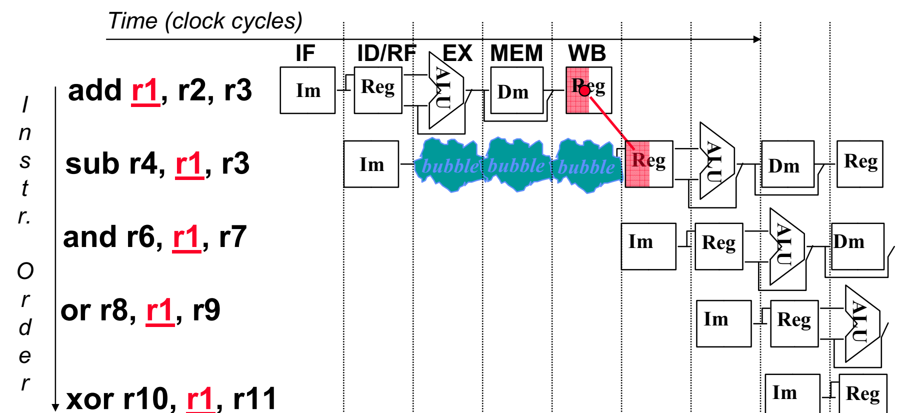


Data Hazard Solution

- For RAW hazards,
  - You need to delay instruction until data is available
- Options
  - How to wait
  - When do you declare the data is available
    - Talk about this soon
- How to wait – how to stall the pipeline
  - Have the software insert NOPs so this problem does not occur
  - Create hardware that stalls the pipeline
    - Dynamically create the NOPs
    - Pipeline interlocks
- Most machines will stall if needed
  - But the compiler tries to minimize stalls

Data Hazard - Stalls

- Eliminate reverse time dependency by stalling



Performance Effect

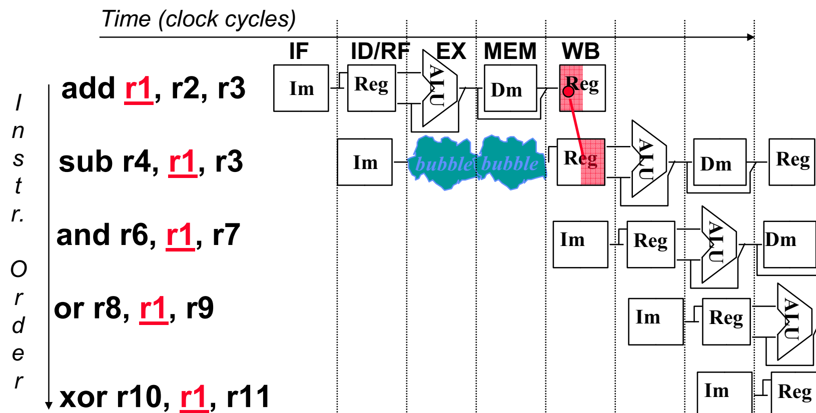
- Stalls can have a significant effect on performance
- Consider the following case
  - The ideal CPI of the machine is 1
  - A RAW hazard causes a 3 cycle stall
- If 40% of the instructions cause a stall?
  - The new effective CPI is $1 + 3 \times 0.4 = 2.2$
  - And the real % is probably higher than 40%
- You get less than ½ the desired performance!

Reducing Stalls

- Key is being careful about when you say data is available
- In our simple model
  - Cycle after WB
    - Easiest from hardware, but slow
- Since RF access are fast,
  - We can allow data to flow through register file
    - If you read a register when it is being written, you get new value
    - Or assume write during 1st ½ of cycle, read during 2nd ½
    - Now you stall only 2 cycles

Data Hazard – Register File

- Register file writes on first half and reads on second half



Performance Effect

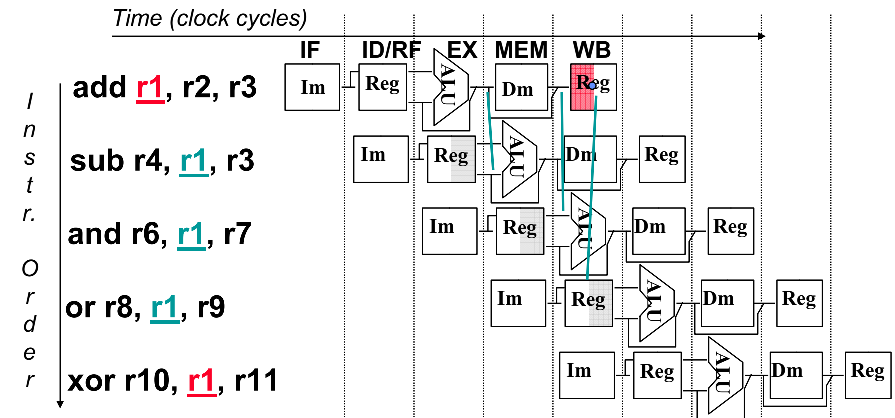
- Stalls can have a significant effect on performance
- Consider the following case
  - The ideal CPI of the machine is 1
  - A RAW hazard causes a 2 cycle stall
- If 40% of the instructions cause a stall?
  - The new effective CPI is $1 + 2 \times 0.4 = 1.8$
  - And the real % is probably higher than 40%
- You get a little more than ½ the desired performance!

Reducing Stalls – one step beyond

- Key is being careful about when you say data is available
- But you really have the value in the machine at end of ALU
  - If you can use this value, the stall for ALU is zero!
    - Fastest, but requires more hardware – called forwarding

Data Hazard - Forward

- "Forward" the data to the appropriate unit

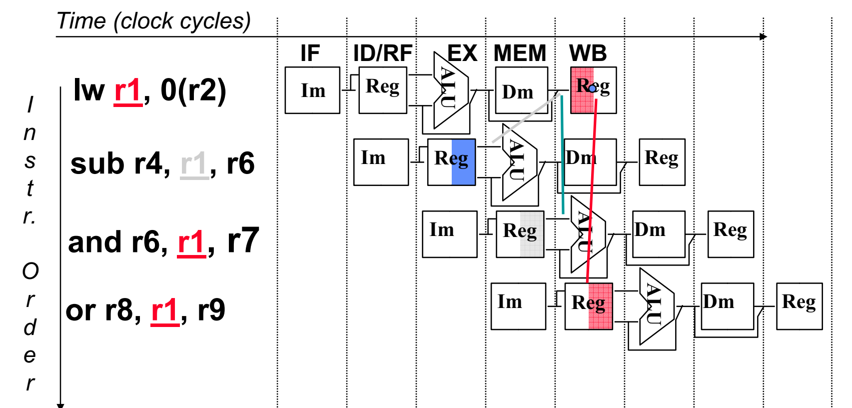


Forwarding Limitations

- Can't forward data you don't have
  - Since ALU complete in a cycle
    - Always have results for the next instruction
  - Memory does not return until the end of MEM
    - That is one cycle later than ALU
    - So next instruction can't get the memory result
      - It is not in the processor yet
    - Must stall the next instruction until the data returns
- With forwarding
  - No ALU-to-ALU delay
  - 1 cycle load-to-ALU delay

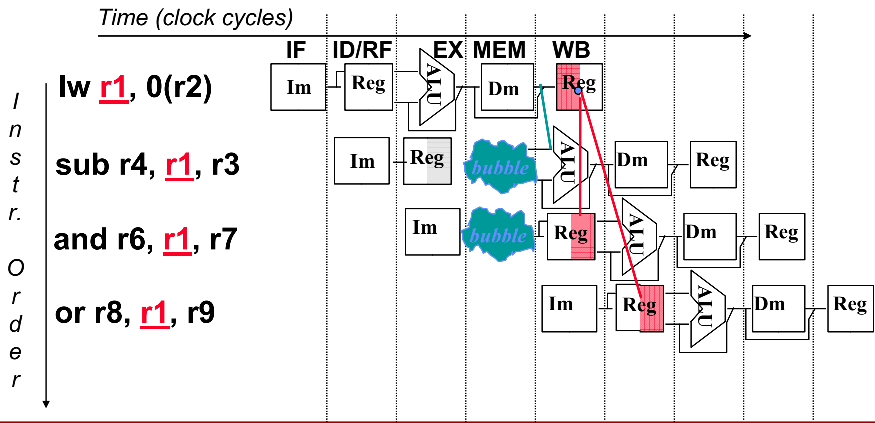
Data Hazard with Forwarding

- Data is not available yet to be forwarded



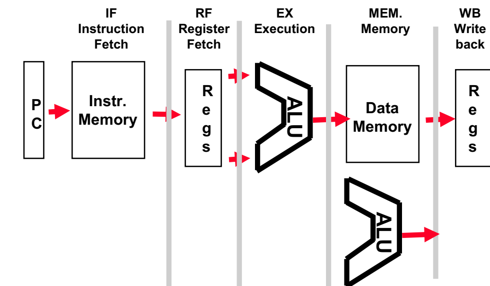
Hardware Stall

- A *pipeline interlock* checks and stops the *instruction issue*



Question

- What happens if we have two ALUs for each instruction
  - One is used to generate effective address (during ALU)
  - One is used to generate results during MEM
- What stalls does this machine have?



New Pipeline

- Old pipeline was:

IF	RF	EX	MEM	WB		
	IF	RF	EX	MEM	WB	
		IF	RF	EX	MEM	WB

  - This meant there was a delay slot after a load instruction

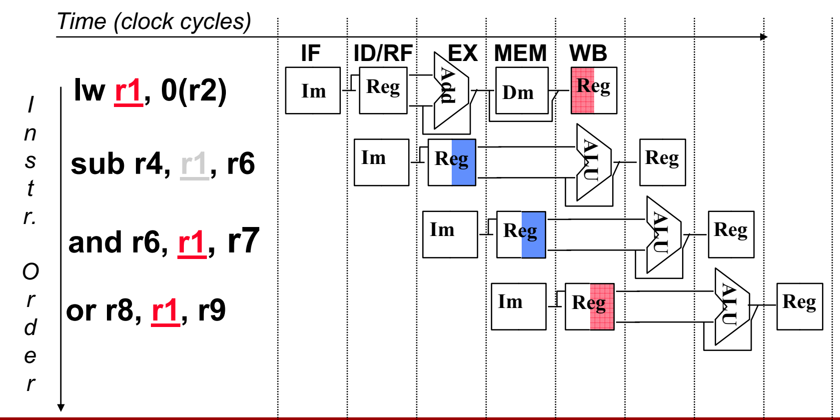
- New pipeline (requires two adders) is

IF	RF	Addr	EX	WB		
	IF	RF	Addr	EX	WB	
		IF	RF	Addr	EX	WB

  - Now memory fetch is performed during EX phase
  - No slot after a load

Data Hazard with Forwarding

- What happens now?



Pipeline Example

- Old pipeline:

lw r4 8(r1)	IF	RF	**EX**	**MEM**	WB	
(slot)		IF	RF	EX	MEM	WB
add r2, r4, r1			IF	RF	**EX**	MEM

- New pipeline (requires two adders) is

lw r4 8(r1)	IF	RF	**Addr**	**EX**	WB
add r2, r4, r1		IF	RF	Addr	EX

Pipeline Example (cont)

- Old pipeline:

add r1, r2, r3	IF	RF	**EX**	MEM	WB	
lw r4 8(r1)		IF	RF	**EX**	**MEM**	WB

- New pipeline (requires two adders) is

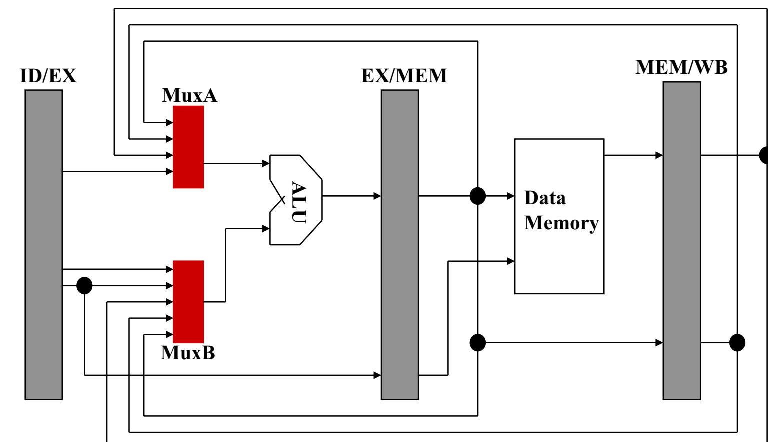
add r1, r2, r3	IF	RF	Addr	**EX**	WB		
(slot)		IF	RF	Addr	EX	WB	
lw r4 8(r1)			IF	RF	**Addr**	**EX**	

- Which is better?
  - Depends on which pair of instructions happens more often

Forwarding Hardware

- What does forwarding cost?
  - Need to add stuff to datapath and stuff to control
- Datapath
  - Need to add multiplexers to functional units
  - Source to function unit could come from
    - Register file
    - Memory
    - ALU of last cycle
    - ALU from two cycles ago
  - Adding this mux increases the critical path of design
    - Needs to be designed carefully

Forward Hardware - Datapath



Forward Hardware - Control

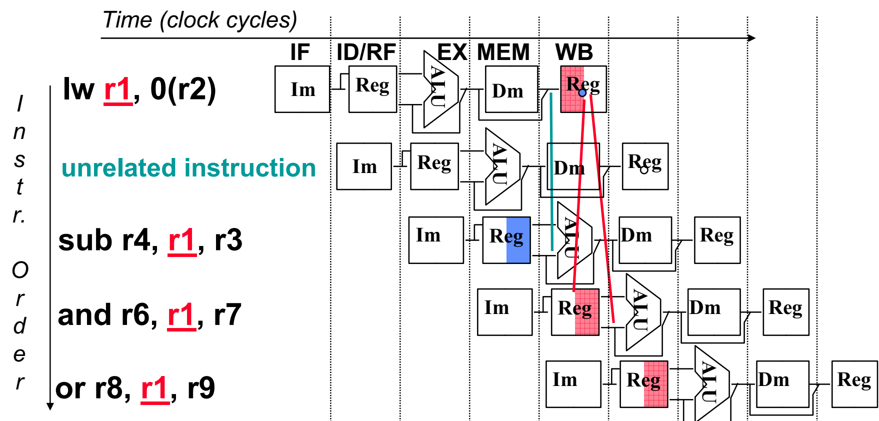
- Need to decide which multiplexer input to enable
  - Doesn't seem that hard but it can get troublesome
    - Especially with machines that issue multiple instructions/cycle
- Which is the correct result
  - Need to tag ALU, MEM results with registerID
  - Need to compare register fetch with tags
    - All this takes hardware, but can be done in parallel
  - Need to find youngest version of the register
    - Multiple tags can match
    - Need to find freshest version of the data

Compilers

- Compilers rearrange code to try to fill slots with useful stuff
  - Fill load delay slot with a good instruction
  - When successful, the slot has no cost
    - The next instruction does not depend on load result
    - Does not need to stall
    - Show the advantage of the pipeline
  - When can't fill the slot
    - Need to output a NOP if there is no hardware interlock
- Since the pipeline is very machine dependent
  - Need hardware interlocks to run old code
  - Most machines have interlocks!
  - Microprocessor without Interlocked Pipeline Stages

Rearranged Code

- Compiler inserts independent instruction



Control Hazard

- The missing data so far has been a result
  - Used by another instruction
- What happens when the missing data is the instruction pointer?
  - This is called a control hazard
- Control hazards:
  - Branch instruction
    - If a branch is not taken then control simply continues with PC + 4
    - If the branch is taken, then the PC jumps to a new address
      - Whether the branch is to be taken, however, is now known until the 3rd (MEM) stage
  - Jump instruction
- Causes a greater performance problem than data hazards
  - Instruction fetch happens very early in the pipeline