

---

## Lecture 10

# Control Hazards and Advanced Pipelining

Christos Kozyrakis

Stanford University

<http://eeclass.stanford.edu/ee108b>

# Announcements

---

- Announcements will be made through email

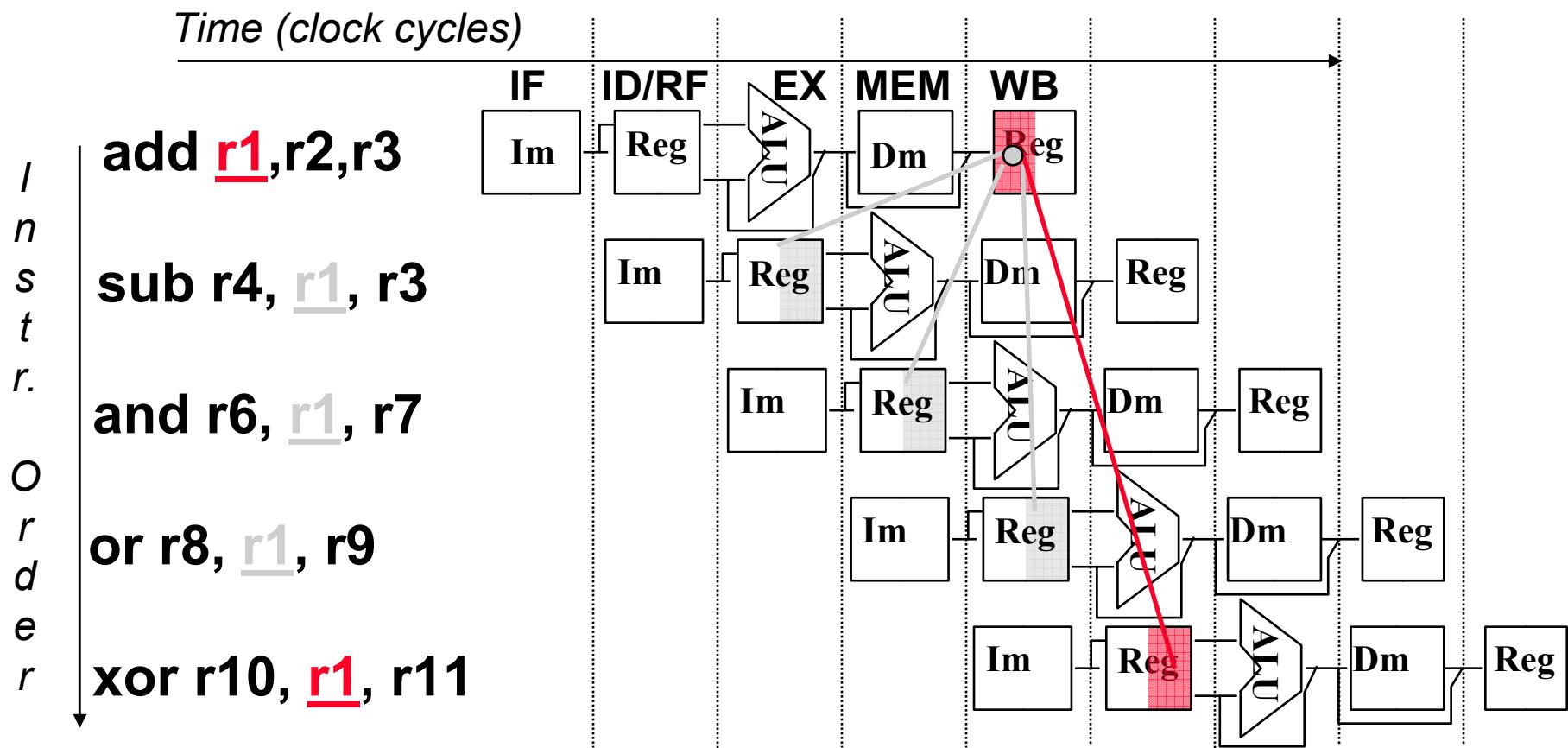
# Review: Pipeline Hazards

---

- These are dependencies between instructions that are exposed by pipelining
  - Causes pipeline to lose 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

# Review: Data Hazard Example

- Dependencies forwards in time are hazard



# 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

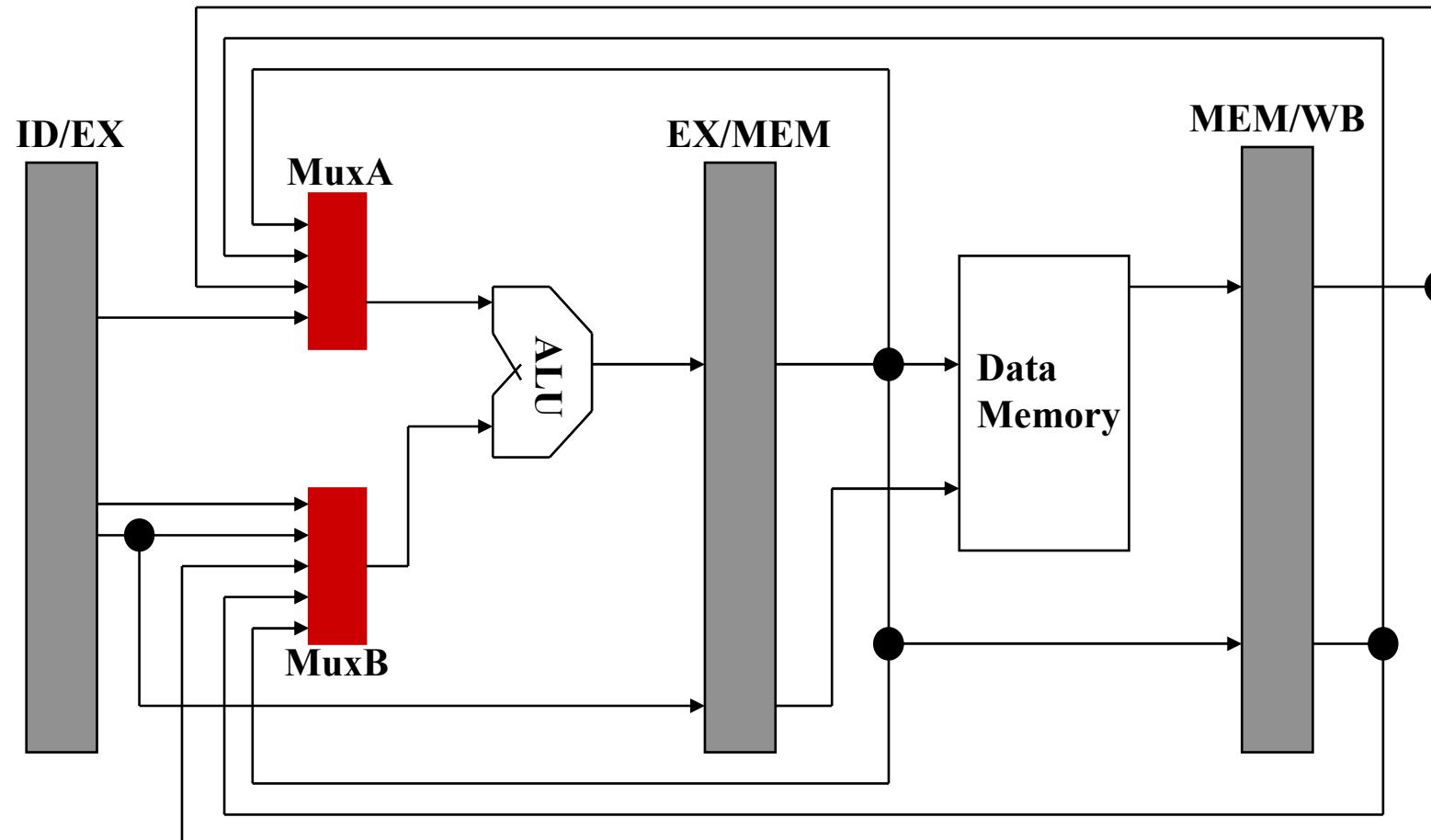
# Discovering Forwarding Paths in Pipelines

---

- Can get out of hand if not carefull
- Simple procedure
  - Identify all pipeline stages that produce new values
    - In our case, EX and MEM
  - All pipeline stages after the earliest producer can be the source of a forwarded operand
    - In our case, MEM
  - Identify all pipeline stages that really consume values
    - In our case, EX and MEM
  - These stages are the destinations of a forwarded operand
  - Add multiplexor for each pair of source/destination stages

# Forward Hardware - Datapath

---

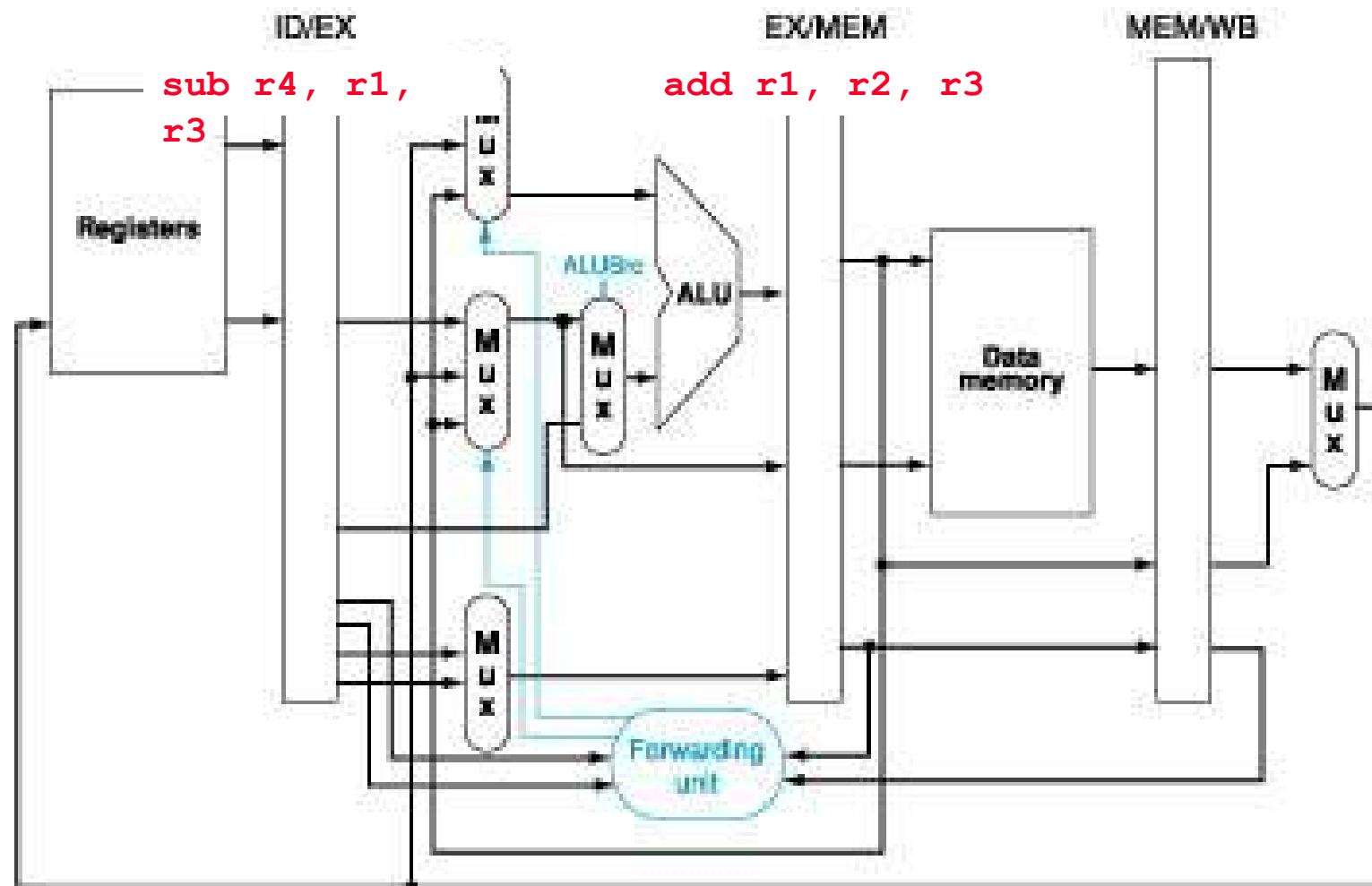


# Forward Hardware - Control

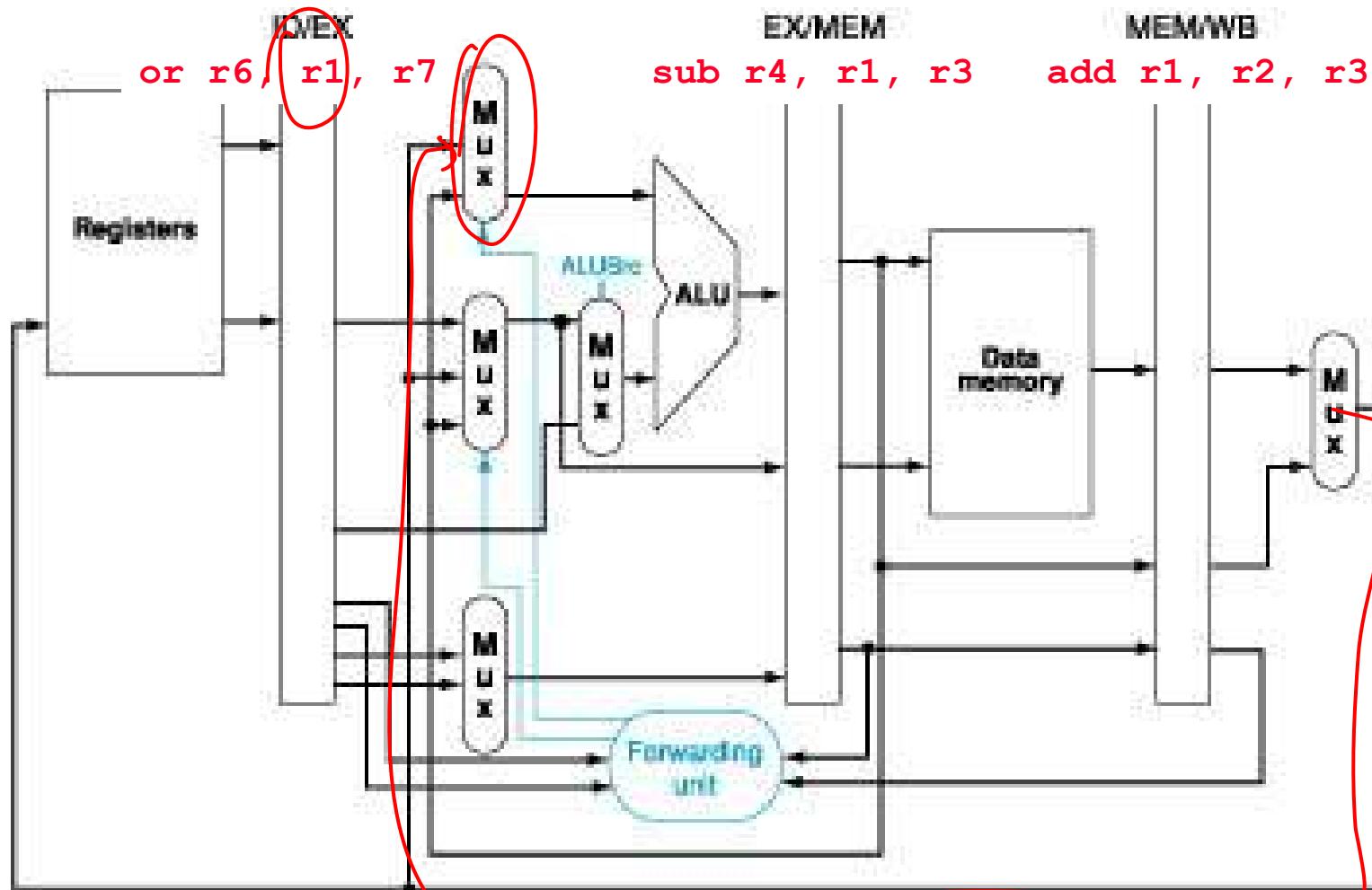
---

- 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

# Forward Hardware - Datapath & Control 1

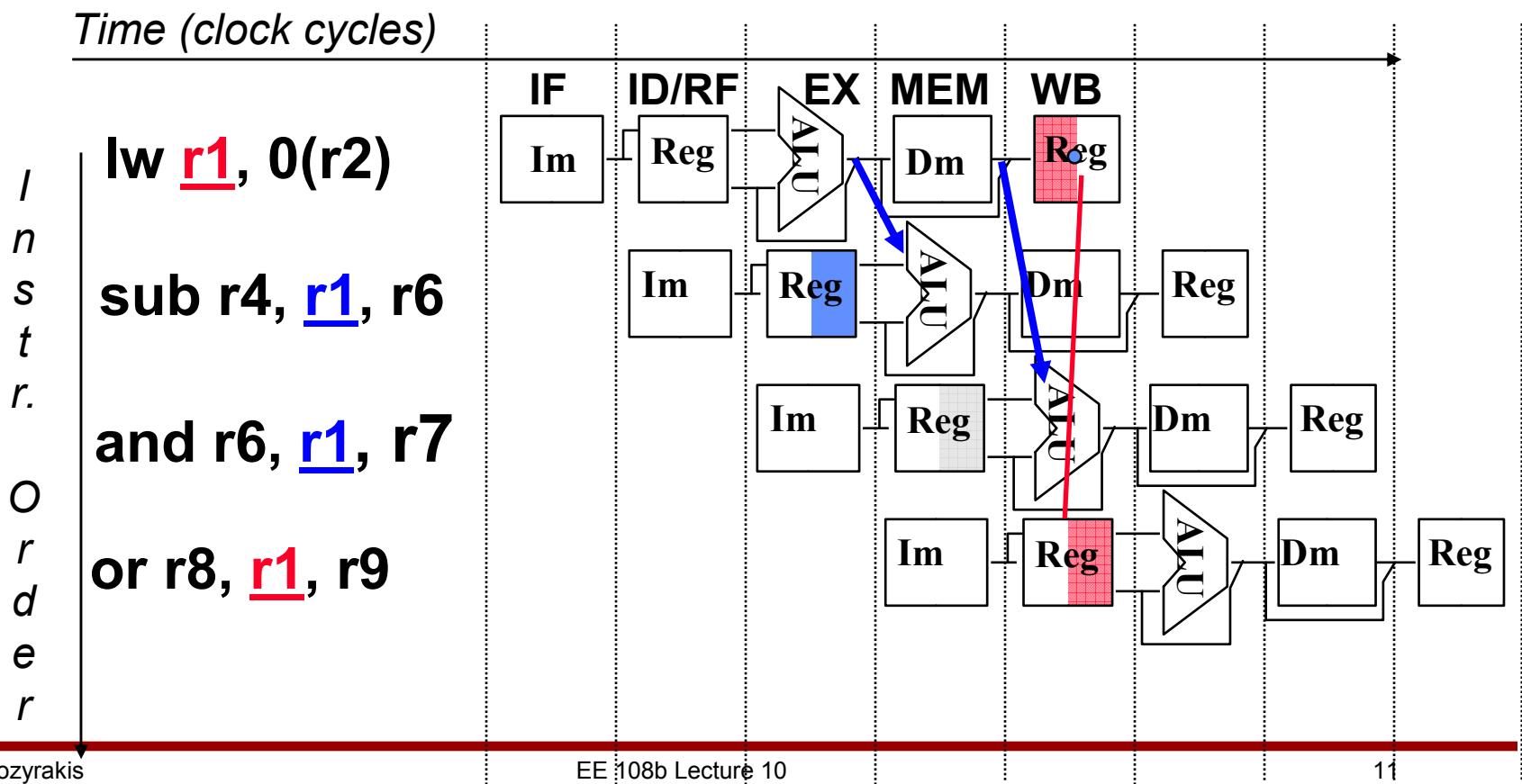


# Forward Hardware - Datapath & Control 2



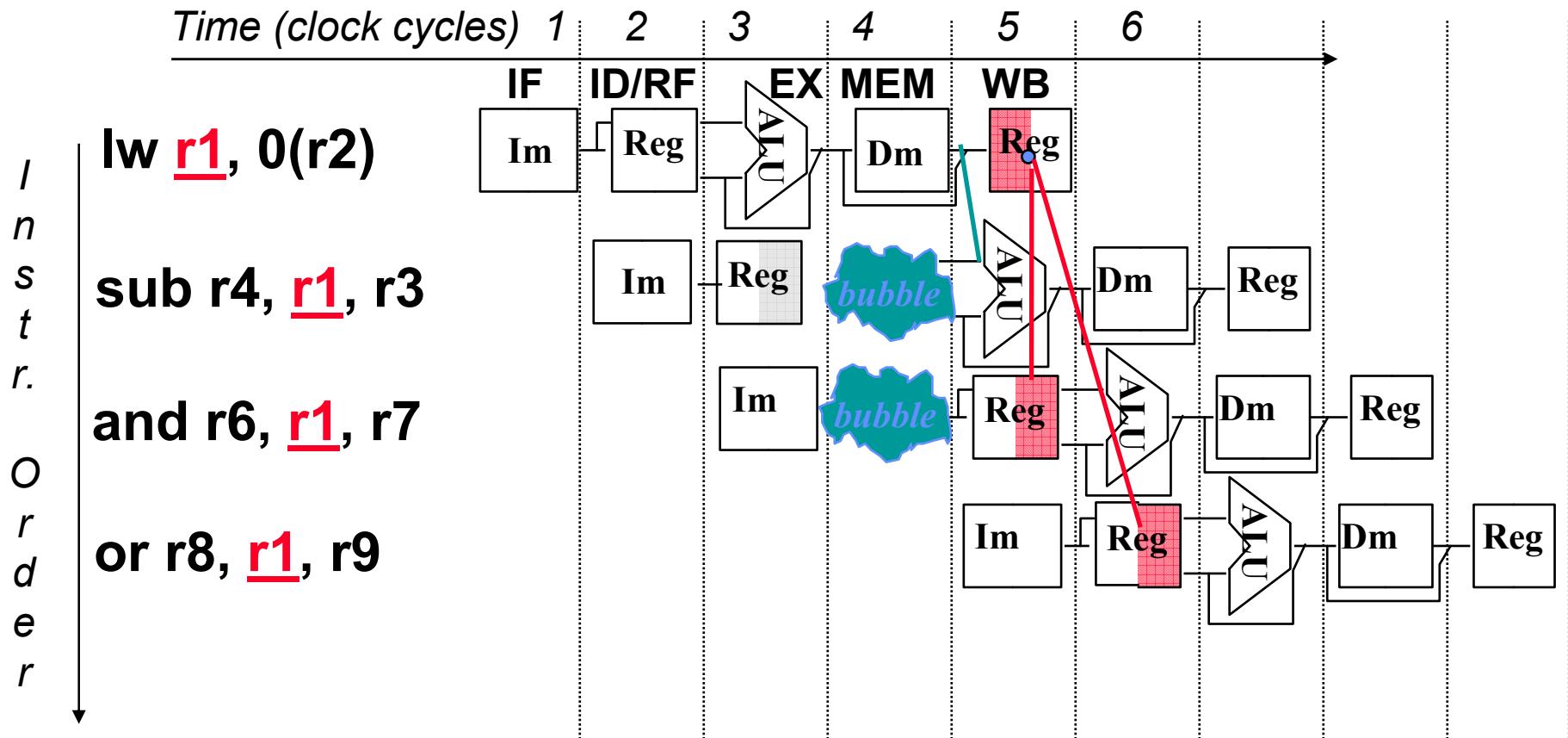
# Review: Data Hazard with Forwarding

- Data is not available yet to be forwarded

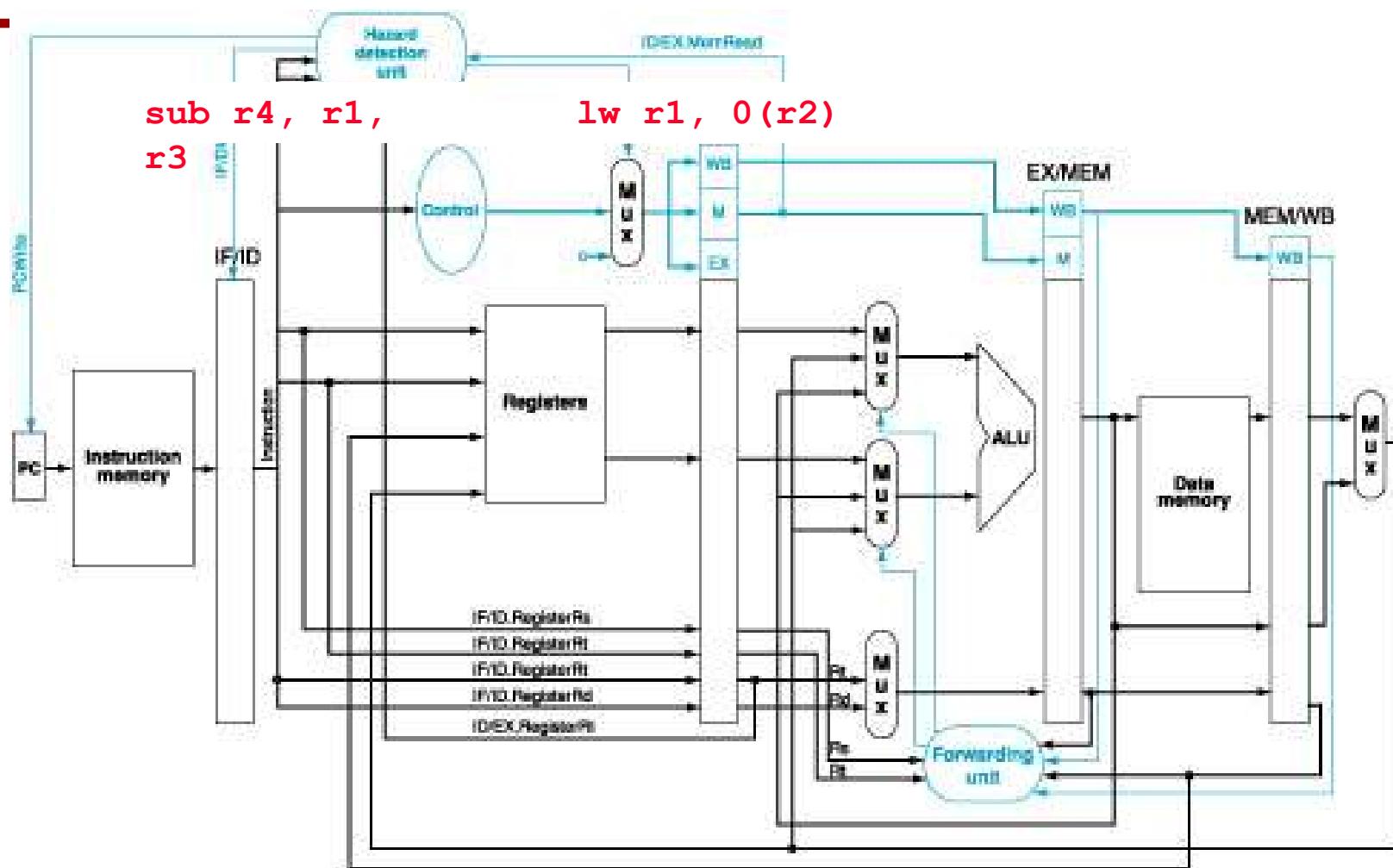


# Review: Hardware Stall

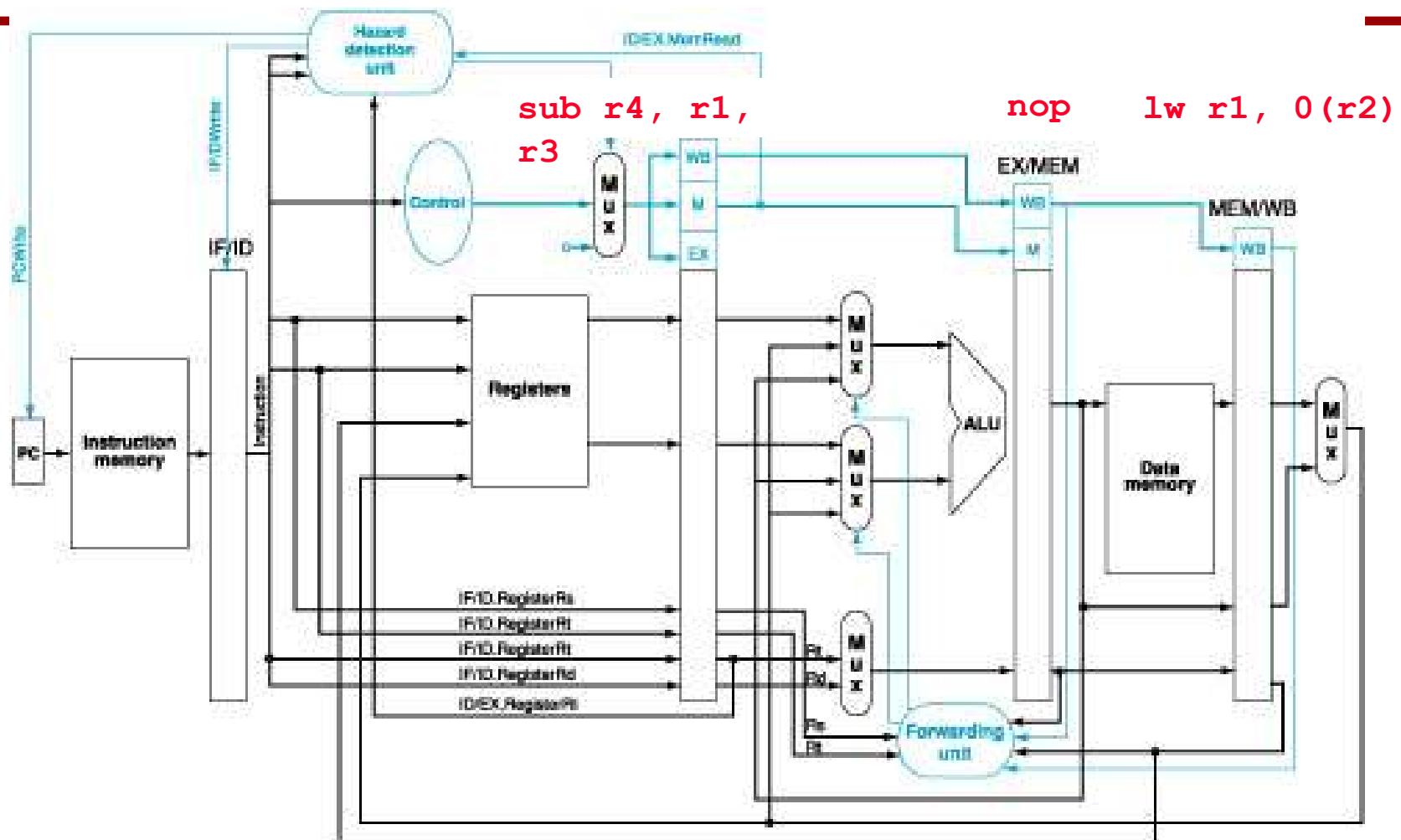
- A *pipeline interlock* checks and stops the *instruction issue*



# Hazard Detection 1



# Hazard Detection 2 (2 cycles later)



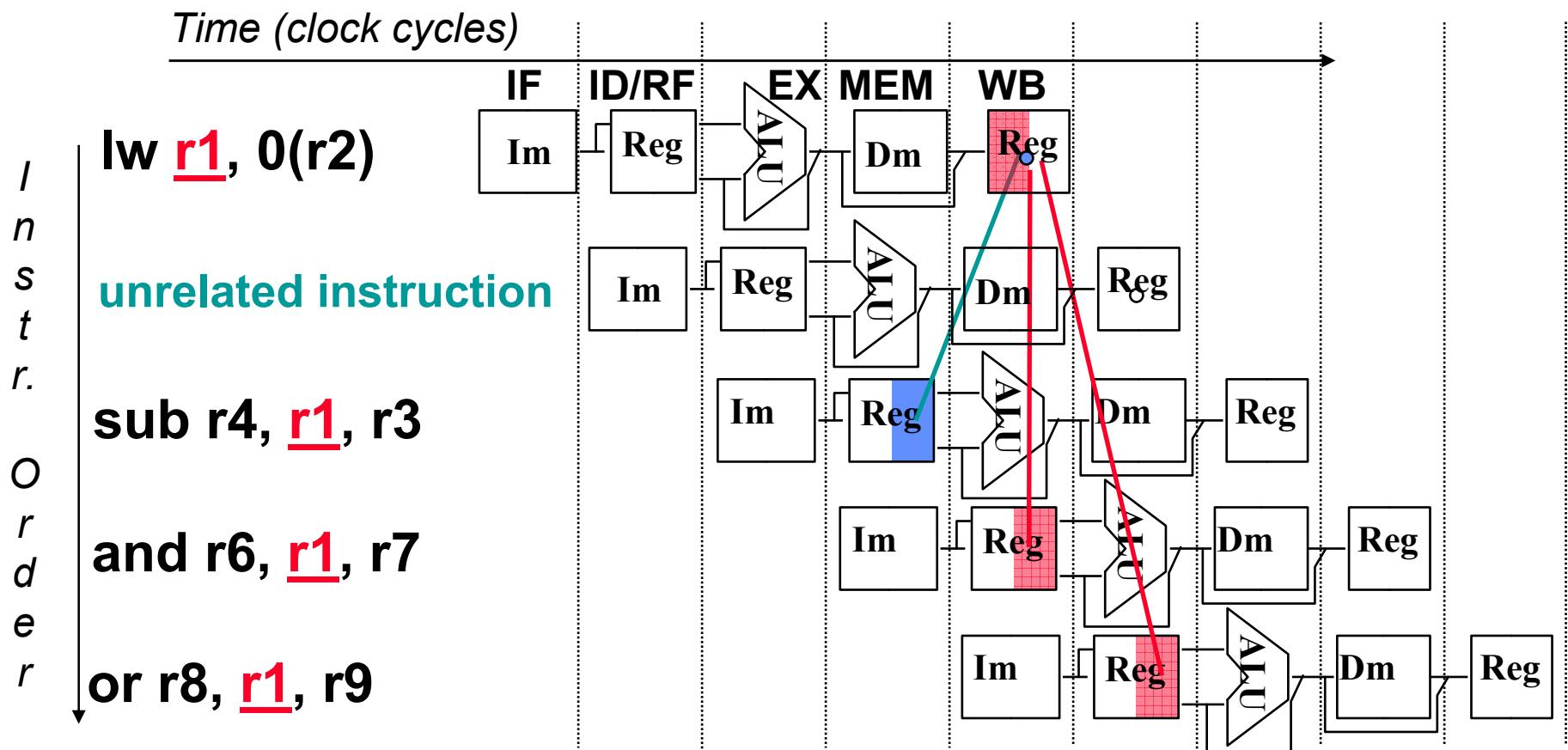
# Compilers & Data Hazards

---

- 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

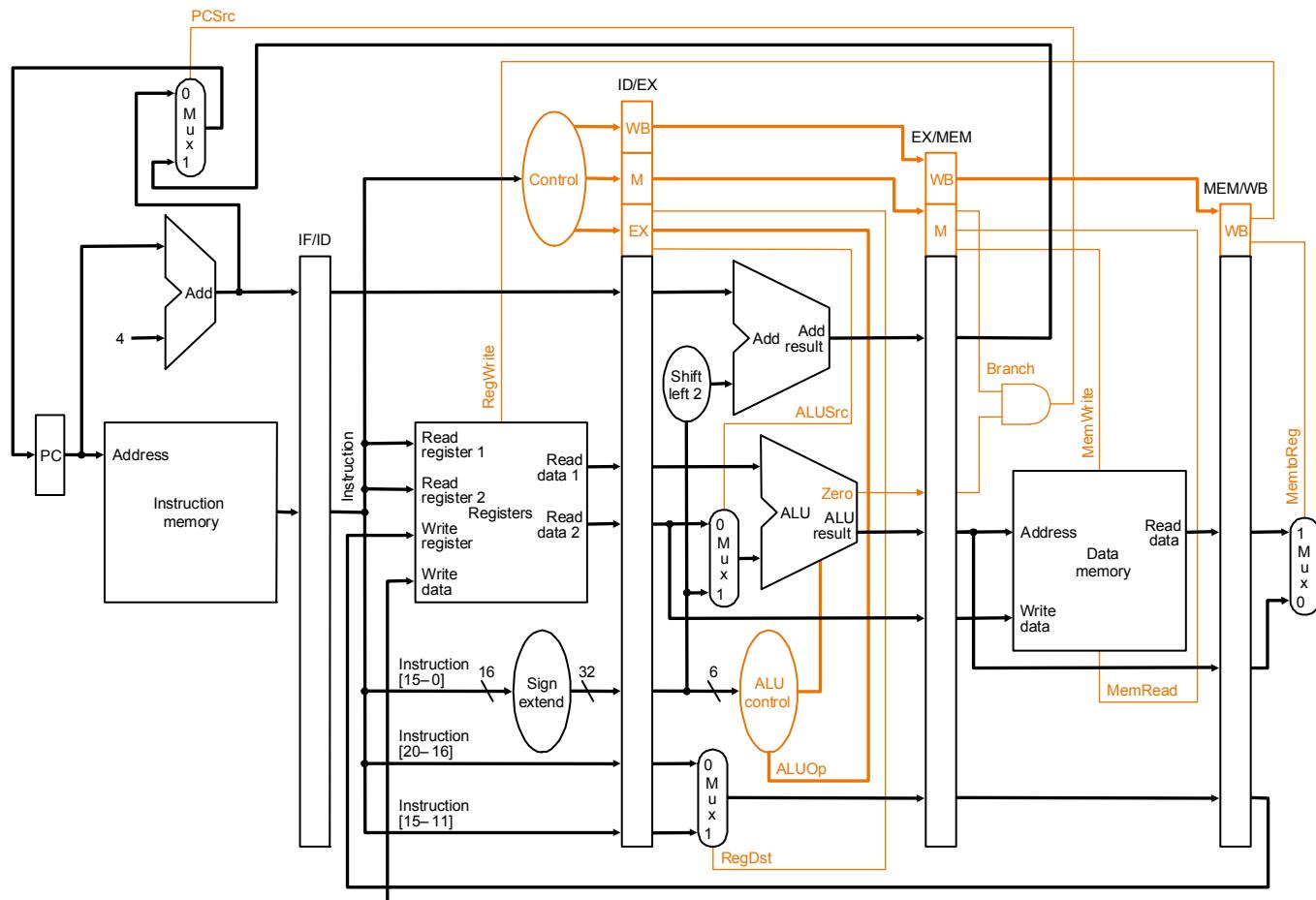


# Control Hazard

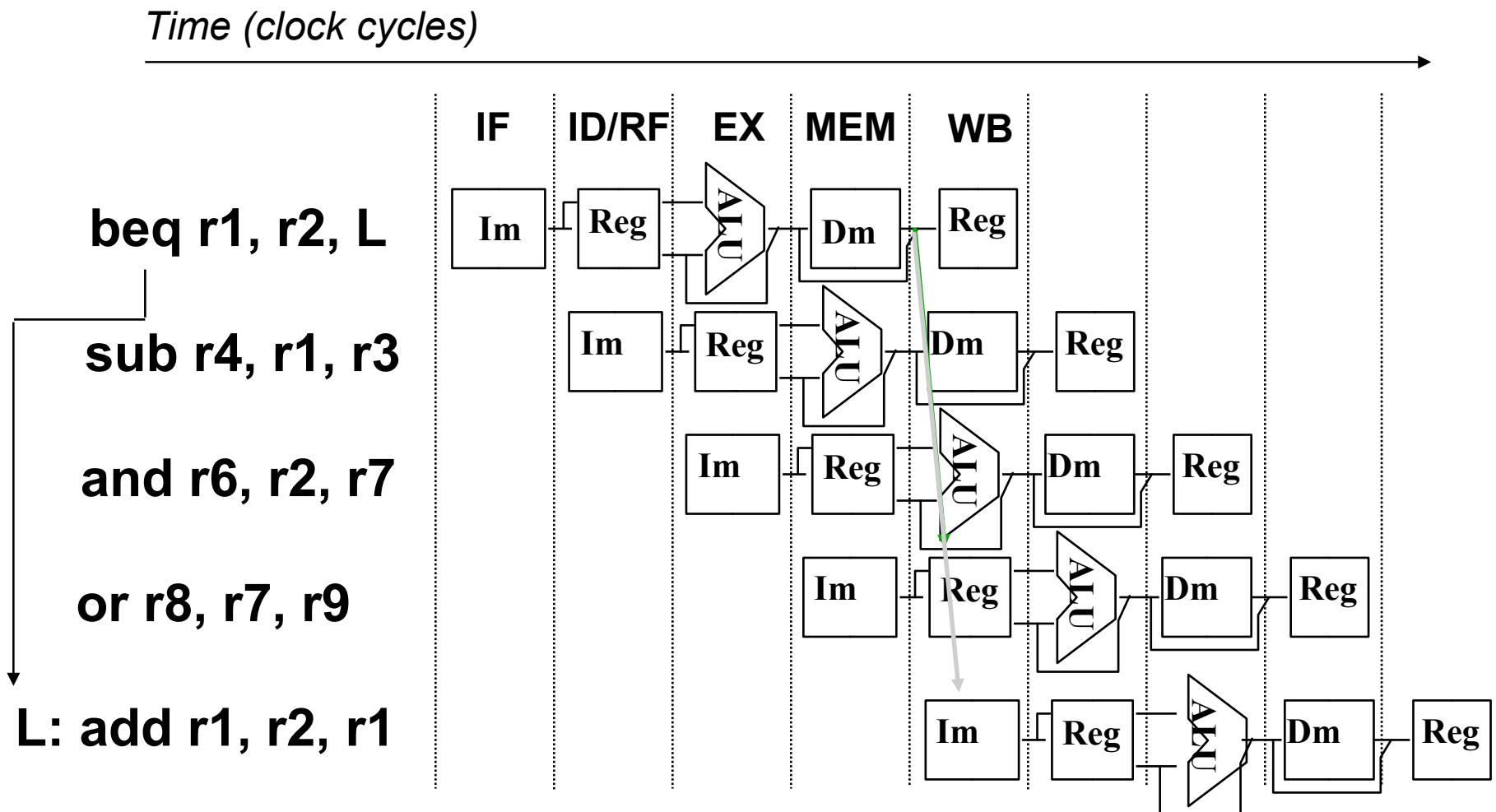
---

- Data Hazard caused by missing data
  - Used by another instruction
- What happens when the missing data is the next PC?
  - 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
  - Jump instruction
- Causes a greater performance problem than data hazards
  - Instruction fetch happens very early in the pipeline

# Pipeline so far (without forwarding and hazard detection)



# Branch Control Hazard

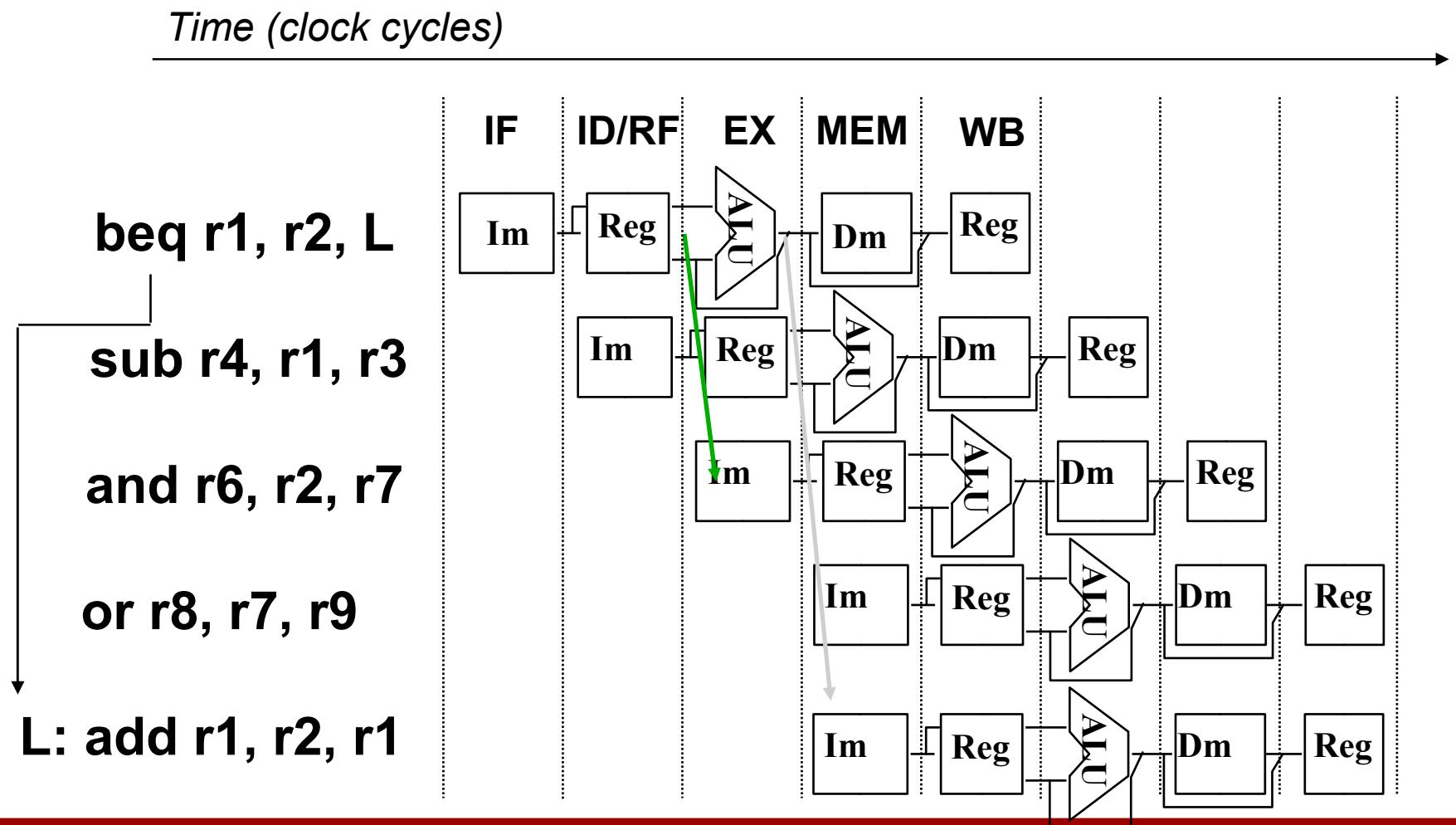


# Control Hazard Solutions

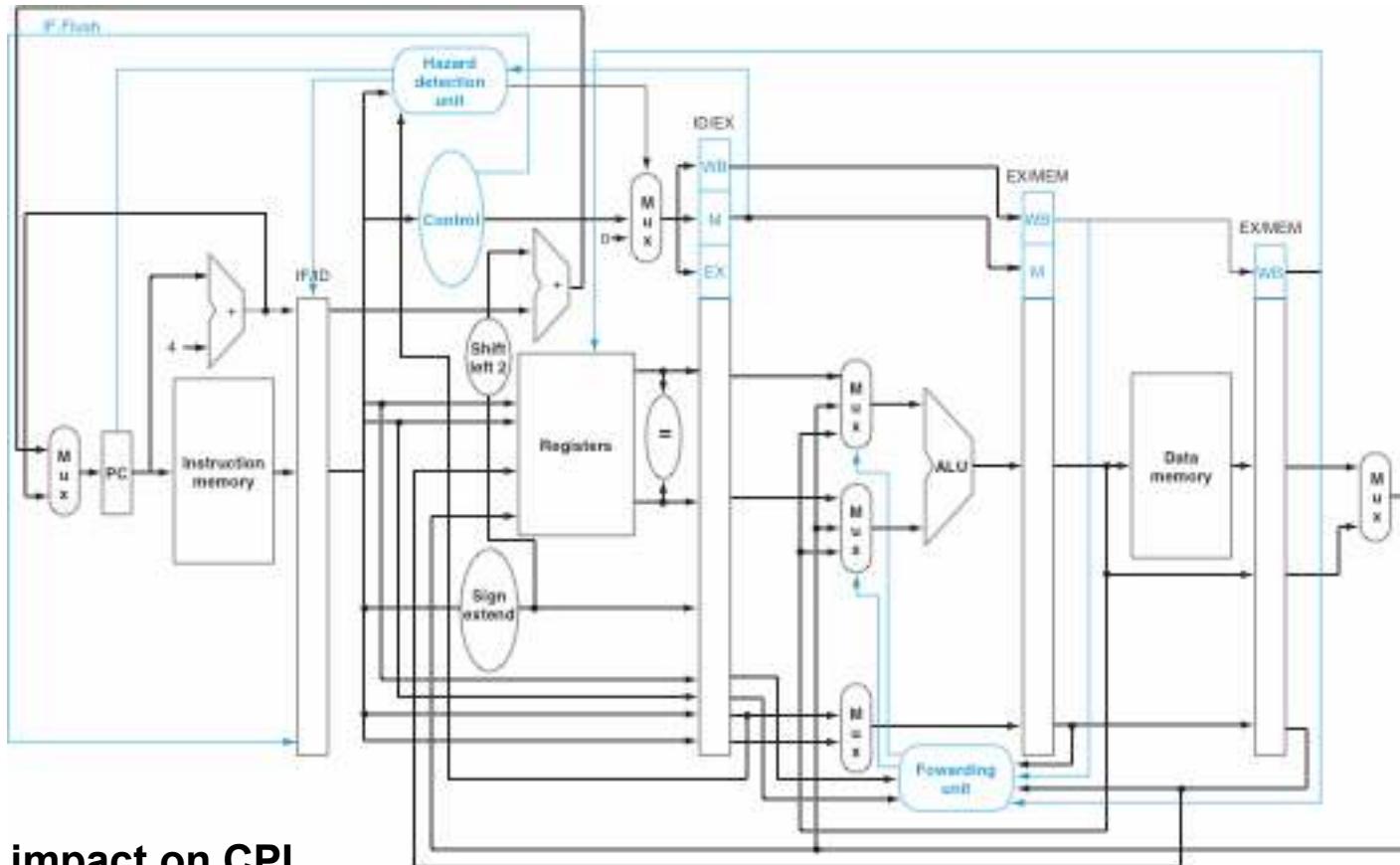
---

- Stall
  - Wait until you know the answer
  - But makes CPI of branch large (about 3)
- Predict not-taken
  - Continue to fetch and execute instruction
  - Need to be able to nullify instruction if prediction was wrong
    - Can be tricky, but is not that hard if done correctly
- Predict taken
  - More complex, since need to compute destination
    - Generally this takes some time, so
    - Store destination address prediction with instruction
  - Still need to nullify when wrong
- Most machines do some type of prediction (taken, not-taken)

# Branch Control Hazard: Another look (Moving the control point)



# Reducing Branch Delay



**What is impact on CPI and CCT?**

# Branch Stall

---

- Need to stall for one cycle on every branch!
  - ID control point
- Consider the following case
  - The ideal CPI of the machine is 1
  - The branch causes a stall
- What is the new effective CPI if 15% of the instructions are branches?
- The new effective CPI is  $1 + 1 \times 0.15 = 1.15$
- The old effective CPI was  $1 + 3 \times 0.15 = 1.45$

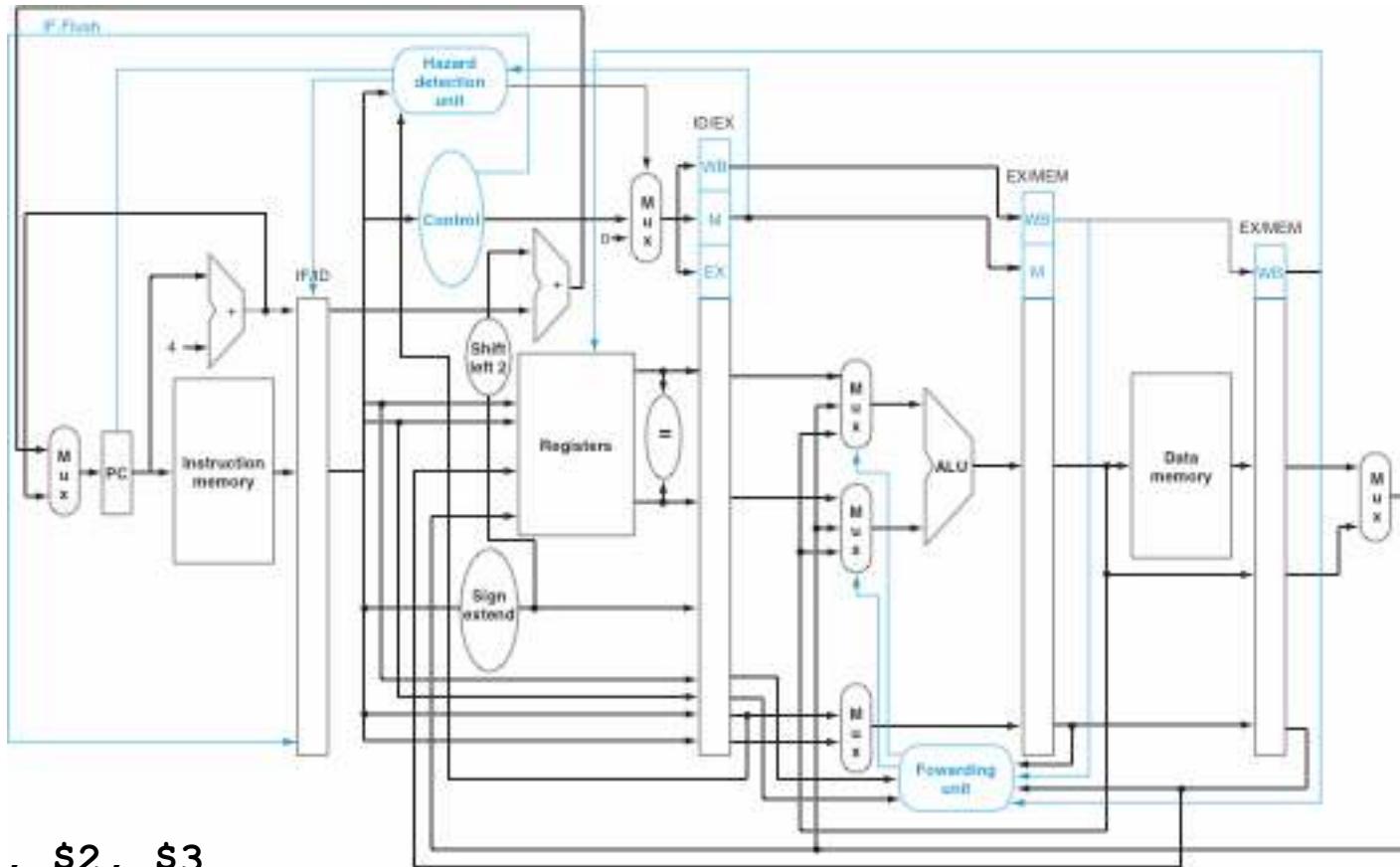
# Delayed Branches

---

- Solution used in early MIPS machines
  - Had a branch delay of one
  - Branch does not take effect until the cycle after its execution
- Example:

beq r1, r2, L	Branch instruction
sub r4, r1, r3	This operation ALWAYS is executed
and r6, r2, r7	This operation executes if branch fails
- Worked well initially, but now is a pain
  - Compiler can fill one slot 50% of the time
  - Machine have many branch delay slots
  - Issue more than one instruction per cycle
  - Modern machines use branch prediction

# Reducing Branch Delay



# Exceptions

---

- Something bad happens to an instruction
  - Need to stop the machine
  - Fix up the problem
  - Start the machine up again
- Definition: precise exceptions
  - All previous instructions had completed
  - The faulting instruction was not started
  - None of the next instructions were started
    - No changes to the architecture state (registers, memory)
- With a single cycle machine, this is easy
  - Why?

# Preview: Exceptions and Pipelining

---

- With a pipelined machine it is a little more complex
  - Need to make sure you can stop the machine precisely
    - Instructions  $< j$  complete,  $>j$  have no effect on machine
- Most instructions results only go to register file
  - If we prevent their write into the registers
    - Need to prevent results from bypassing too
  - It is like they did not exist
- Complexity comes from instructions that change state earlier
  - Like branches and jumps
  - Need to store old PC values and restore them on exceptions

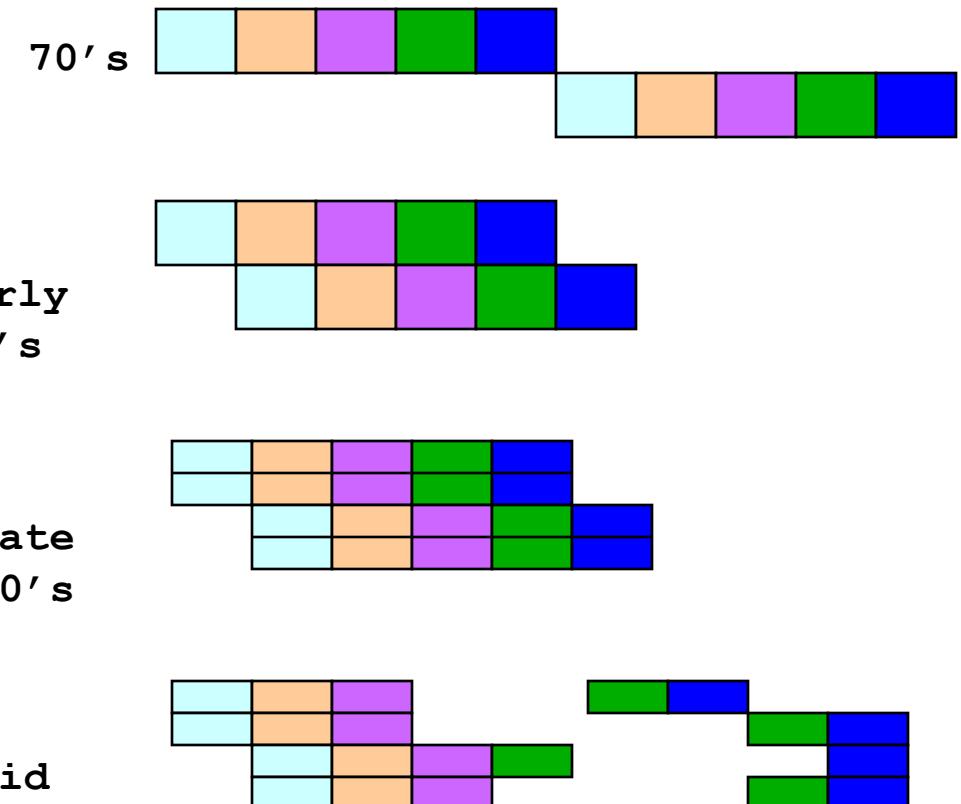
# Overview of First Generation RISC

---

- Used static scheduling
  - Instructions were ordered by the compiler
  - Hardware does not reorder instructions
- Register write occurs in the first part of the clock cycle
  - Reads are performed in the second half of the clock cycle
  - Simplifies some hazards and eliminates others
- Memory access occurs in the 4th stage
  - Avoids all memory hazards
- RAW hazards use forwarding, except on load results
  - Loads resolved by load delay and stalls
- Control hazards use delayed branch
- Good pipeline performance
  - CPI = 1.1 – 1.5 on integer programs

# Computer Architect's Job

- Convert transistors to performance
- Use transistors to
  - Exploit parallelism
  - Or create it (speculate)
- Processor generations
  - Simple machine
    - Reuse hardware
  - Pipelined
    - Separate hardware for each stage
  - Super-scalar
    - Multiple port mems, function units
  - Out-of-order
    - Mega-ports, complex scheduling
  - Speculation
- Each design has more logic to accomplish same task (but faster)



# Advanced Pipelining

---

- Where have all the transistors gone?
  - MIPS R3000 : 120 thousand transistors
  - Intel Pentium 4 : 160 Million transistors
  - Many transistors in the cache
- Fancy techniques to decrease CPI and increase clock frequency
  - Superscalar (multiple instruction execution)
  - Deep pipelining
  - Dynamic scheduling (out-of-order execution)
  - Dynamic branch prediction
  - Register renaming
- All pipelining techniques exploit instruction-level parallelism (ILP)

# What is ILP?

---

- Independence among instructions

- Example with ILP

- add \$t0, \$t1, \$t2  
or \$t3, \$t1, \$t2  
sub \$t4, \$t1, \$t2  
and \$t5, \$t1, \$t2

**ILP in real programs  
is limited**

- Example with no ILP

- add \$t0, \$t1, \$t2  
or \$t3, \$t0, \$t2  
sub \$t4, \$t3, \$t2  
and \$t5, \$t4, \$t2

# Superscalar

---

- Fetch and execute multiple instructions per cycle => CPI < 1
- Example: 2-way Superscalar
  - Fetch 2 instructions/clock cycle
  - Decision to execute two instructions handled dynamically
  - Can only execute 2nd instruction if 1st instruction executes (in-order exec.)

<u>Pipe Stages</u>					
	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB
<b>Ideal CPI =</b>	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB
		IF	ID	EX	MEM
		IF	ID	EX	WB

- Resources: double amount of hardware (FUs, Register file ports)
- Issues: hazards, branch delay, load delay

# Deeper Pipelining

---

- Increase number of pipeline stages
  - Fewer levels of logic per pipe stage
  - Higher clock frequency
- Example 9-stage pipeline

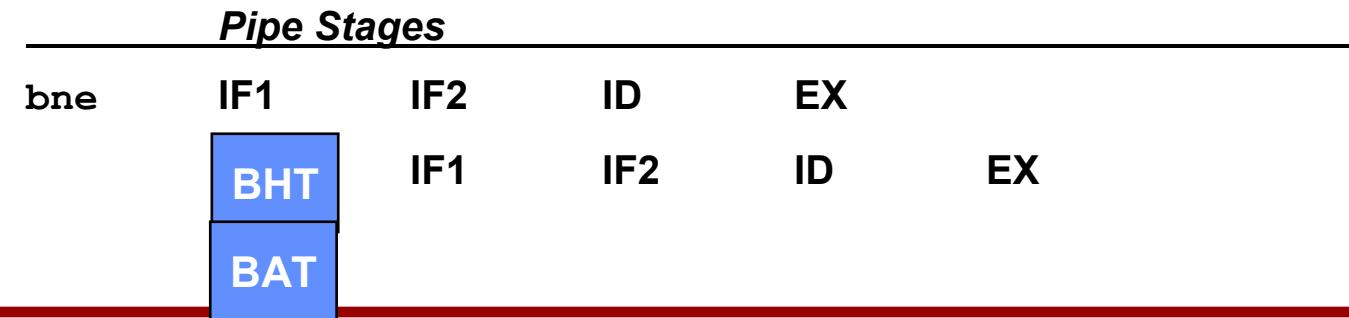
## Pipe Stages

IF1	IF2	ID	EX	MEM1	MEM2	MEM3	WB	
IF1	IF2	ID	EX	MEM1	MEM2	MEM3	WB	
• Almost double number of pipe stage registers (MIPS R4000)								
• Issues: branch delay, load delay: CPI = 1.4 - 2.0								
• Modern pipelines								
– Pentium 4: 24 stages, 3+ GHz								
– Prescott: 35 stages, 4+ GHz								

# Dynamic Branch Prediction

---

- Predict direction of branches based on past behavior
  - Maintain a table of branch behavior and look up to get prediction
- *Branch prediction buffer* (or *branch history table BHT*)
  - Lower bits of PC address index table of 1 bit values
  - Says whether or not branch taken last time
  - Evaluate actual branch condition and correct if incorrect
    - Recover by flushing pipeline and restarting fetch
    - Reset prediction
  - Branch prediction using 2 bits often more accurate



# Dynamic Scheduling

---

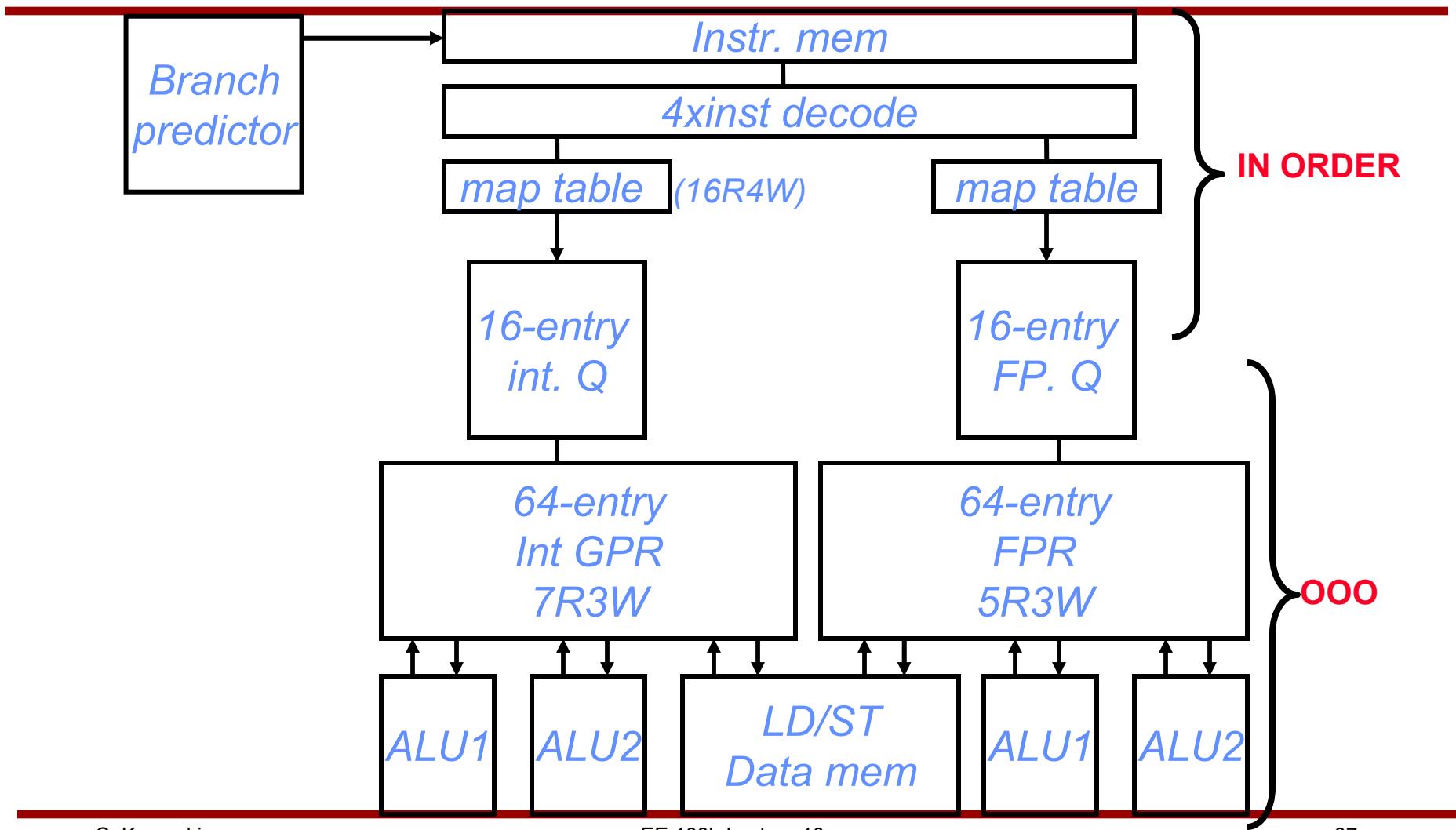
- Execute instructions out-of-order
- Fetch multiple instructions per cycle into instruction queue using branch prediction
- Figure out which are independent and execute them in parallel
- Example
  - add \$t0, \$t1, \$t2
  - or \$t3, \$t0, \$t2
  - sub \$t0, \$t1, \$t2
  - and \$t5, \$t0, \$t2
- Superscalar + Dynamic scheduling
  - add \$t0, \$t1, \$t2              sub \$t0, \$t1, \$t2
  - or \$t3, \$t0, \$t2              and \$t5, \$t0, \$t2
- What's wrong with this?

# Register Renaming

---

- Rename (map) architectural registers to physical registers in decode stage to get rid of false dependencies
- add \$t0, \$t1, \$t2  
or \$t3, \$t0, \$t2  
sub \$t0, \$t1, \$t2  
and \$t5, \$t0, \$t2
- Superscalar + Dynamic scheduling + register renaming
- add \$t0<sub>A</sub>, \$t1, \$t2      sub \$t0<sub>B</sub>, \$t1, \$t2  
or \$t3, \$t0<sub>A</sub>, \$t2      and \$t5, \$t0<sub>B</sub>, \$t2
- Need more physical registers than architectural registers
- Physical registers are automatically recycled

# Dynamic Scheduling in a Modern OOO MIPS R10000



# Limits of Advanced Pipelining

---

- Limited ILP in real programs
- Pipeline overhead
- Limited branch prediction accuracy (85%-98%)
- Memory inefficiency
- Complexity of implementation

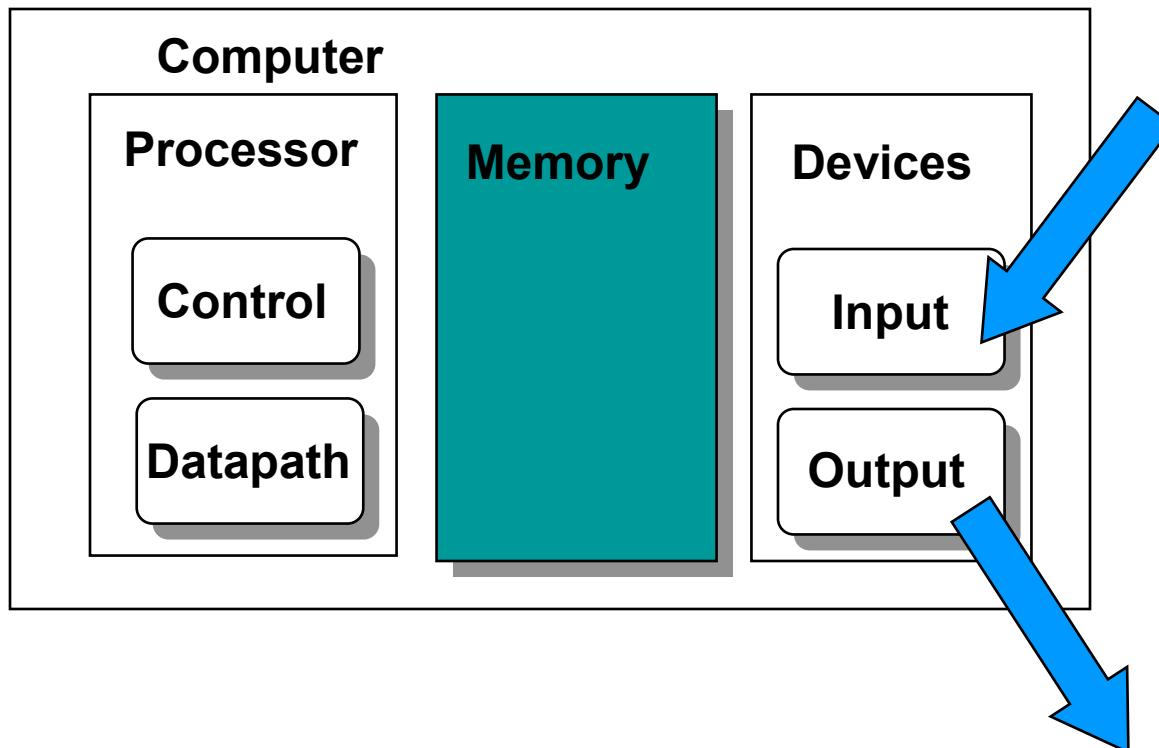
# Are We Done With Hardware?

---

- Unfortunately (or fortunately depending on your view) no
  - Still need to talk about the other main hardware piece
  - Memory
- Why talk about memory?
  - Isn't it just a large array of bits?
    - Used to be, but not any more
- Problem:
  - Typically want a lot of memory in your machine
    - Usually hundreds of dollars
    - Since it is expensive, you want the most bits/\$
  - Since the memory price is #1 issue, you have other issues

# Five Components

---



- Datapath
- Control
- **Memory**
- Input
- Output