
EECS 150 -- Digital Design

Lecture 11-- Processor Pipelining

2010-2-23

John Wawrzynek

Today's lecture by John Lazzaro

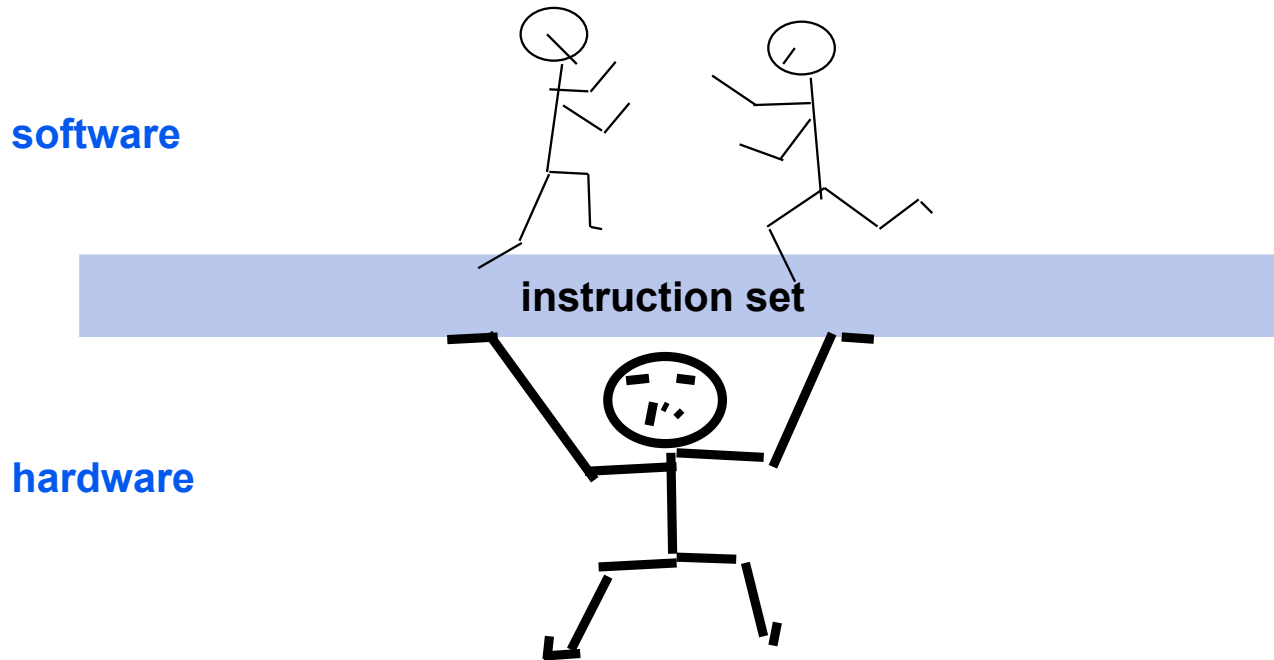
www-inst.eecs.berkeley.edu/~cs150



Today: Pipelining

- * How to apply the **performance equation** to our single-cycle CPU.
- * **Pipelining**: an idea from assembly line production applied to CPU design
- * Why pipelining is **hard**: data hazards, control hazards, structural hazards.
- * **Visualizing** pipelines to evaluate hazard detection and resolution.
- * A tool kit for hazard **resolution**.

New successful instruction sets are rare



Implementors suffer with original sins of ISAs,
to support the installed base of software.

define: The Architect's Contract

- * To the **program**, it **appears** that instructions execute in the correct order defined by the ISA.
- * As each instruction completes, the machine state (regs, mem) **appears** to the **program** to obey the ISA.
- * What the machine actually **does** is up to the hardware designers, as long as the **contract is kept**.

Goal: Keep contract and run programs faster.



Performance Measurement

(as seen by a CPU designer)

Q. Why do we care about a program's performance?

A. We want the CPU we are designing to run it well !



Step 1: Analyze the right measurement!

Guides
CPU
design

CPU Time:

Time the CPU spends running program under measurement.

Measuring CPU time (Unix):

% time <program name>

25.77u 0.72s 0:29.17 90.8%

Guides
system
design

Response Time:

Total time: CPU Time + time spent waiting (for disk, I/O, ...).



CPU time: Proportional to Instruction Count

Q. Once ISA is set, who can influence instruction count?

A. Compiler writer, application developer.

**Q. Static count? (lines of program printout)
Or dynamic count? (trace of execution)**

A. Dynamic.

$$\frac{\text{CPU time}}{\text{Program}} \propto \frac{\text{Machine Instructions}}{\text{Program}}$$

Rationale: Every additional instruction you execute takes time.

Q. How does an architect influence the number of machine instructions needed to run an algorithm?

A. Create new instructions: instruction set architect.



CPU time: Proportional to Clock Period

Q. How can architects (not technologists) reduce clock period?

A. Shorten the machine's critical path.

Q. What ultimately limits an architect's ability to reduce clock period?

A. Clock-to-Q, setup times.

$$\frac{\text{Time}}{\text{Program}} \propto \frac{\text{Time}}{\text{One Clock Period}}$$

Rationale:

**We measure each instruction's execution time in "number of cycles".
By shortening the period for each cycle, we shorten execution time.**



Completing the performance equation

What factors make different programs have different CPIs?

- Cache behavior varies.
- Instruction mix varies.
- Branch prediction varies.

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \frac{\text{Cycles}}{\text{Instruction}} \frac{\text{Seconds}}{\text{Cycle}}$$

We need all three terms, and only these terms, to compute CPU Time!

“CPI” -- The Average Number of Clock Cycles Per Instruction
 → For the Program ←



When is it OK to compare clock rates?

Consider Lecture 10 single-cycle CPU ...

Thr 2/18

Lec #10: Project Introduction: Serial I/O MIPS Microarchitecture:
[\[PDF\]](#)

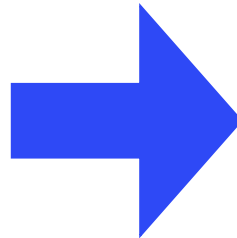
- * All instructions take **1 cycle** to execute every time they run.
- * CPI of **any program** running on machine? **1.0**

“average CPI for the program” is a more-useful concept for more complicated machines ...



Consider machine with a data cache ...

A program's load instructions "stride" through every memory address.



The cache never "hits", so every load goes to DRAM (100x slower than loads that go to cache).

Thus, the average number of cycles for load instructions is higher for this program.

Thus, the average number of cycles for all instructions is higher for this program.

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \frac{\text{Cycles}}{\text{Instruction}} \frac{\text{Seconds}}{\text{Cycle}}$$

Thus, program takes longer to run!



Final thoughts: Performance Equation

Seconds
Program

↑
Goal is to optimize execution time, not individual equation terms.

=

Instructions
Program

↑
Machines are optimized with respect to program workloads.

Cycles
Instruction

↑
The CPI of the program. Reflects the program's instruction mix.

Seconds
Cycle

↑
Clock period. Optimize jointly with machine CPI.

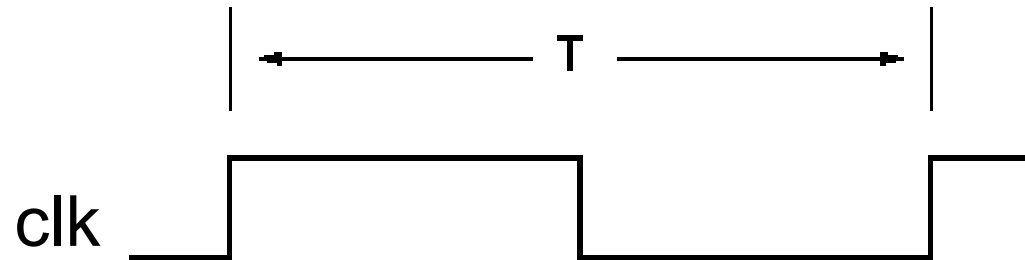


Pipelining



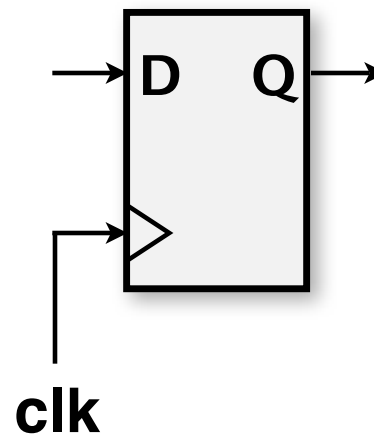
Clocking methodology ...

Processor uses synchronous logic design (a “clock”).

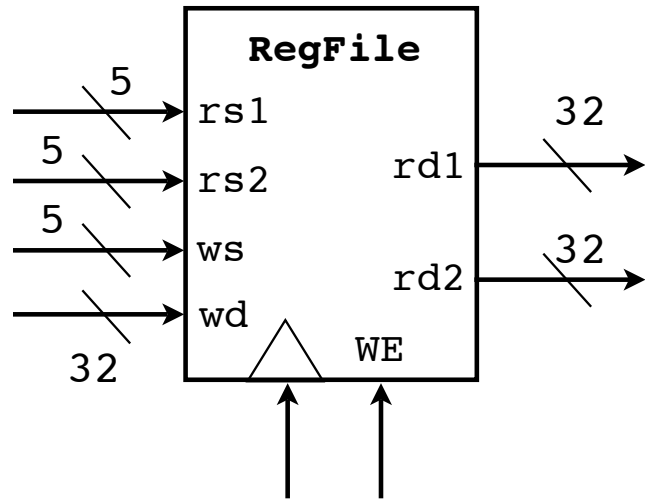


f	T
1 MHz	1 μ s
10 MHz	100 ns
100 MHz	10 ns
1 GHz	1 ns

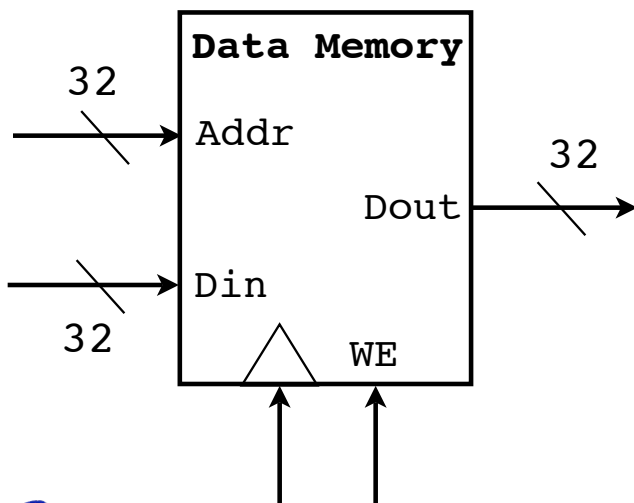
All state elements act like positive edge-triggered flip flops.



Memory and register file semantics ...



Reads are **combinational**:
Put a stable address on
input, a short time later
data appears on output.



Writes are **clocked**: If **WE**
is high, memory **Addr**
(or register file **ws**)
captures memory **Din**
(or register file **wd**) on
positive edge of clock.



Note: May not be best choice for project.

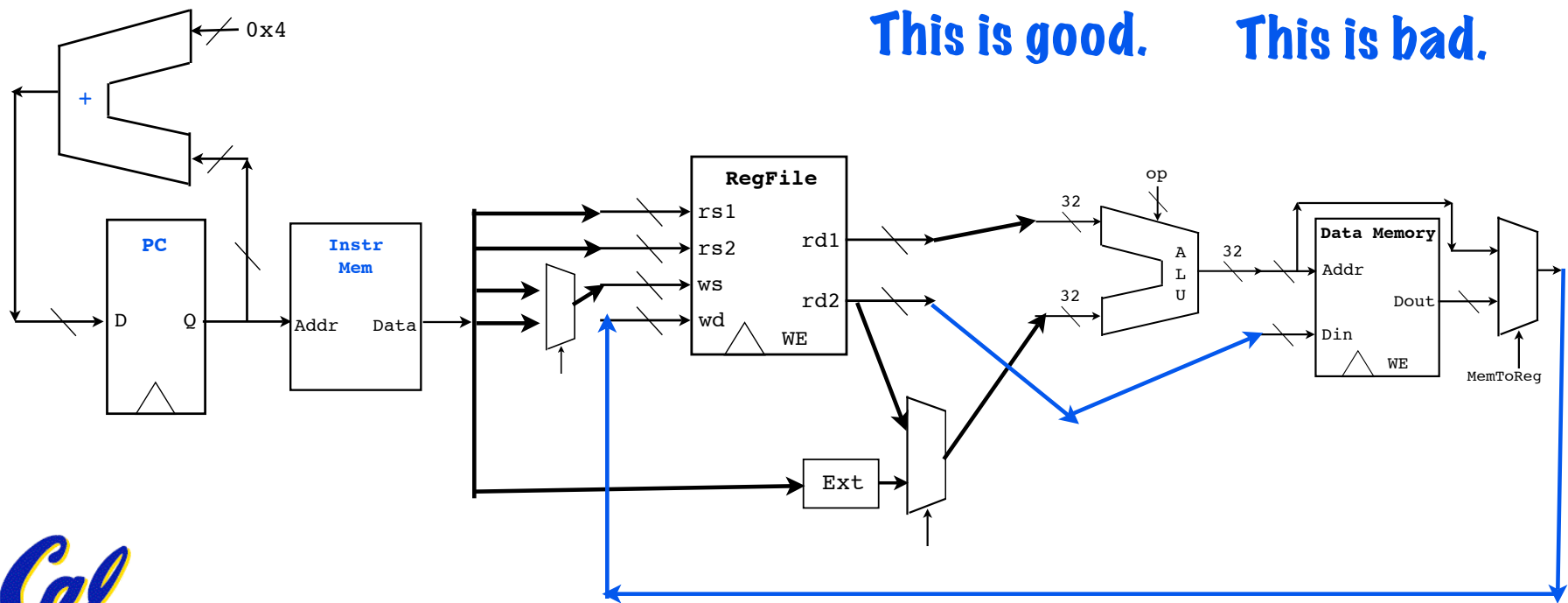
Recall: A single-cycle processor

Challenge: Speed up clock while keeping $CPI == 1$

$$\frac{\text{Seconds Program}}{\text{Program}} = \frac{\text{Instructions Program}}{\text{Program}} \frac{\text{Cycles Instruction}}{\text{Instruction}} \frac{\text{Seconds Cycle}}{\text{Cycle}}$$

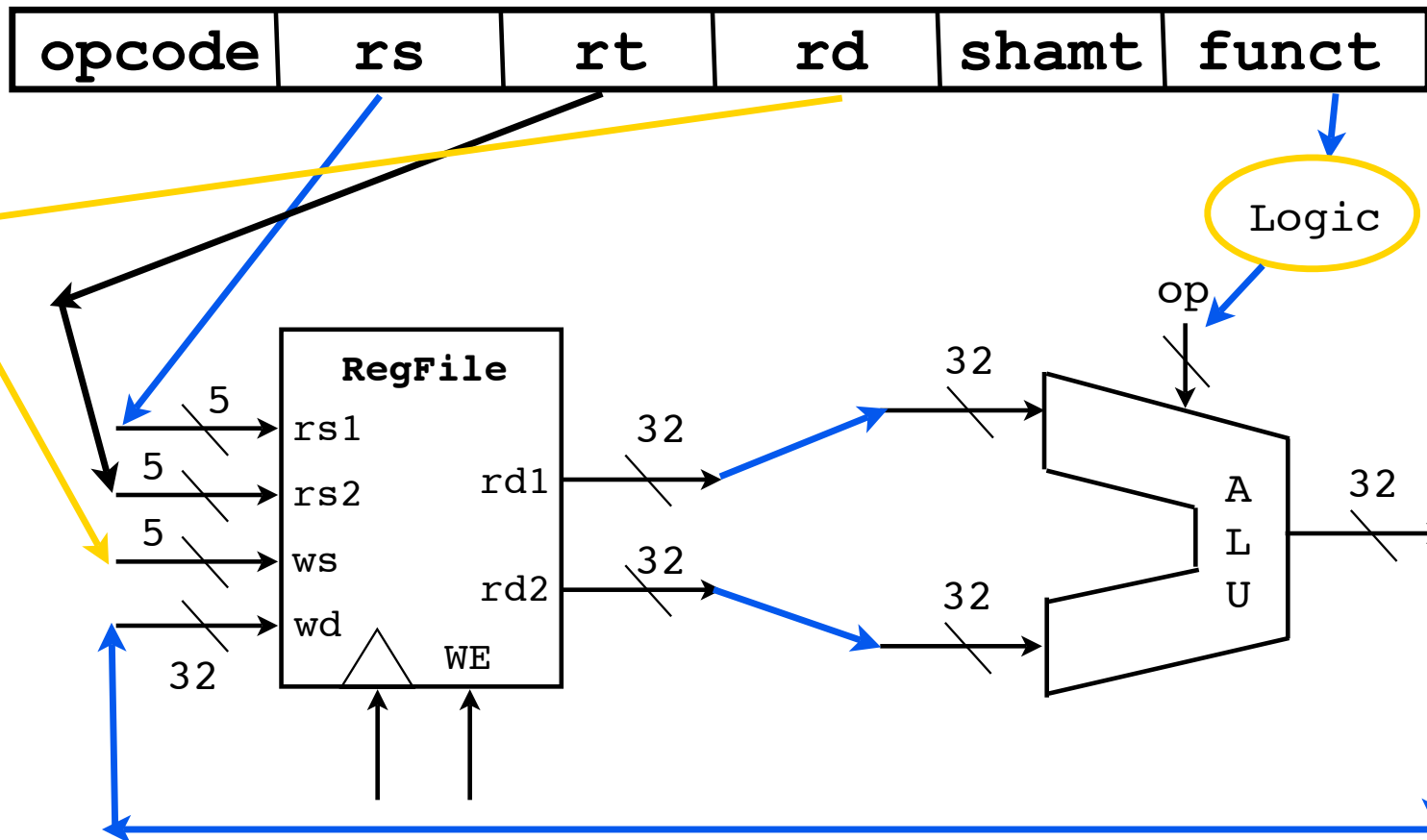
↑
 $CPI == 1$
This is good.

↑
Slow.
This is bad.

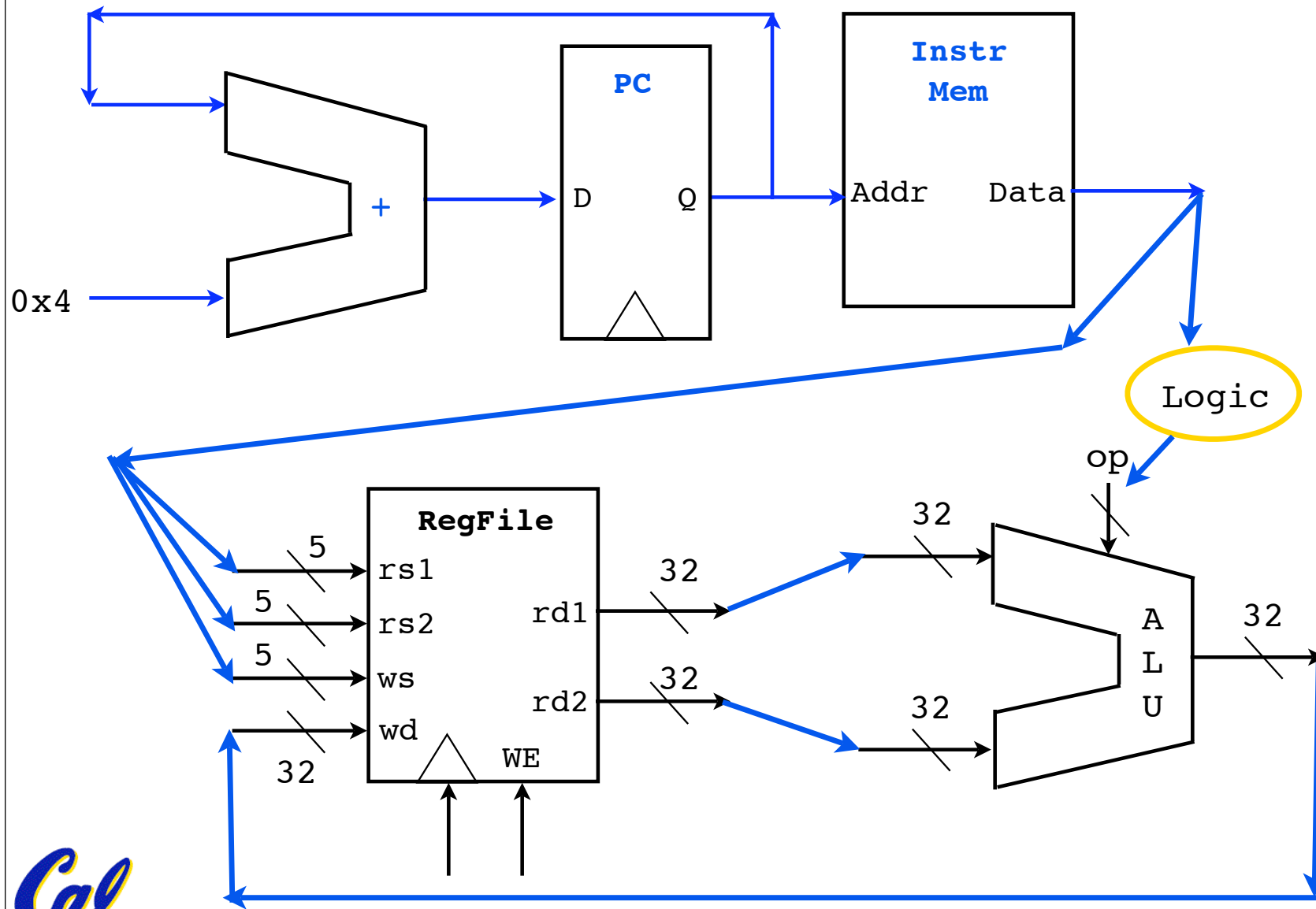


A MIPS R-format CPU design

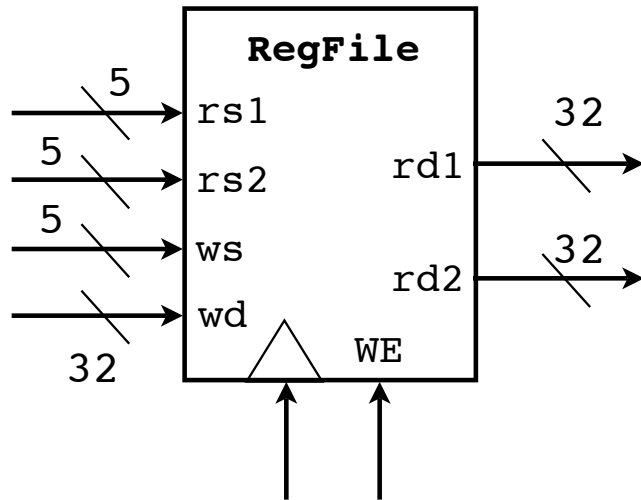
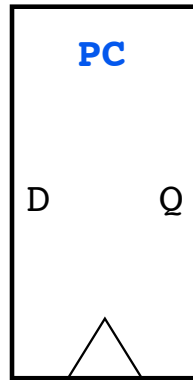
Decode fields to get : ADD \$8 \$9 \$10



How data flows after posedge



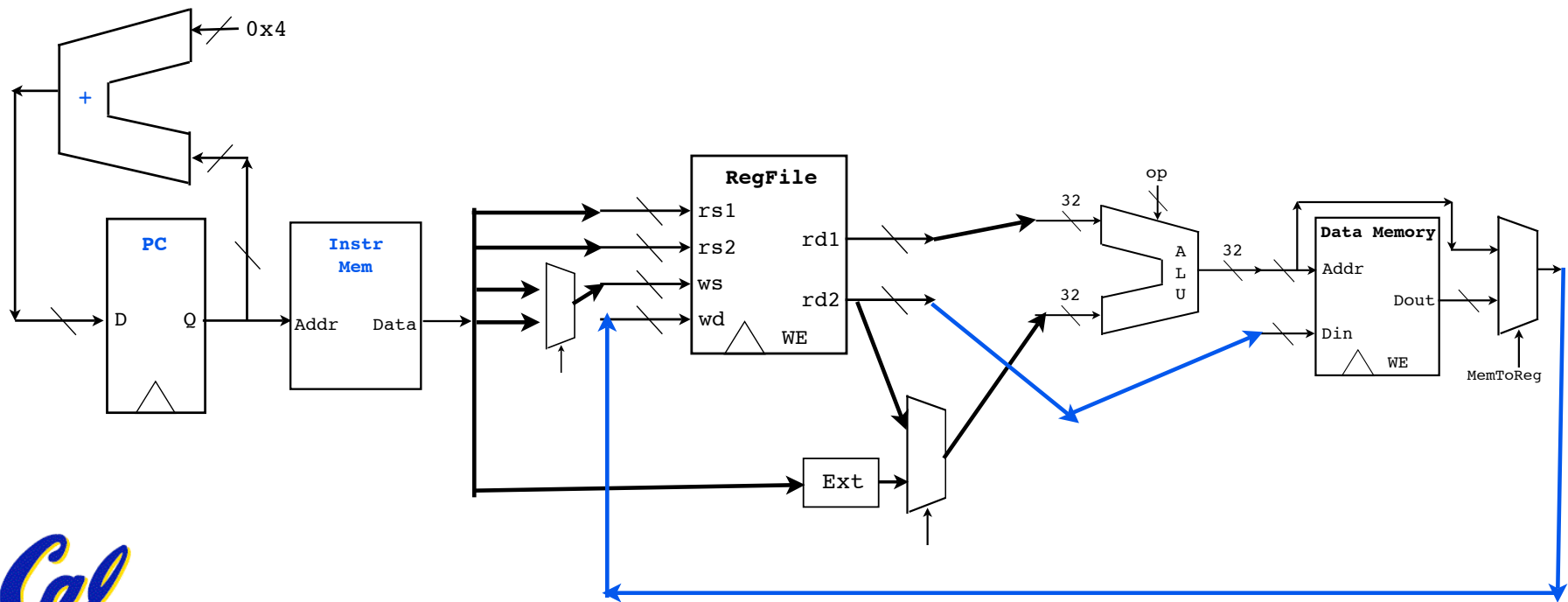
Next posedge: Update state and repeat



Observation: Logic idle most of cycle

For most of cycle, ALU is either “waiting” for its inputs, or “holding” its output

Ideal: a CPU architecture where each part is always “working”.



Inspiration: Automobile assembly line

Assembly line moves on a steady clock.
Each station does the same task on each car.

The clock

Car body shell

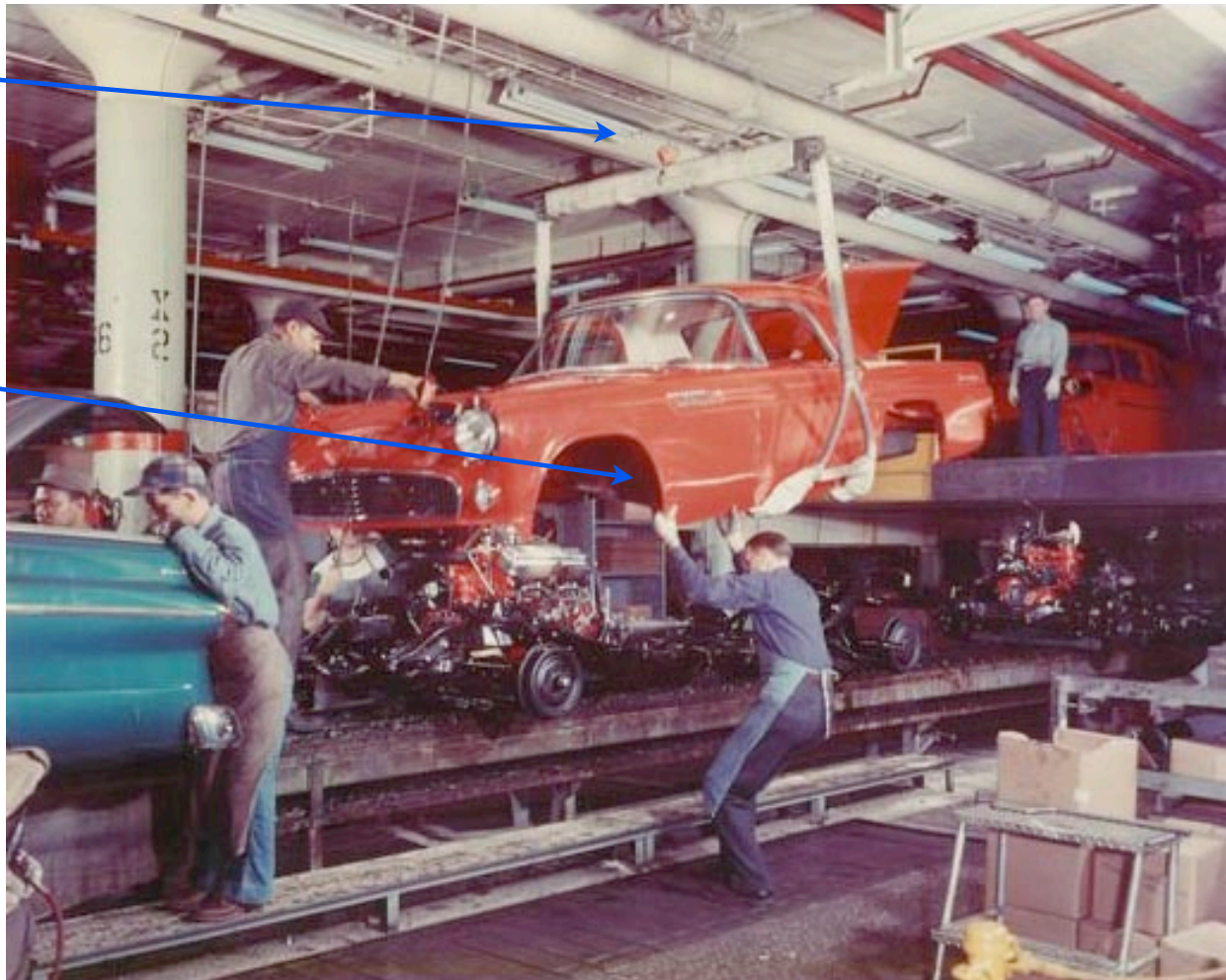
Merge station



Bolting station



Car chassis



Lessons from car assembly lines



Faster line movement yields more cars per hour off the line.



Faster line movement requires more stages, each doing simpler tasks.



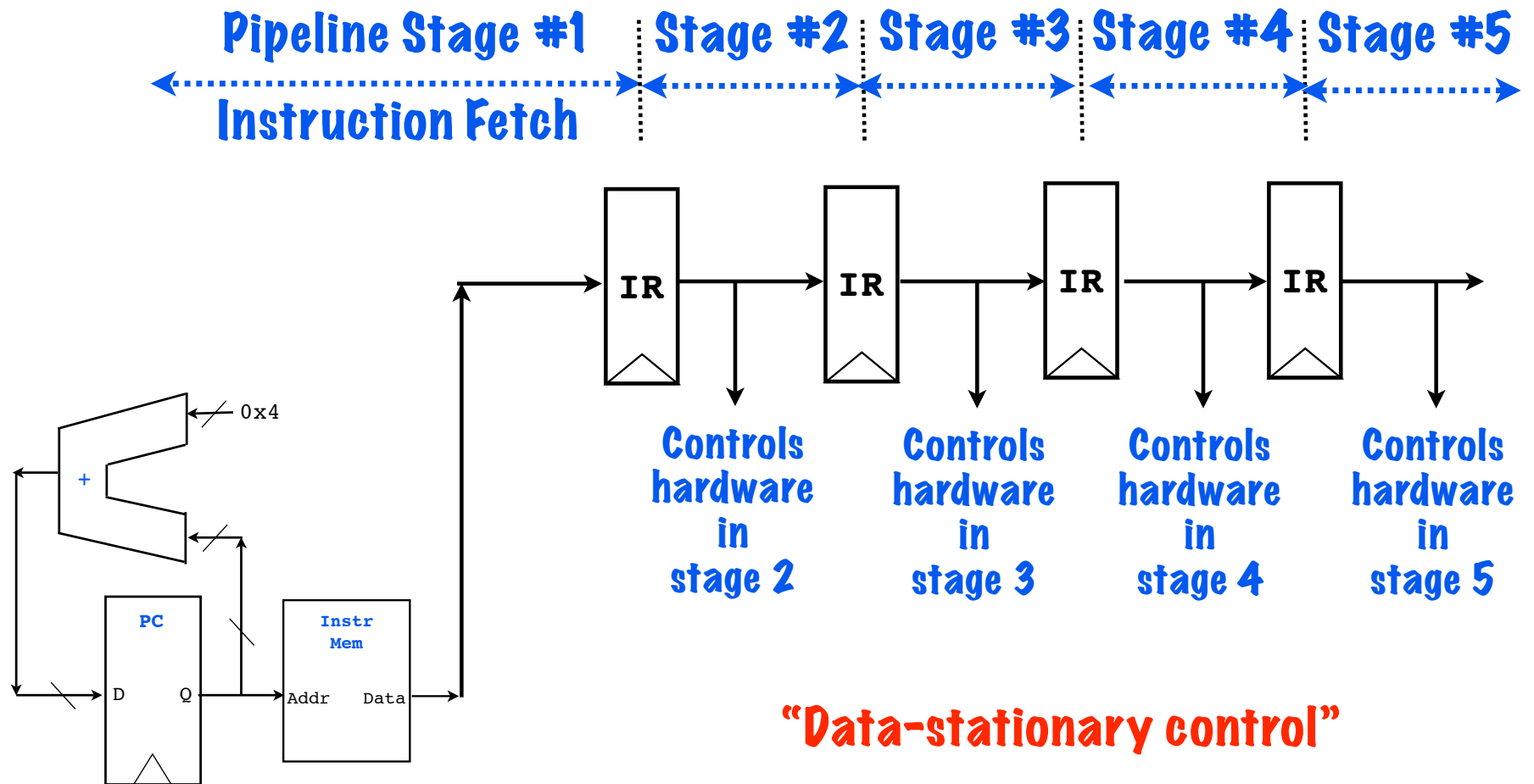
To maximize efficiency, all stages should take same amount of time (if not, workers in fast stages are idle)



“Filling”, “flushing”, and “stalling” assembly line are all bad news.



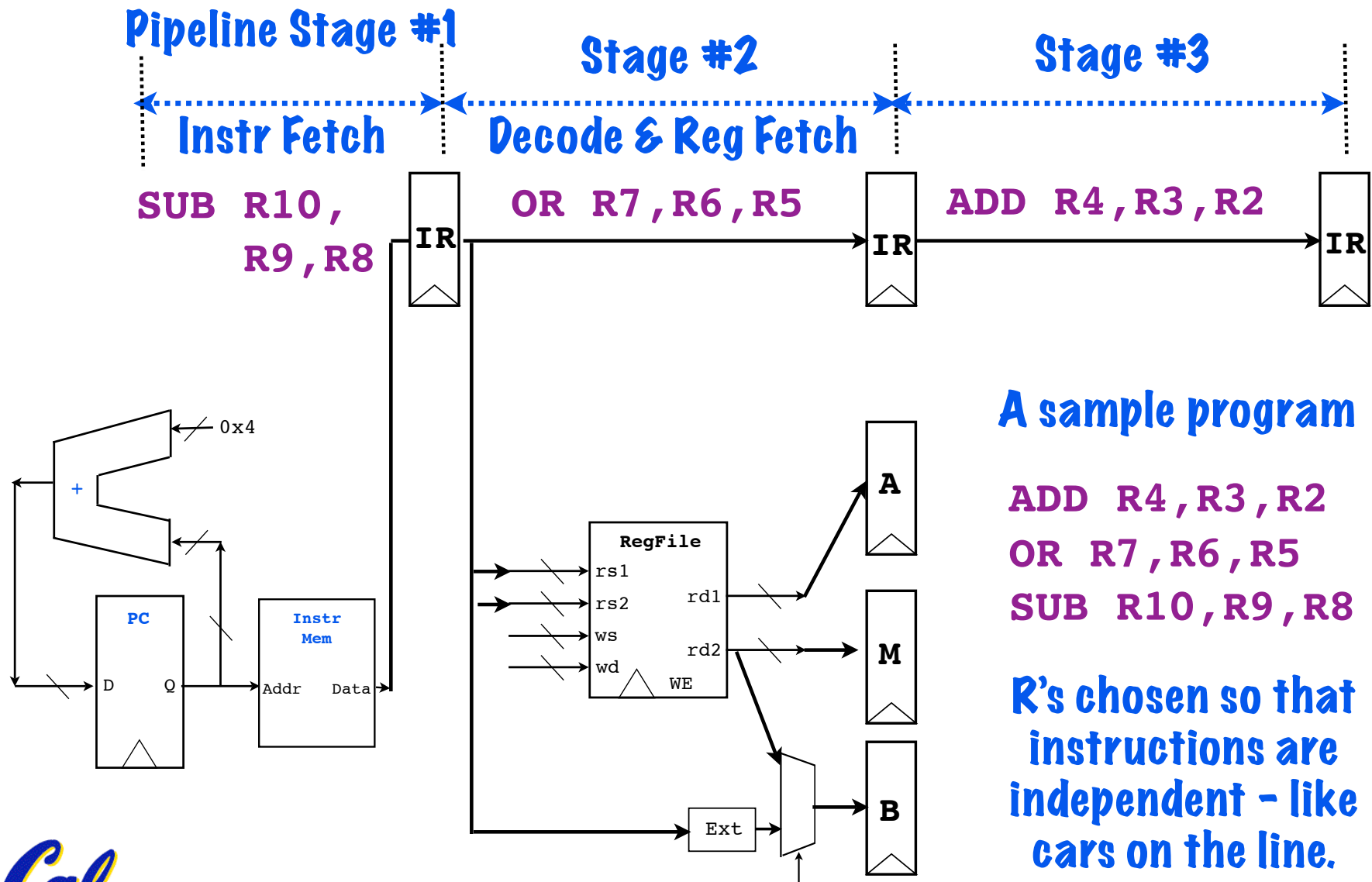
Key analogy: The instruction is the car



"Data-stationary control"

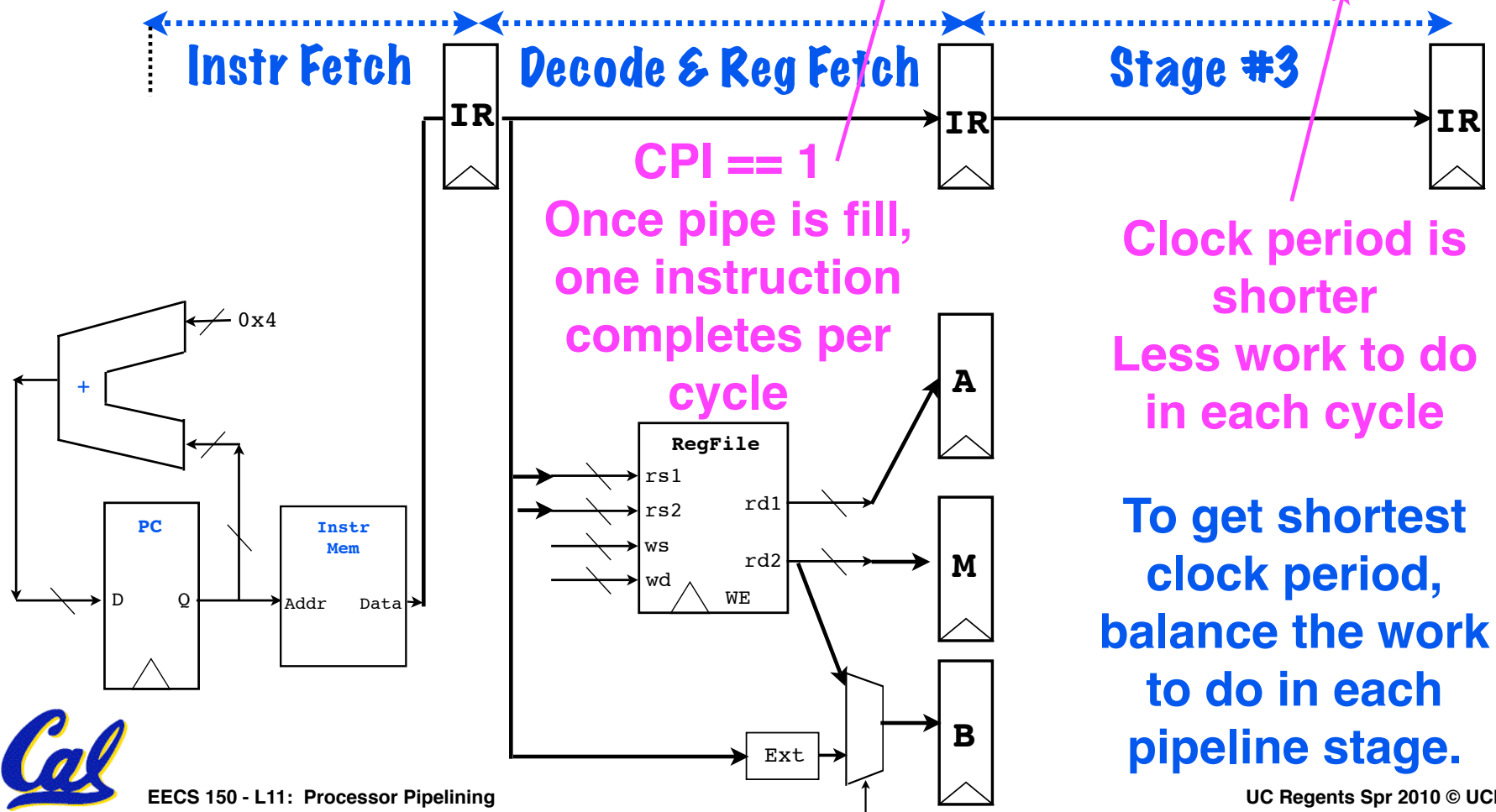


Example: Decode & Register Fetch stage

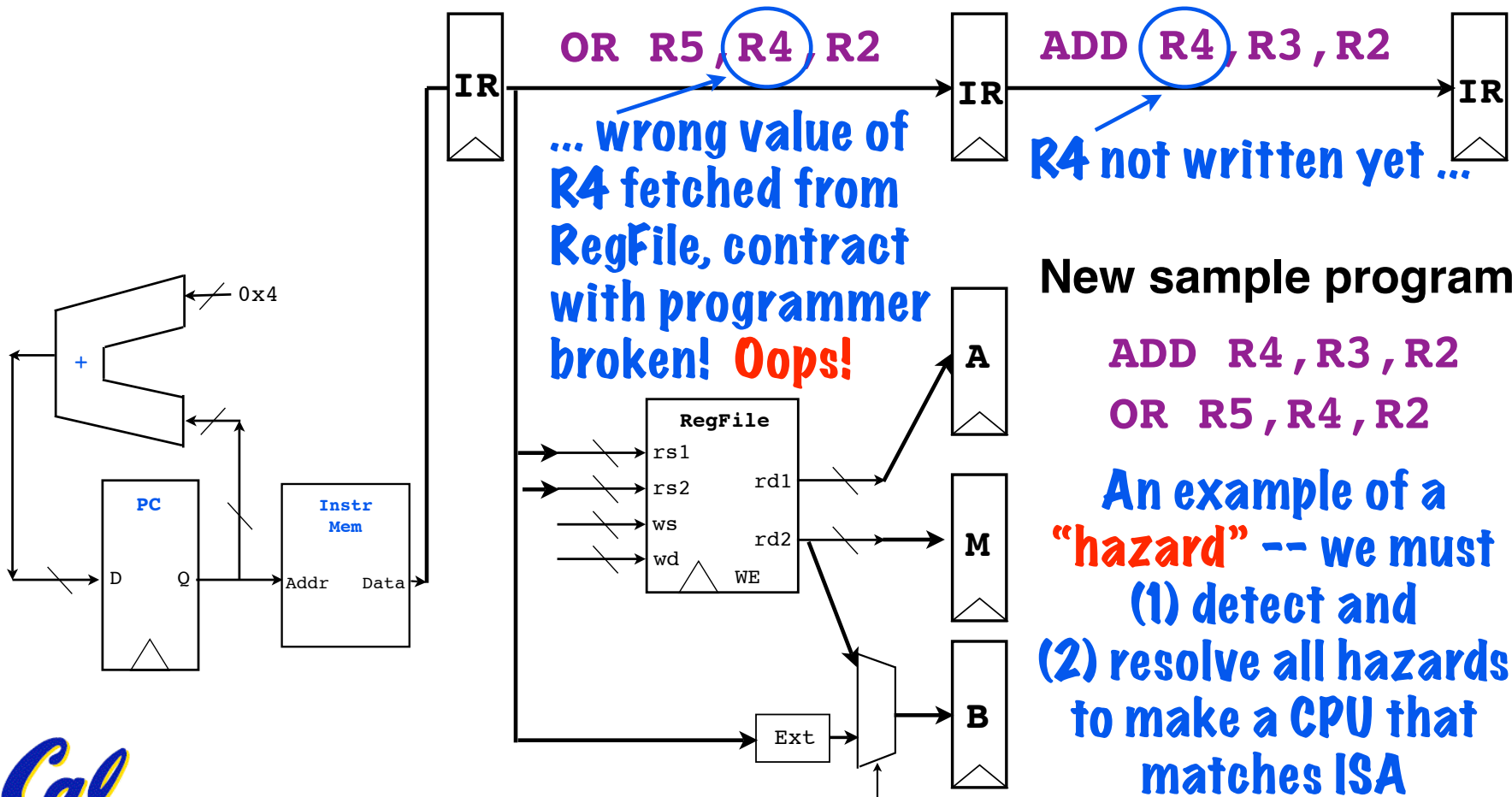


Performance Equation and Pipelining

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$



Hazards: An instruction is not a car ...



New sample program

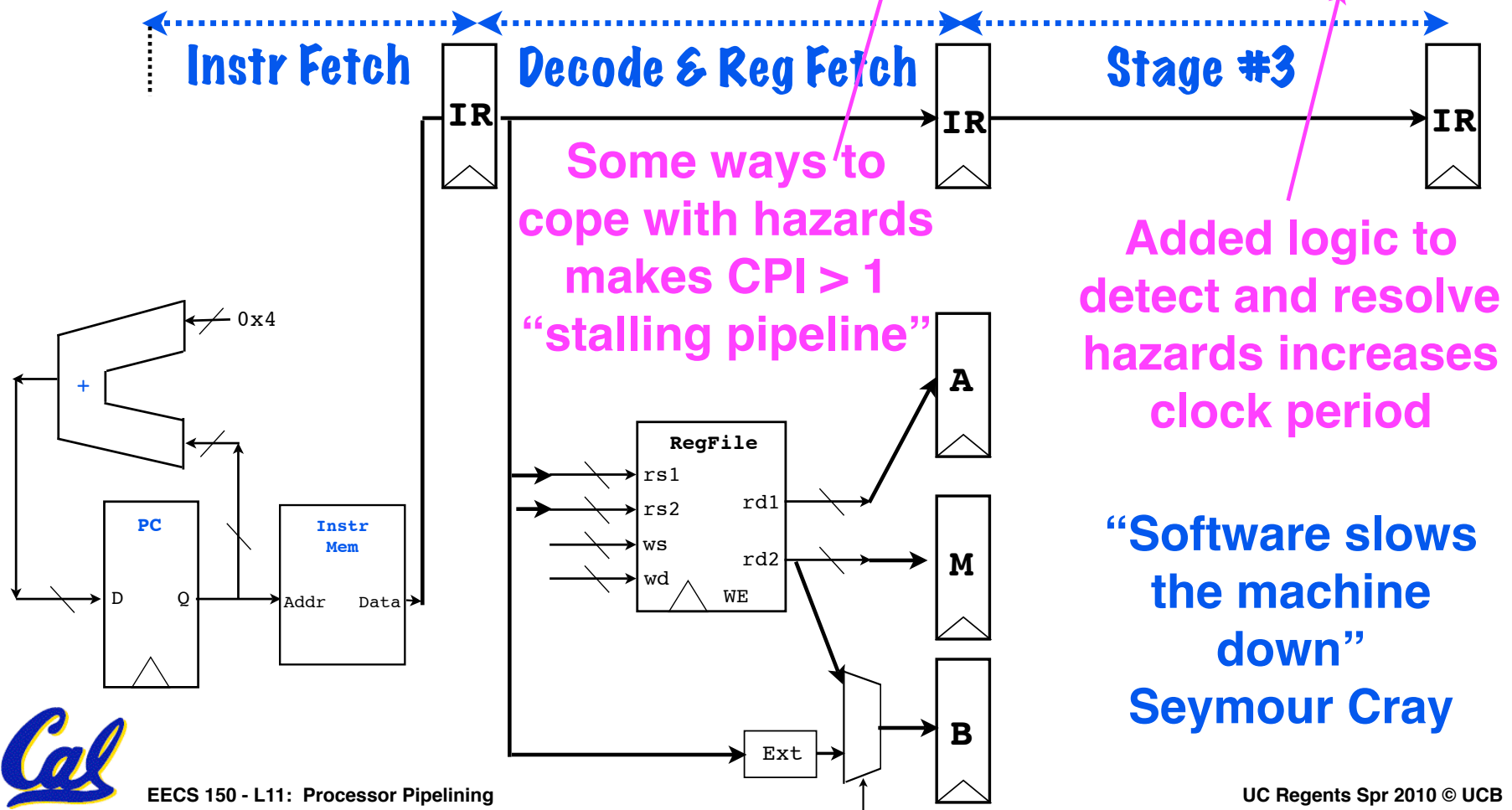
```
ADD R4, R3, R2
OR R5, R4, R2
```

An example of a **“hazard”** -- we must
 (1) detect and
 (2) resolve all hazards
 to make a CPU that
 matches ISA

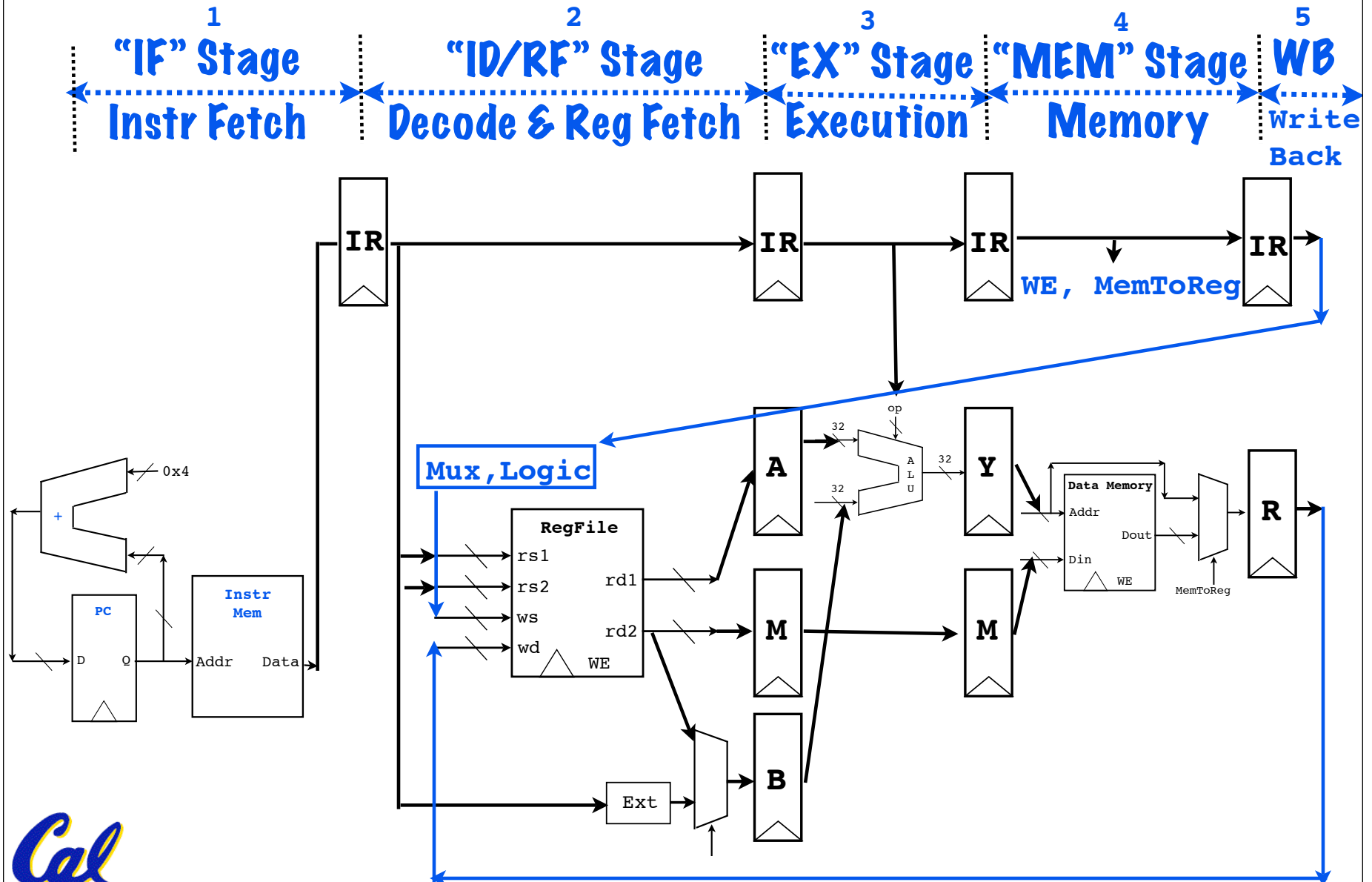


Performance Equation and Hazards

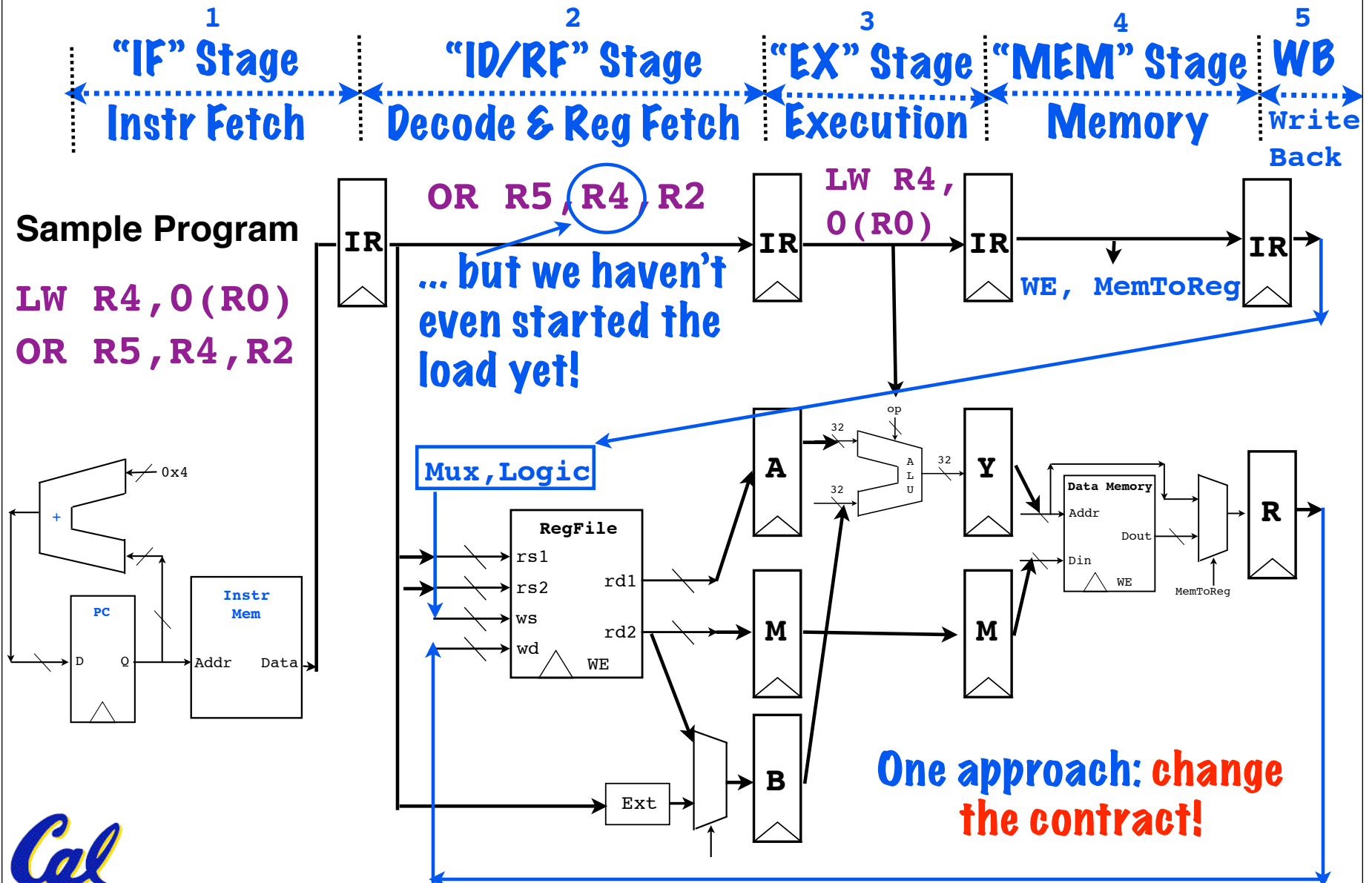
$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Seconds}}{\text{Cycle}}$$



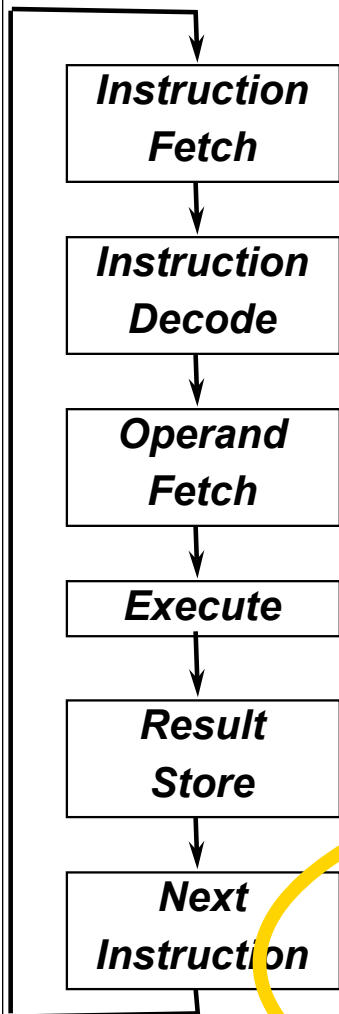
A (simplified) 5-stage pipelined CPU



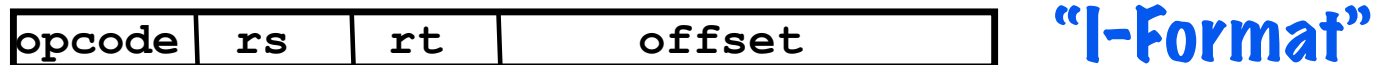
Sometimes, “contract” is a challenge



MIPS LW contract: Delayed Loads



Fetch the load inst from memory



Decode fields to get : LW \$1, 32(\$2)

"Retrieve" register value: \$2

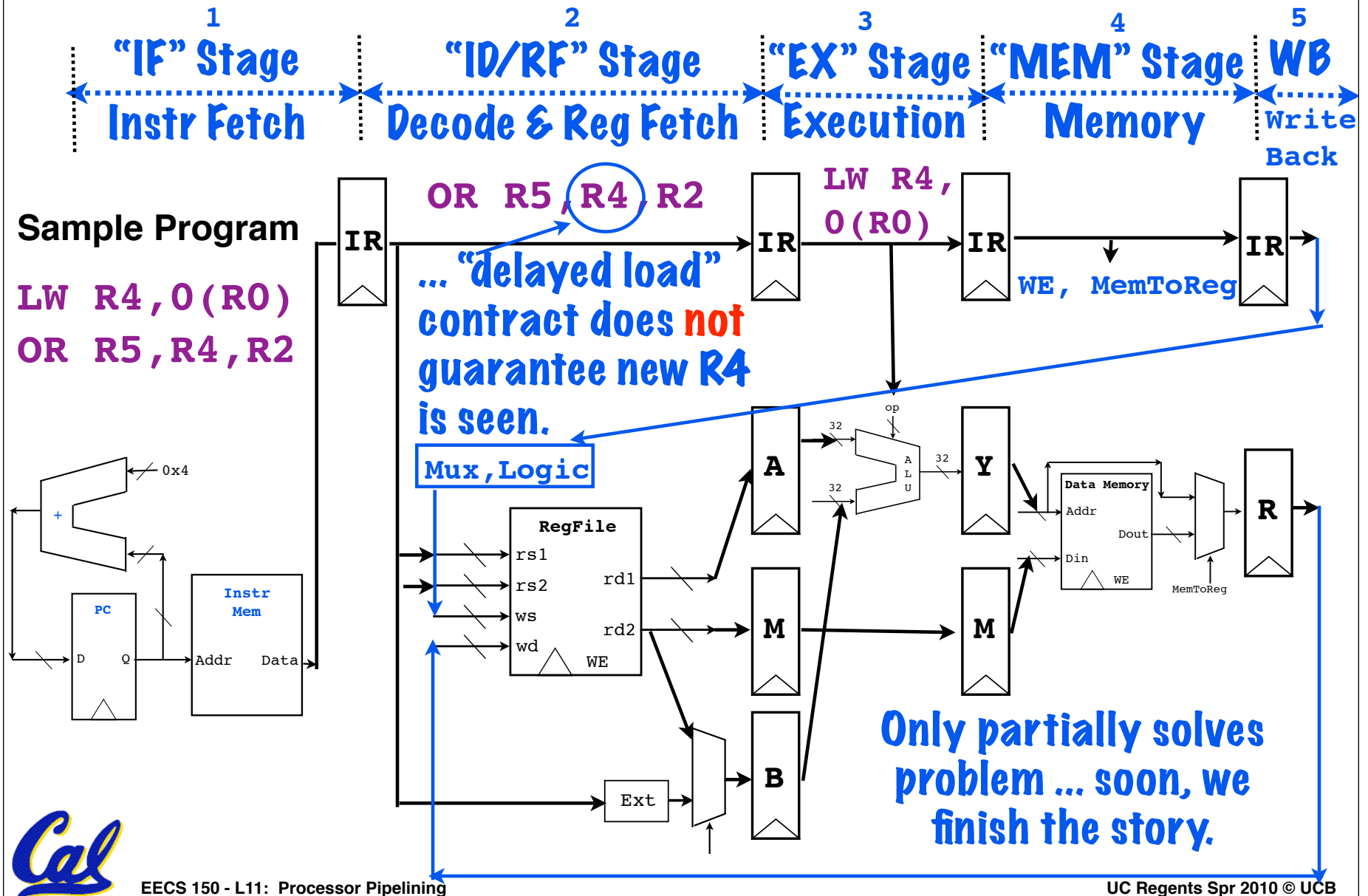
Compute memory address: $32 + \$2$

Load memory address contents into: \$1

Prepare to fetch instr that follows the LW in the program. Depending on load semantics, new \$1 is visible to that instr, or not until the following instr ("delayed loads").



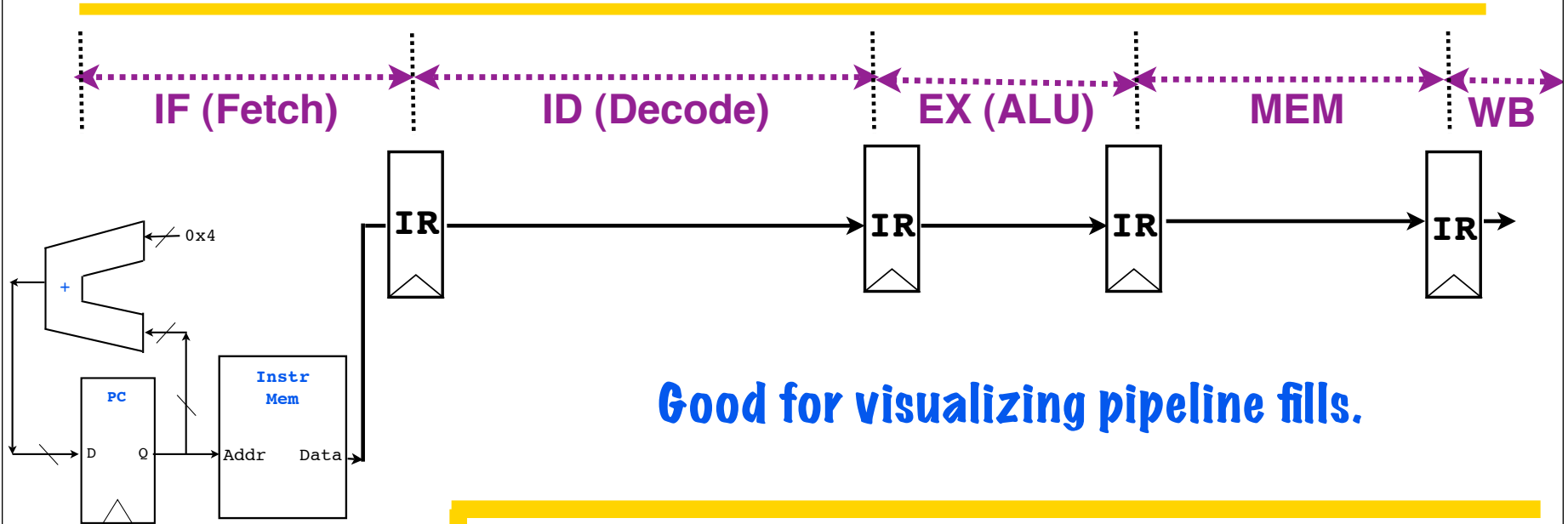
After we change the contract ...



Visualizing Pipelines



Pipeline Representation #1: Timeline



Good for visualizing pipeline fills.

Sample Program

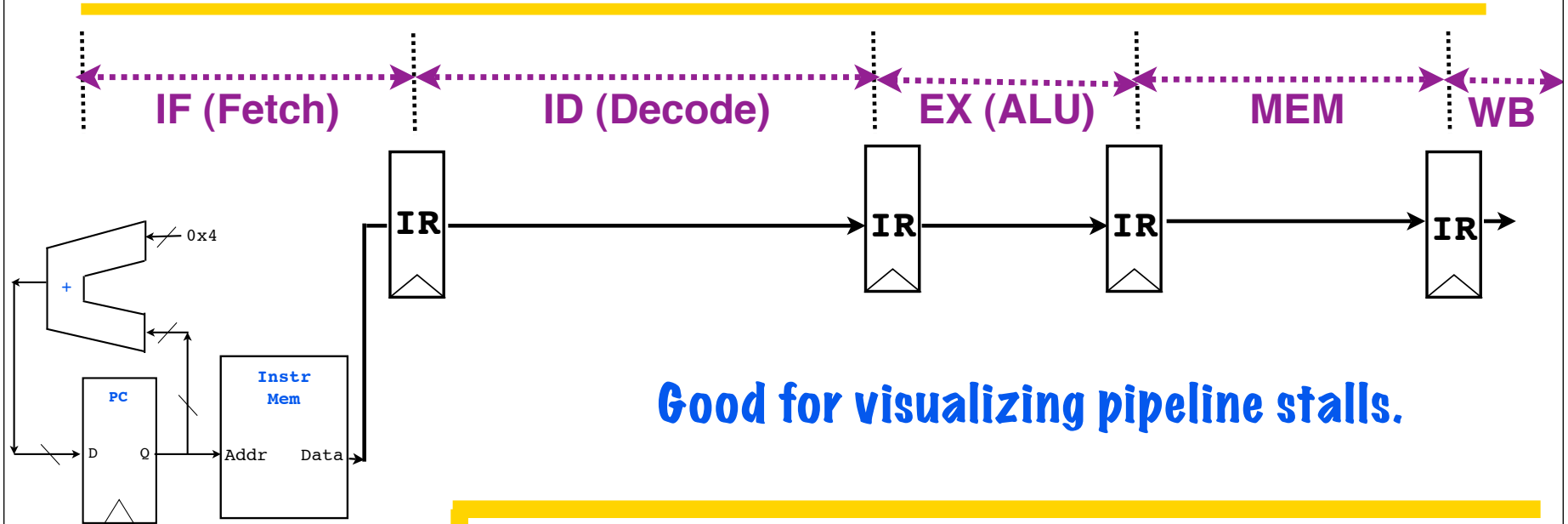
I1: ADD R4, R3, R2
 I2: AND R6, R5, R4
 I3: SUB R1, R9, R8
 I4: XOR R3, R2, R1
 I5: OR R7, R6, R5

Time:	t1	t2	t3	t4	t5	t6	t7	t8
Inst								
I1:	IF	ID	EX	MEM	WB			
I2:		IF	ID	EX	MEM	WB		
I3:			IF	ID	EX	MEM	WB	
I4:				IF	ID	EX	MEM	WB
I5:					IF	ID	EX	MEM
I6:						IF	ID	EX

Pipeline is "full"



Representation #2: Resource Usage



Good for visualizing pipeline stalls.

Sample Program

I1: ADD R4, R3, R2
 I2: AND R6, R5, R4
 I3: SUB R1, R9, R8
 I4: XOR R3, R2, R1
 I5: OR R7, R6, R5

Time:	t1	t2	t3	t4	t5	t6	t7	t8
Stage								
IF:	I1	I2	I3	I4	I5	I6	I7	I8
ID:		I1	I2	I3	I4	I5	I6	I7
EX:			I1	I2	I3	I4	I5	I6
MEM:				I1	I2	I3	I4	I5
WB:					I1	I2	I3	I4

Pipeline is "full"



Hazard Taxonomy



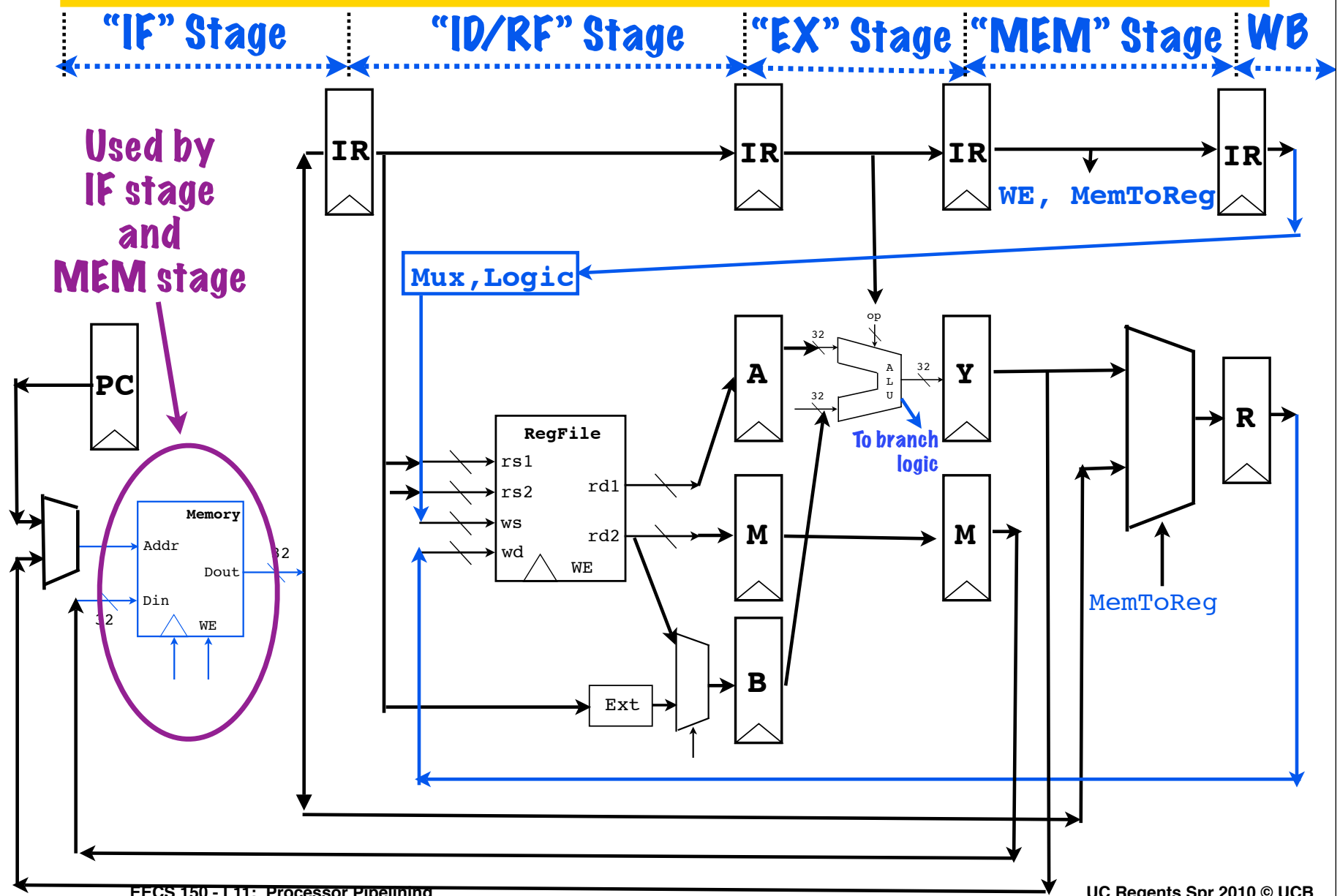
Structural Hazards

Several pipeline stages need to use the **same hardware resource** at the **same time**.

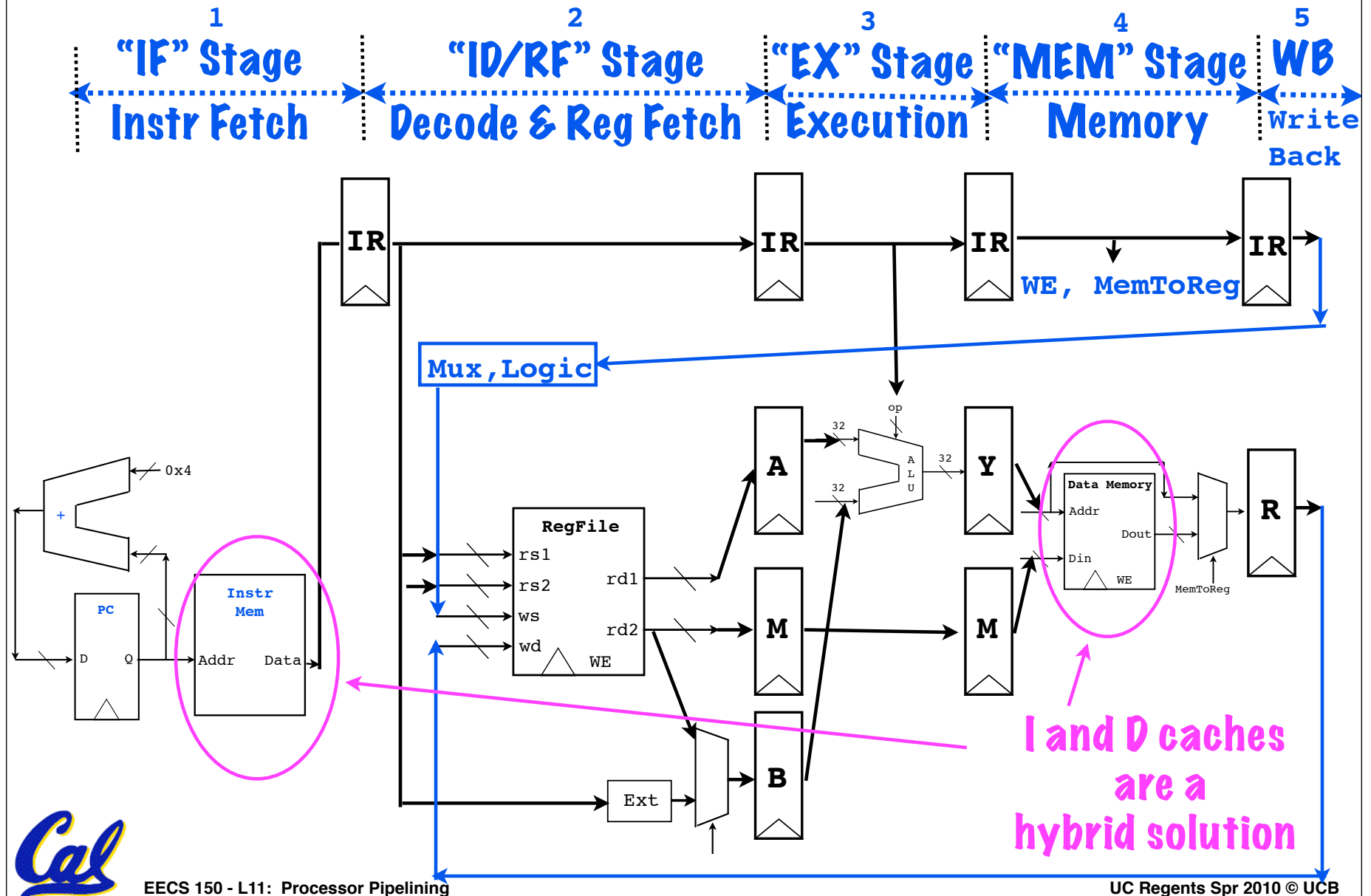
- * **Solution #1: Add **extra copies** of the resource (only works sometime).**
- * **Solution #2: Change resource so that it can handle **concurrent use**.**
- * **Solution #3: Stages “take turns” by **stalling** parts of the pipeline.**



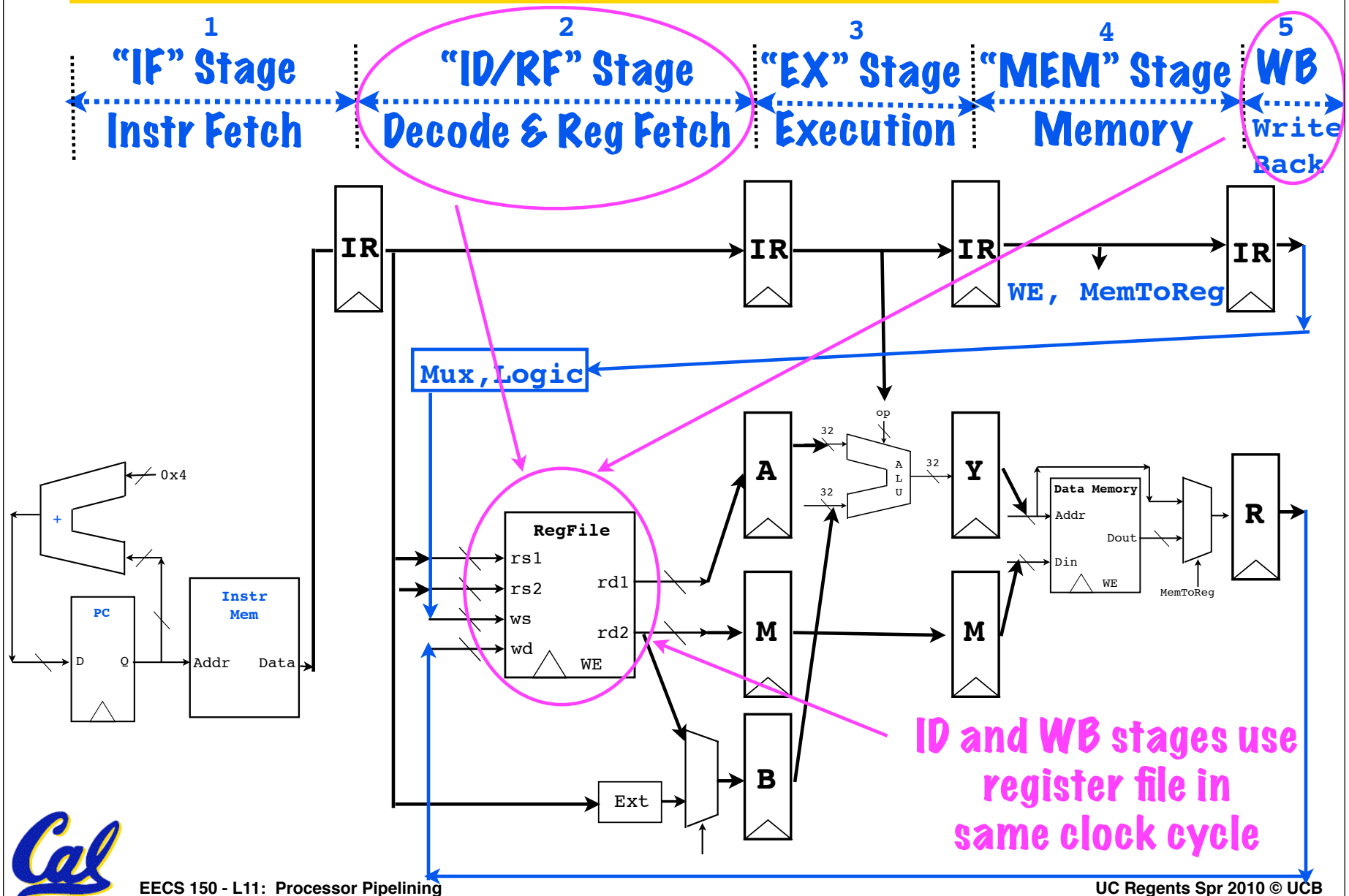
Structural Hazard Example: One Memory



A solution: "Extra copies" of memory



Alternatively: Concurrent use ...



Data Hazards: 3 Types (RAW, WAR, WAW)

Several pipeline stages read or write the same data location in an incompatible way.

Read After Write (RAW) hazards.

Instruction I2 expects to read a data value written by an earlier instruction, but I2 executes “too early” and reads the wrong copy of the data.

Note “data value”, not “register”. Data hazards are possible for any architected state (such as main memory). In practice, main memory hazard avoidance is the job of the memory system.

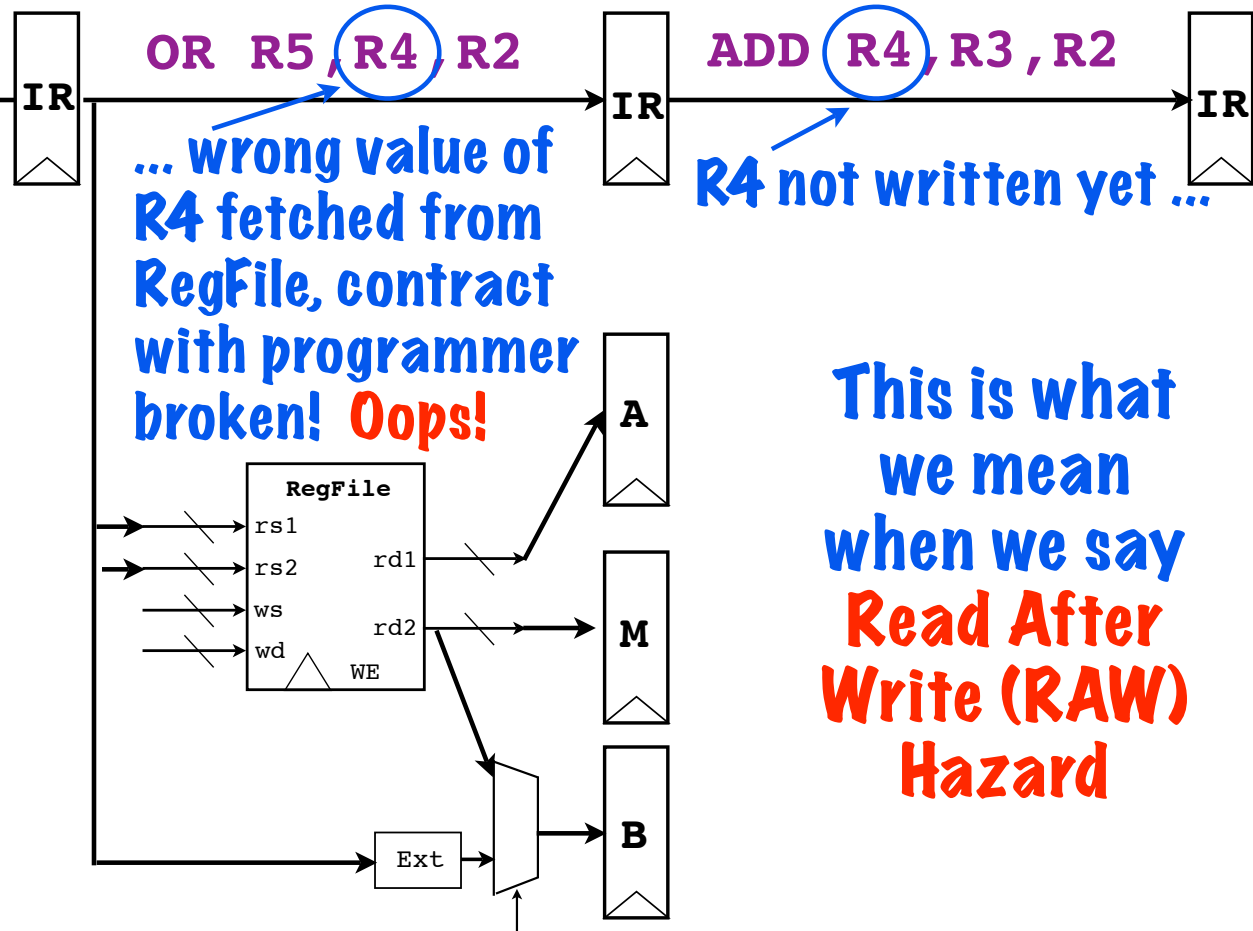
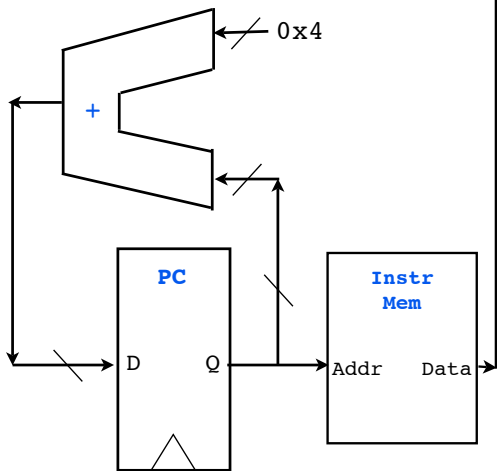


Recall: RAW example



Sample program

```
ADD R4, R3, R2
OR R5, R4, R2
```



This is what we mean when we say **Read After Write (RAW) Hazard**



Data Hazards: 3 Types (RAW, WAR, WAW)

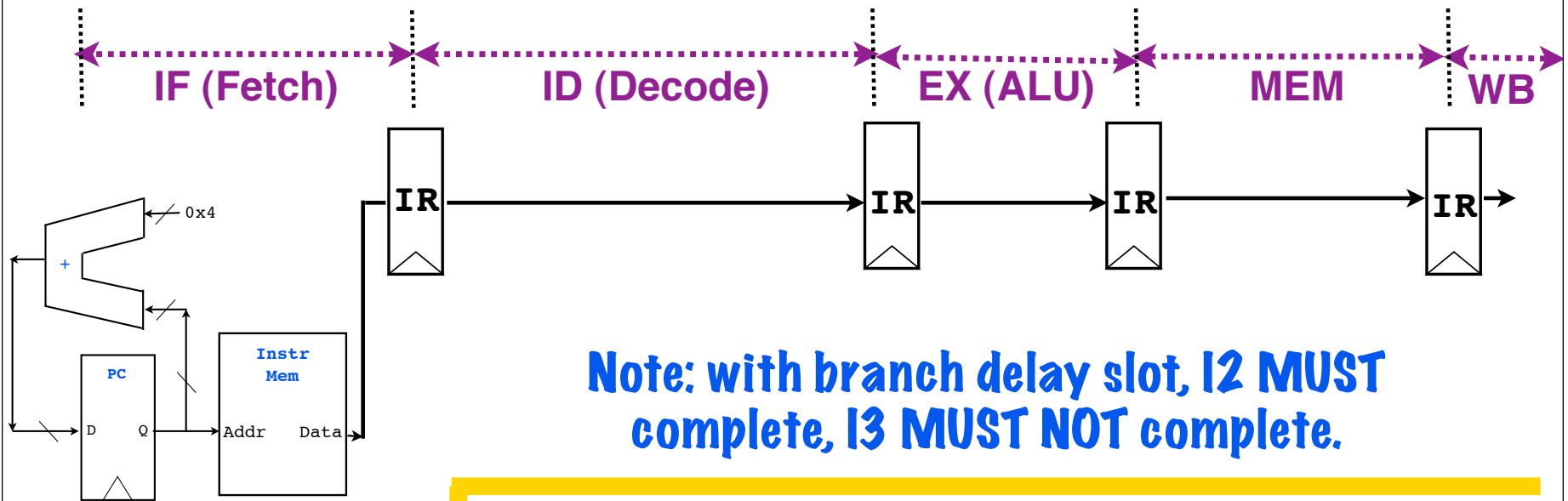
Write After Read (WAR) hazards. Instruction I2 expects to write over a data value after an earlier instruction I1 reads it. But instead, I2 **writes too early**, and I1 sees the new value.

Write After Write (WAW) hazards. Instruction I2 writes over data an earlier instruction I1 also writes. But instead, **I1 writes after I2**, and the final data value is incorrect.

**WAR and WAW not possible in our 5-stage pipeline.
But are possible in other pipeline designs.**



Control Hazards: A taken branch/jump



Note: with branch delay slot, I2 MUST complete, I3 MUST NOT complete.

**Sample Program
(ISA w/o branch
delay slot)**

**I1: BEQ R4, R3, 25
I2: AND R6, R5, R4
I3: SUB R1, R9, R8
I4:
I5:
I6:**

Time:	t1	t2	t3	t4	t5	t6	t7	t8
Inst								
I1:	IF	ID	EX	MEM	WB			
I2:		IF	ID					
I3:			IF					
I4:								
I5:								
I6:								

EX stage computes if branch is taken

If branch is taken, these instructions MUST NOT complete!



Hazards Recap

- * **Structural Hazards**
- * **Data Hazards (RAW, WAR, WAW)**
- * **Control Hazards (taken branches and jumps)**

On each clock cycle, we must detect the presence of all of these hazards, and resolve them before they break the “contract with the programmer”.



Hazard Resolution Tools



The Hazard Resolution Toolkit

- * **Stall** earlier instructions in pipeline.
- * **Forward** results computed in later pipeline stages to earlier stages.
- * **Add** new hardware or **rearrange** hardware design to eliminate hazard.
- * **Change ISA** to eliminate hazard.
- * **Kill** earlier instructions in pipeline.
- * **Make hardware handle concurrent requests** to eliminate hazard.

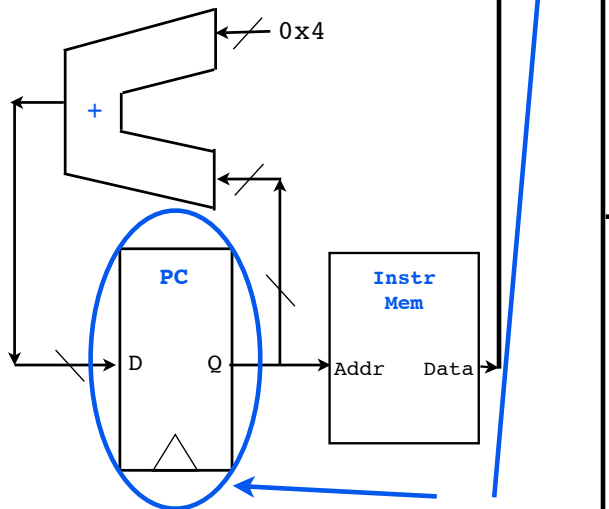


Resolving a RAW hazard by stalling



Sample program

ADD R4, R3, R2
OR R5, R4, R2



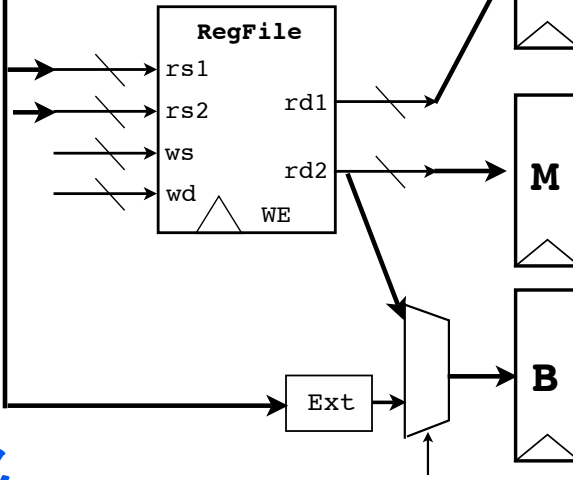
Freeze PC and IR until stall is over.

OR R5, R4, R2

Keep executing OR instruction until R4 is ready. Until then, send NOPS to IR 2/3.

ADD R4, R3, R2

Let ADD proceed to WB stage, so that R4 is written to regfile.



New datapath hardware

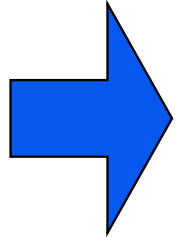
(1) Mux into IR 2/3 to feed in NOP.

(2) Write enable on PC and IR 1/2



The Hazard Resolution Toolkit

* **Stall** earlier instructions in pipeline.



Forward results computed in later pipeline stages to earlier stages.

* **Add** new hardware or **rearrange** hardware design to eliminate hazard.

* **Change ISA** to eliminate hazard.

* **Kill** earlier instructions in pipeline.

* **Make hardware handle concurrent requests** to eliminate hazard.



Resolving a RAW hazard by forwarding

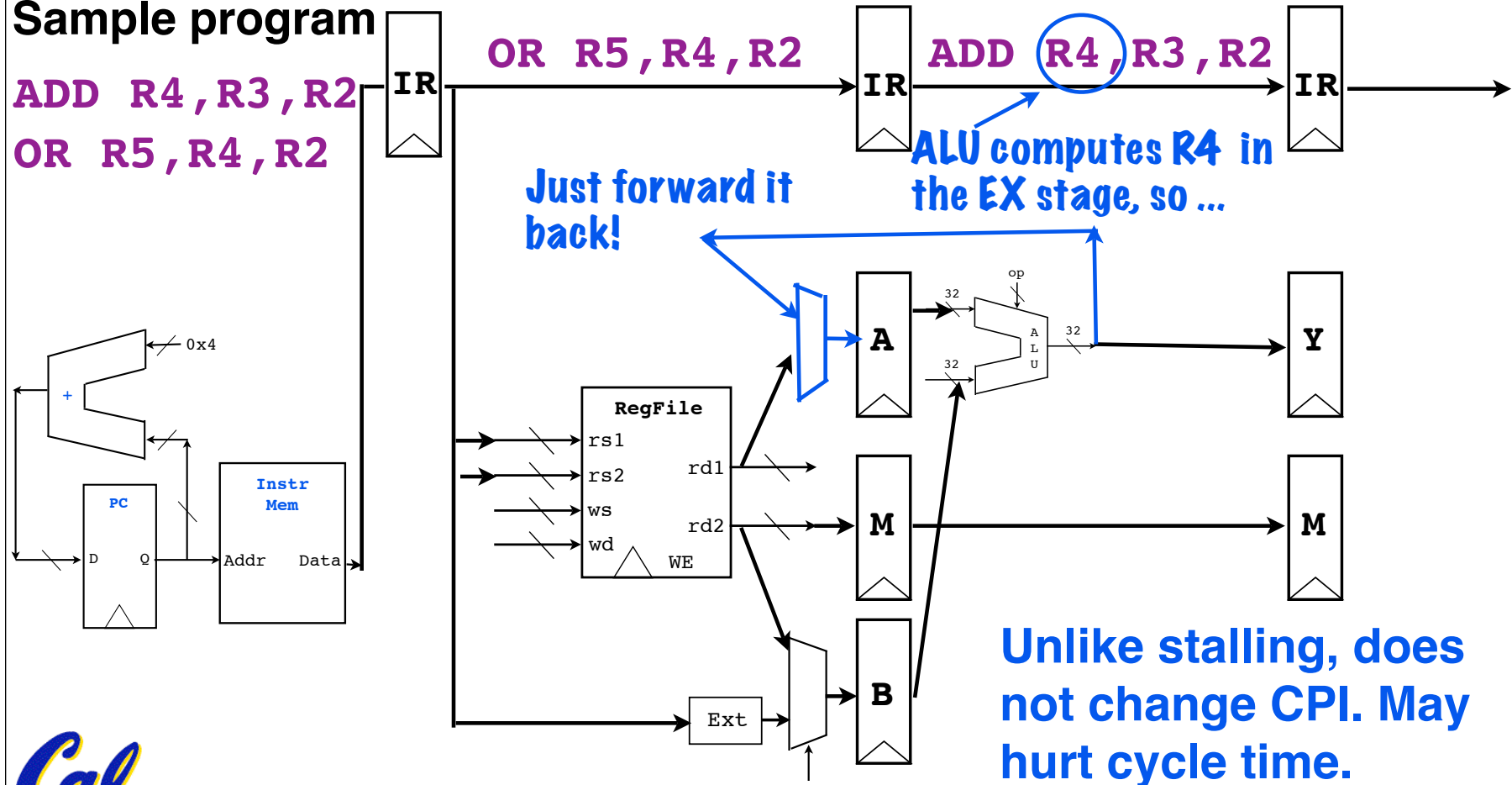


Sample program

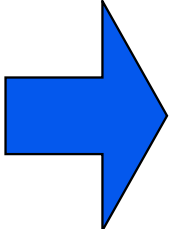
```
ADD R4, R3, R2
OR R5, R4, R2
```

```
OR R5, R4, R2
```

```
ADD R4, R3, R2
```

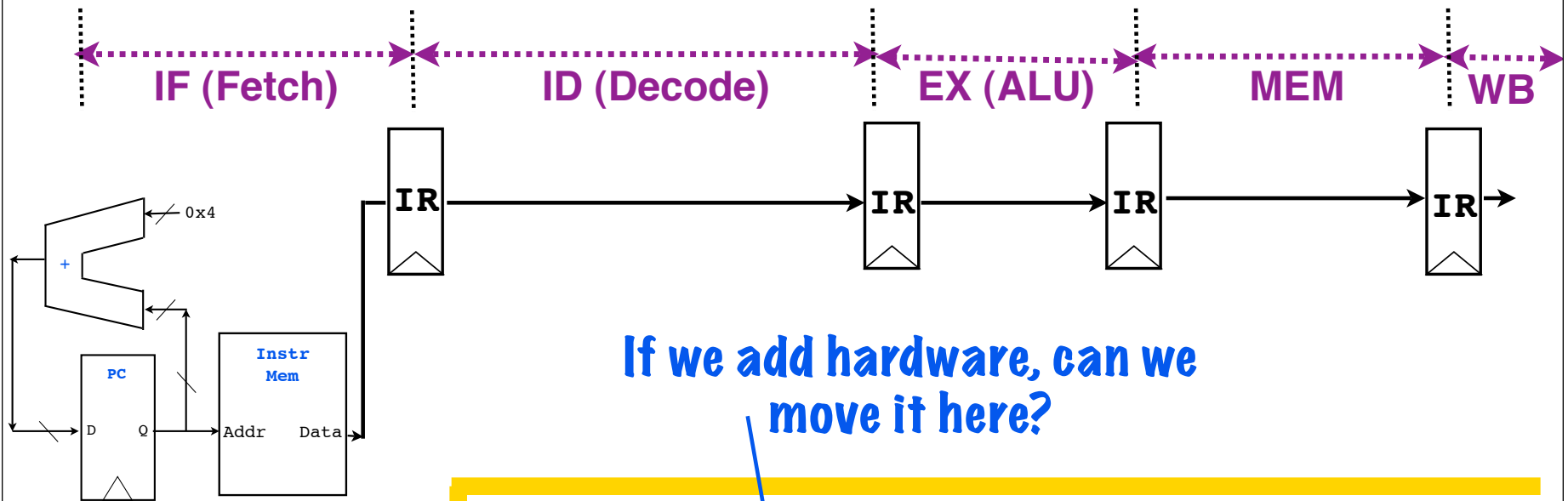


The Hazard Resolution Toolkit

- * **Stall** earlier instructions in pipeline.
- * **Forward** results computed in later pipeline stages to earlier stages.
-  **Add** new hardware or **rearrange** hardware design to eliminate hazard.
- * **Change ISA** to eliminate hazard.
- * **Kill** earlier instructions in pipeline.
- * **Make** hardware handle **concurrent requests** to eliminate hazard.



Control Hazards: Fix with more hardware



If we add hardware, can we move it here?

Sample Program
(ISA w/o branch delay slot)

I1: BEQ R4, R3, 25
I2: AND R6, R5, R4
I3: SUB R1, R9, R8
I4:
I5:
I6:

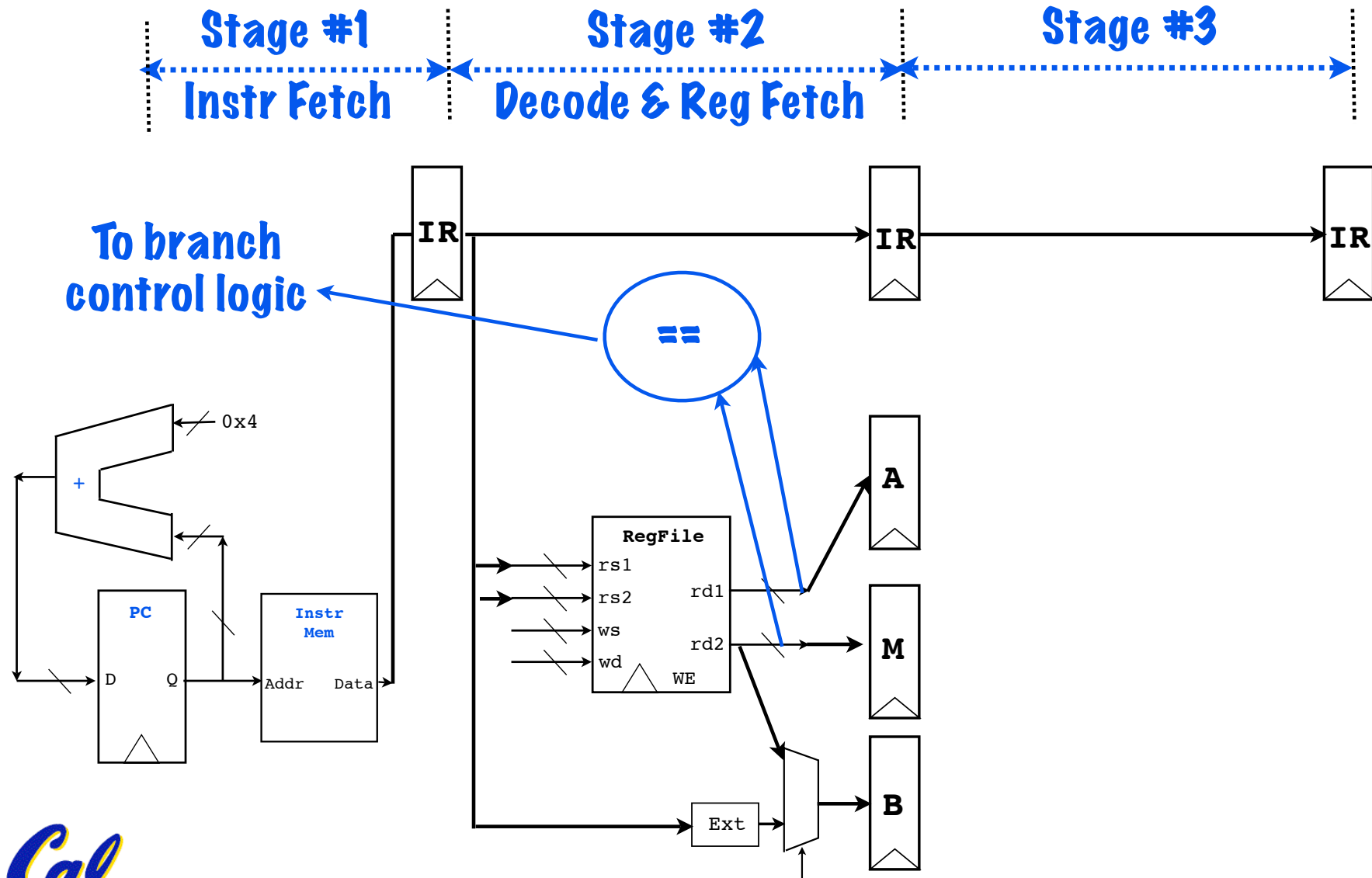
Time:	t1	t2	t3	t4	t5	t6	t7	t8
Inst								
I1:	IF	ID	EX	MEM	WB			
I2:		IF	ID					
I3:			IF					
I4:								
I5:								
I6:								

EX stage computes if branch is taken

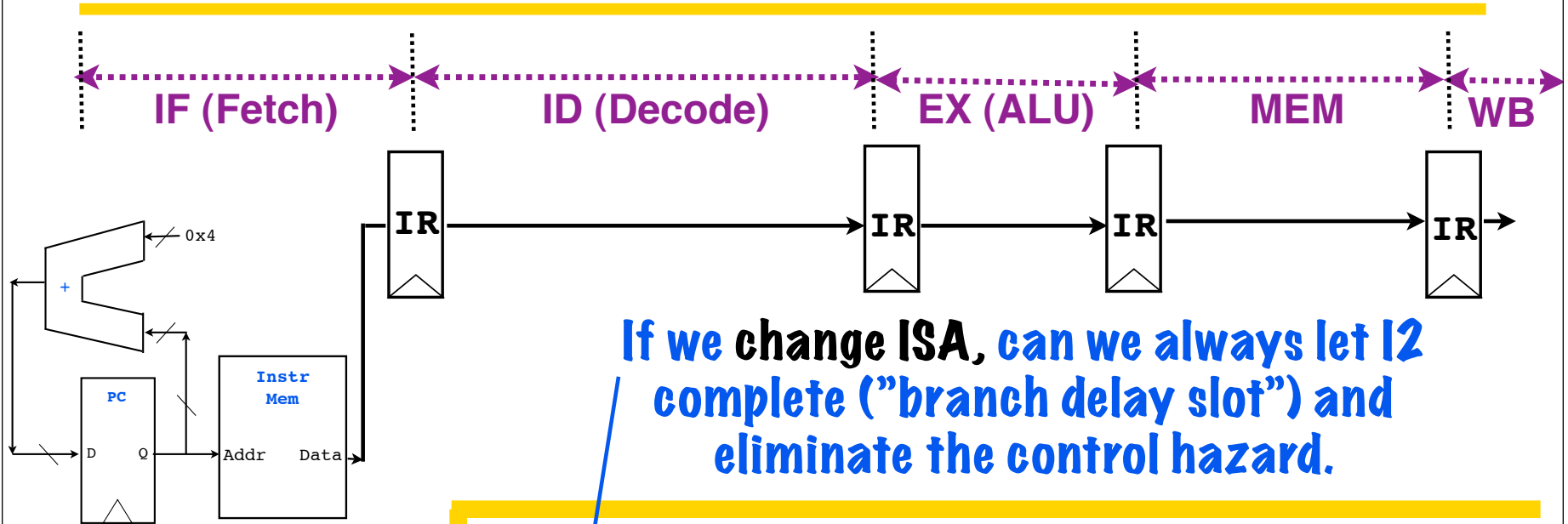
If branch is taken, these instructions MUST NOT complete!



Resolving control hazard with hardware



Control Hazards: After more hardware



If we change ISA, can we always let I2 complete ("branch delay slot") and eliminate the control hazard.

Sample Program
(ISA w/o branch delay slot)

I1: BEQ R4, R3, 25
I2: AND R6, R5, R4
I3: SUB R1, R9, R8

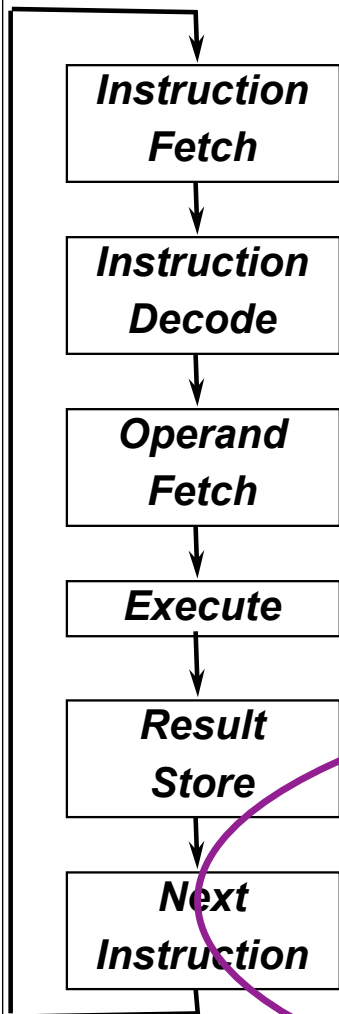
Time:	t1	t2	t3	t4	t5	t6	t7	t8
Inst								
I1:	IF	ID	EX	MEM	WB			
I2:		IF						
I3:								
I4:								
I5:								
I6:								

ID stage computes if branch is taken

If branch is taken, this instruction **MUST NOT** complete!



MIPS Delayed Branch: BEQ \$1, \$2, 25



Fetch branch inst from memory



Decode fields to get: BEQ \$1, \$2, 25

"Retrieve" register values: \$1, \$2

Compute if we take branch: $\$1 == \2 ?

ALWAYS prepare to fetch instr that follows the BEQ in the program ("delayed branch"). IF we take branch, the instr we fetch AFTER that instruction is $PC + 4 + 100$.

PC == "Program Counter"



The Hazard Resolution Toolkit

- * **Stall** earlier instructions in pipeline.
- * **Forward** results computed in later pipeline stages to earlier stages.
- * **Add** new hardware or **rearrange** hardware design to eliminate hazard.
- * **Change ISA** to eliminate hazard.
- ➔ **Kill** earlier instructions in pipeline.
- * **Make hardware handle concurrent requests** to eliminate hazard.

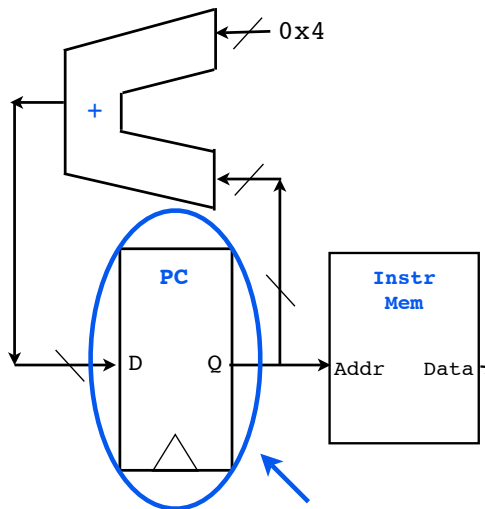


Resolve control hazard by killing instr

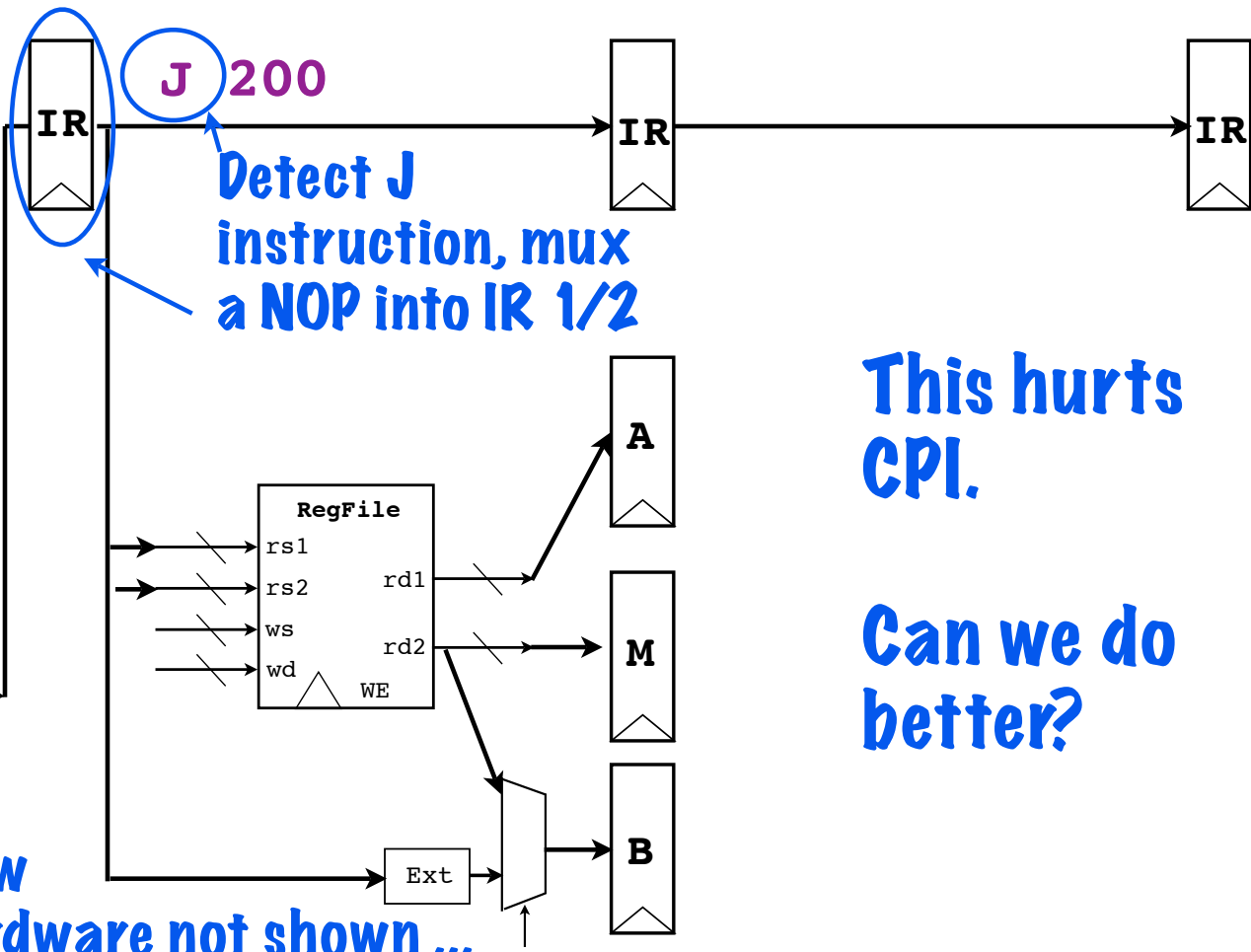


Sample program
(no delay slot)

J 200
OR R5, R4, R2



Compute new PC using hardware not shown ...

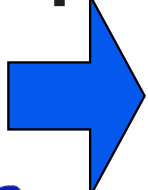


This hurts CPI.

Can we do better?

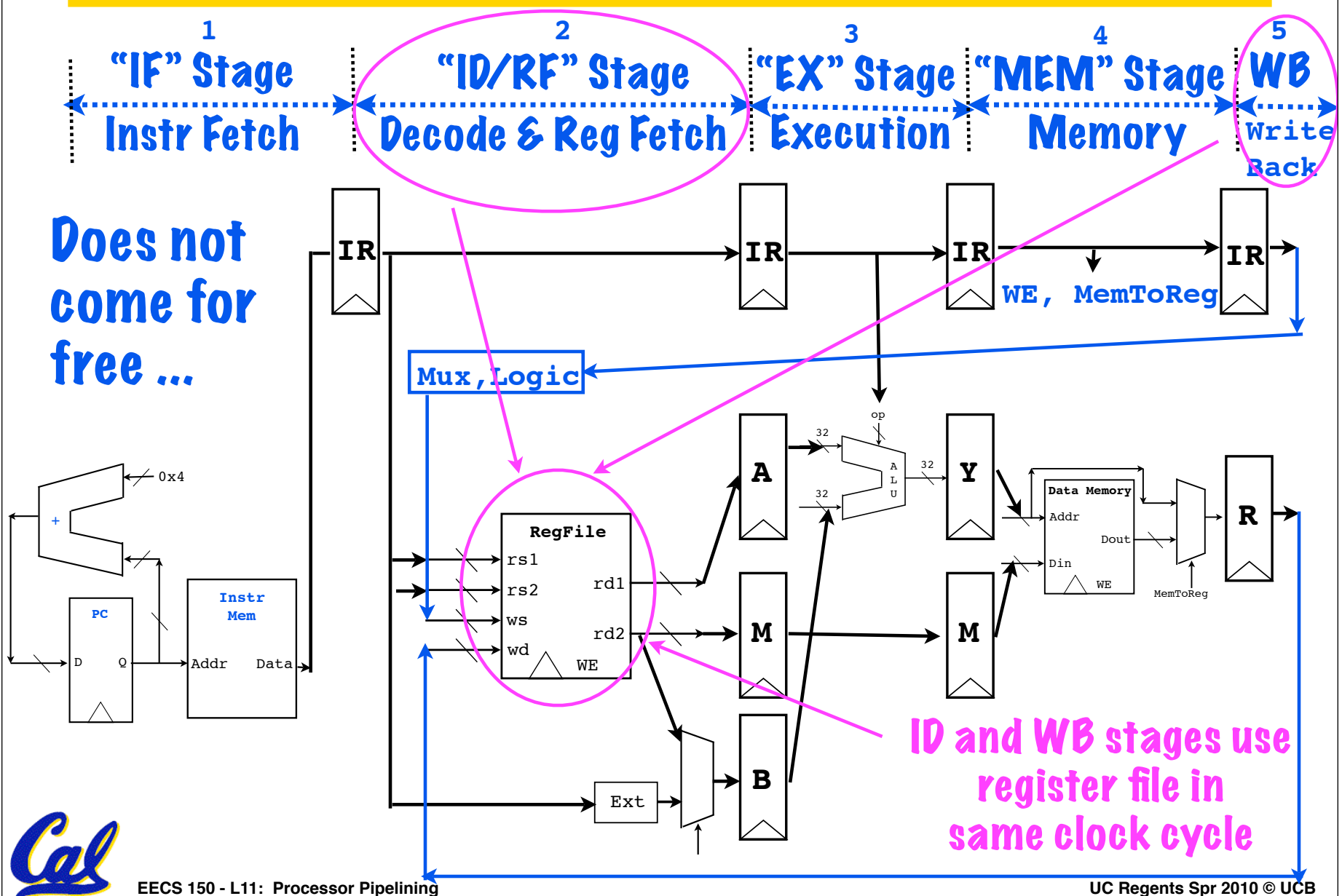


The Hazard Resolution Toolkit

- * **Stall** earlier instructions in pipeline.
- * **Forward** results computed in later pipeline stages to earlier stages.
- * **Add** new hardware or **rearrange** hardware design to eliminate hazard.
- * **Change ISA** to eliminate hazard.
- * **Kill** earlier instructions in pipeline.
-  **Make hardware handle concurrent requests** to eliminate hazard.



Structural hazard solution: concurrent use



Hazard Diagnosis



Data Hazards: Read After Write

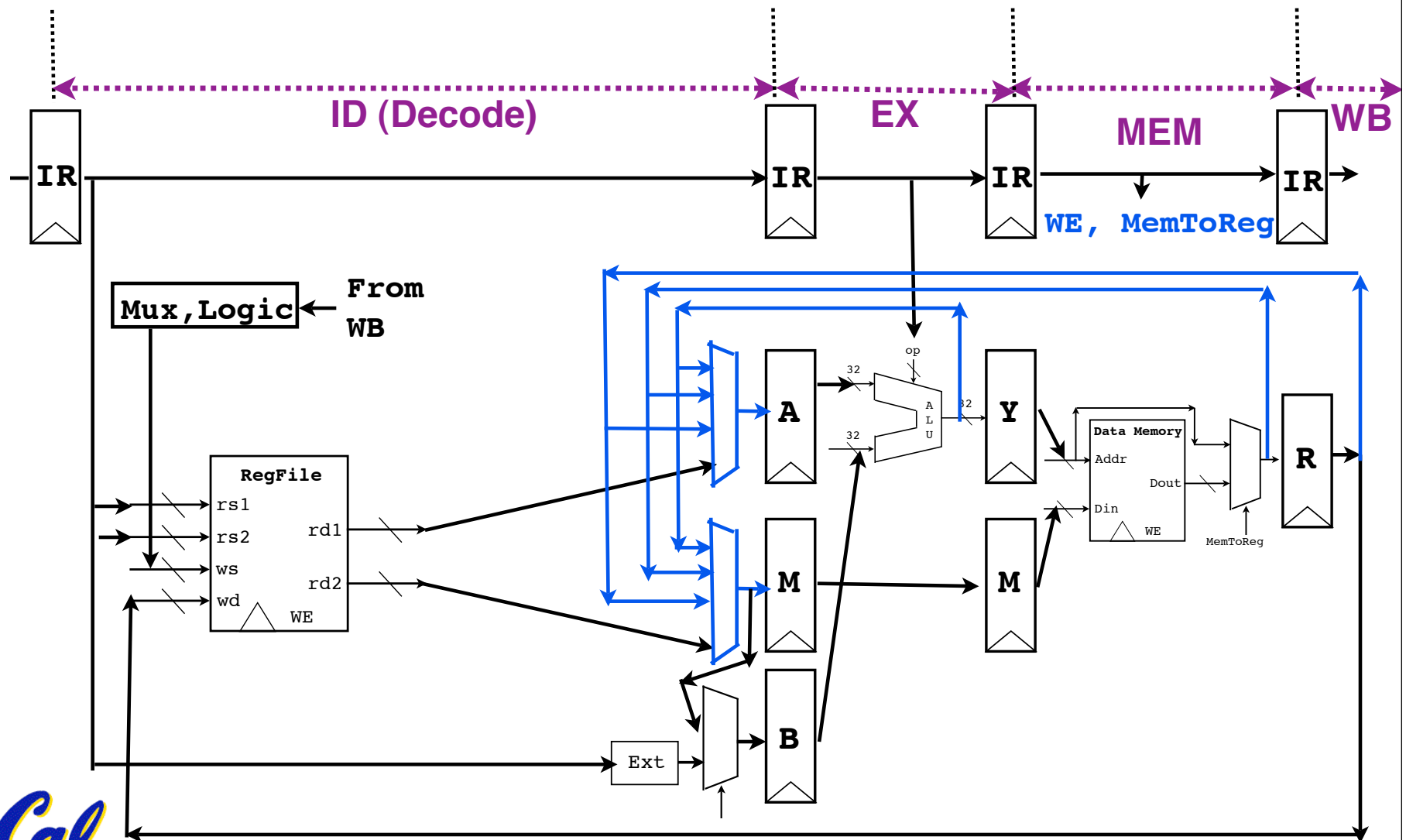
Read After Write (RAW) hazards.

Instruction I2 expects to read a data value written by an earlier instruction, but I2 executes “too early” and reads the wrong copy of the data.

Classic solution: use forwarding heavily, fall back on stalling when forwarding won't work or slows down the critical path too much.



Full bypass network ...

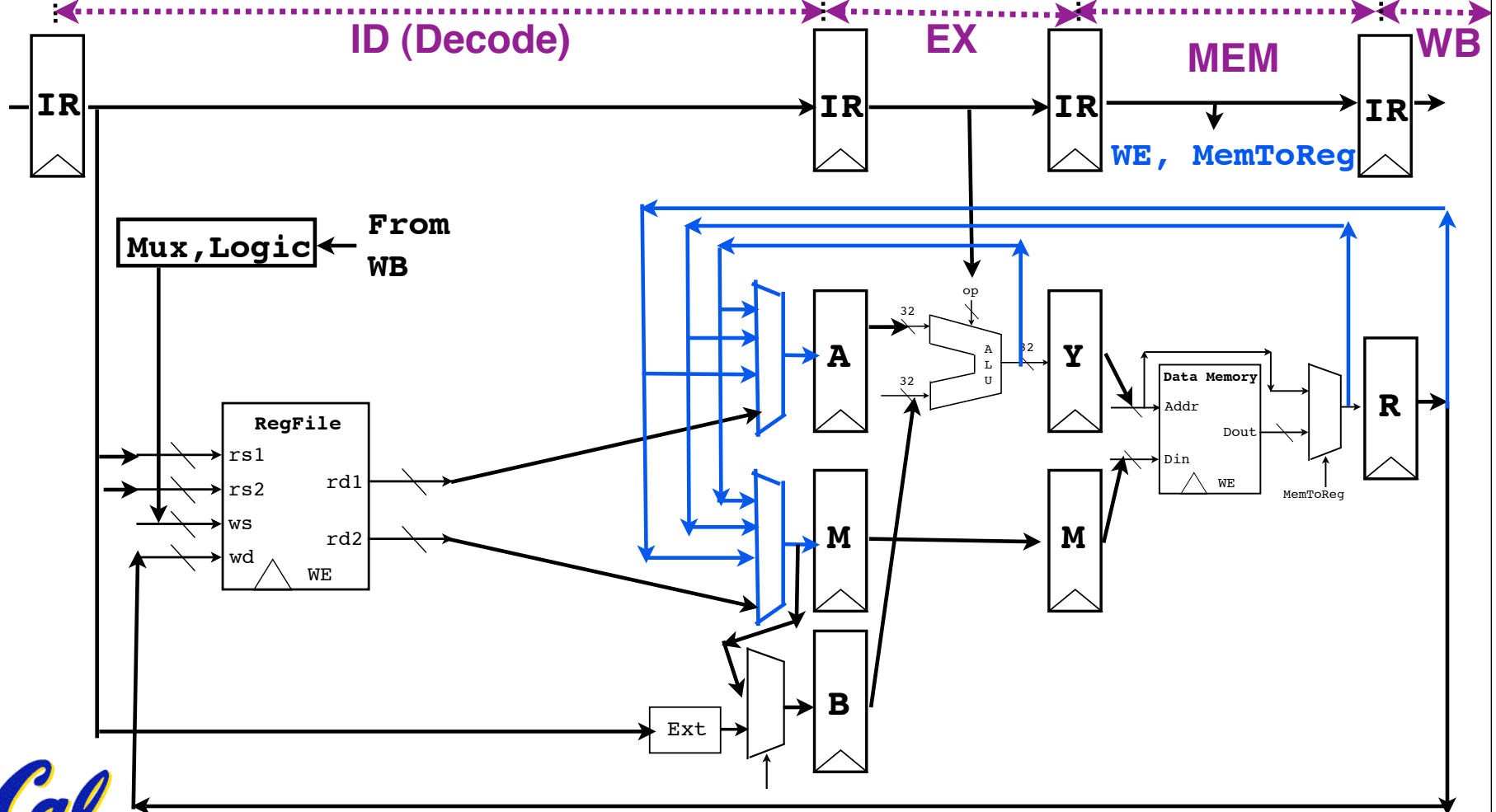


Common bug: Multiple forwards ...

ADD R4, R3, R2

OR R2, R3, R1 AND R2, R2, R1

Which do we forward from?

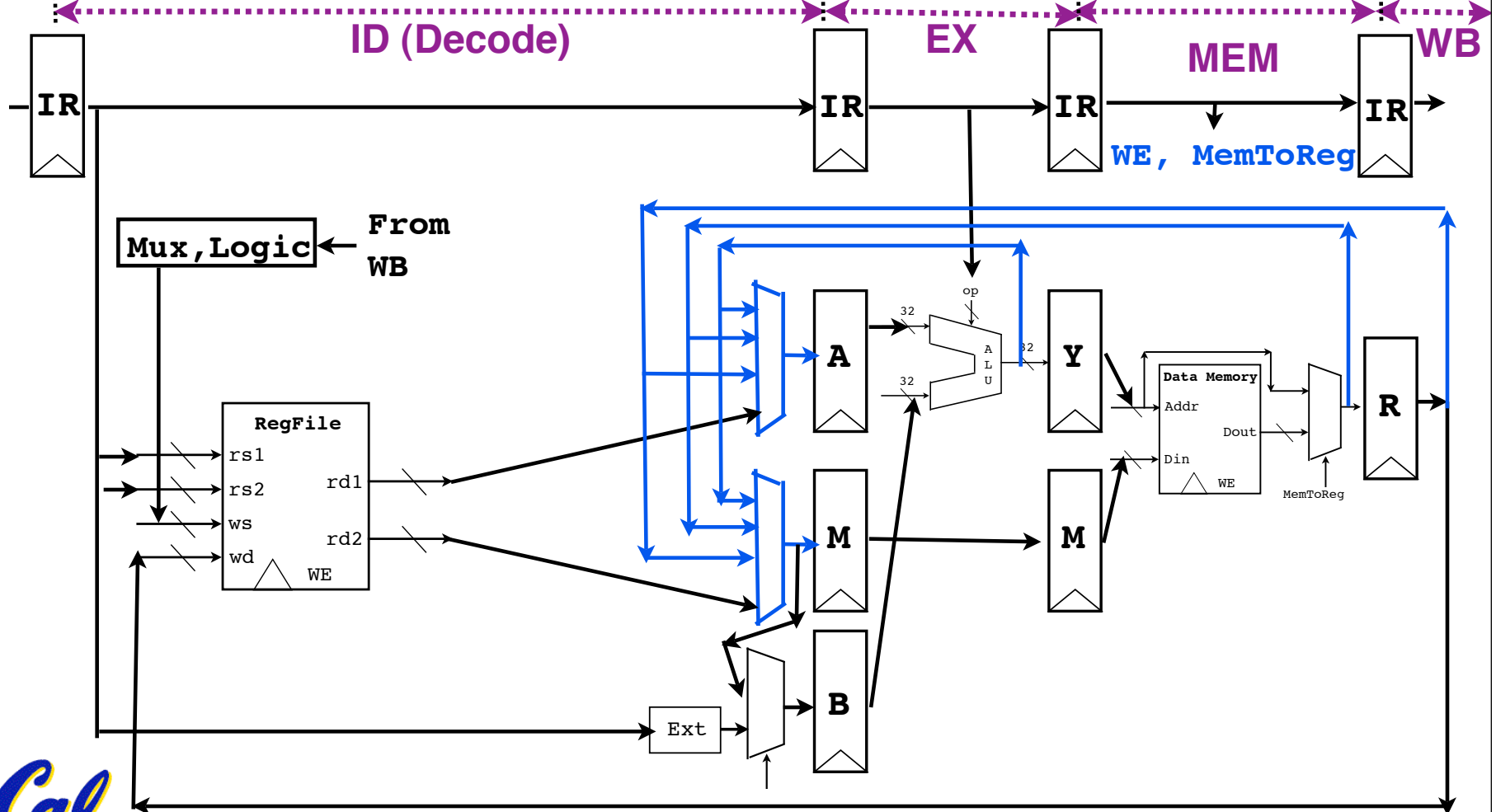


Common bug: Multiple forwards I ...

ADD R4, R0, R2

OR R0, R3, R1 AND R0, R2, R1

Which do we forward from?



LW and Hazards

Type	Instructions
arithmetic	addu, subu, addiu
logical	and, andi, or, ori, xor, xori, lui
shift	sll, sra, srl
compare	slt, slti, sltu, sltui
control	beq, bne, bgez, bltz, j, jr, jal
data transfer	lw, sw
Other:	break

**No load
“delay slot”**

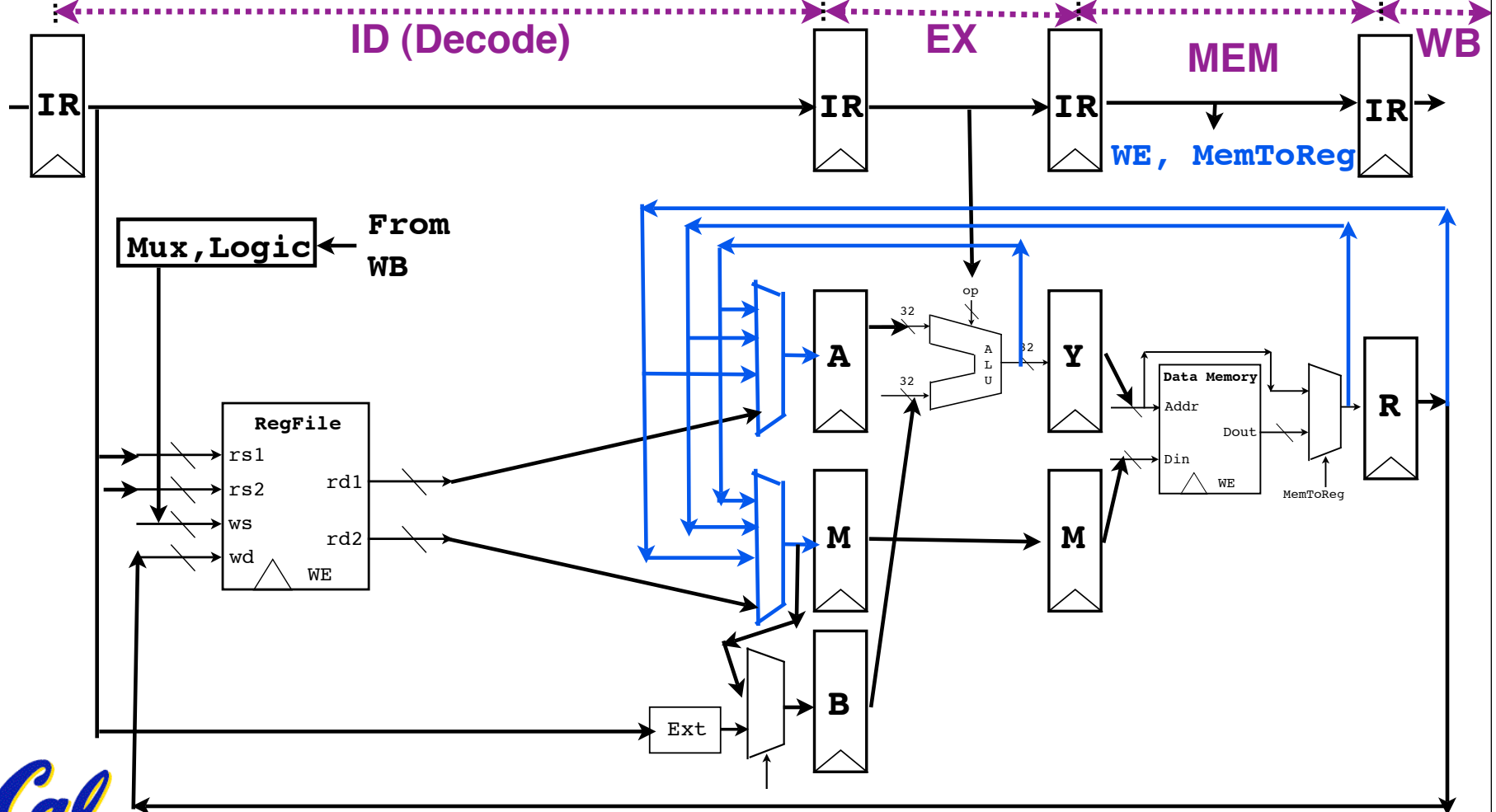


Questions about LW and forwarding

ADDIU R1 R1 24

OR R3,R3,R2 LW R1 128(R29)

Do we need to stall?

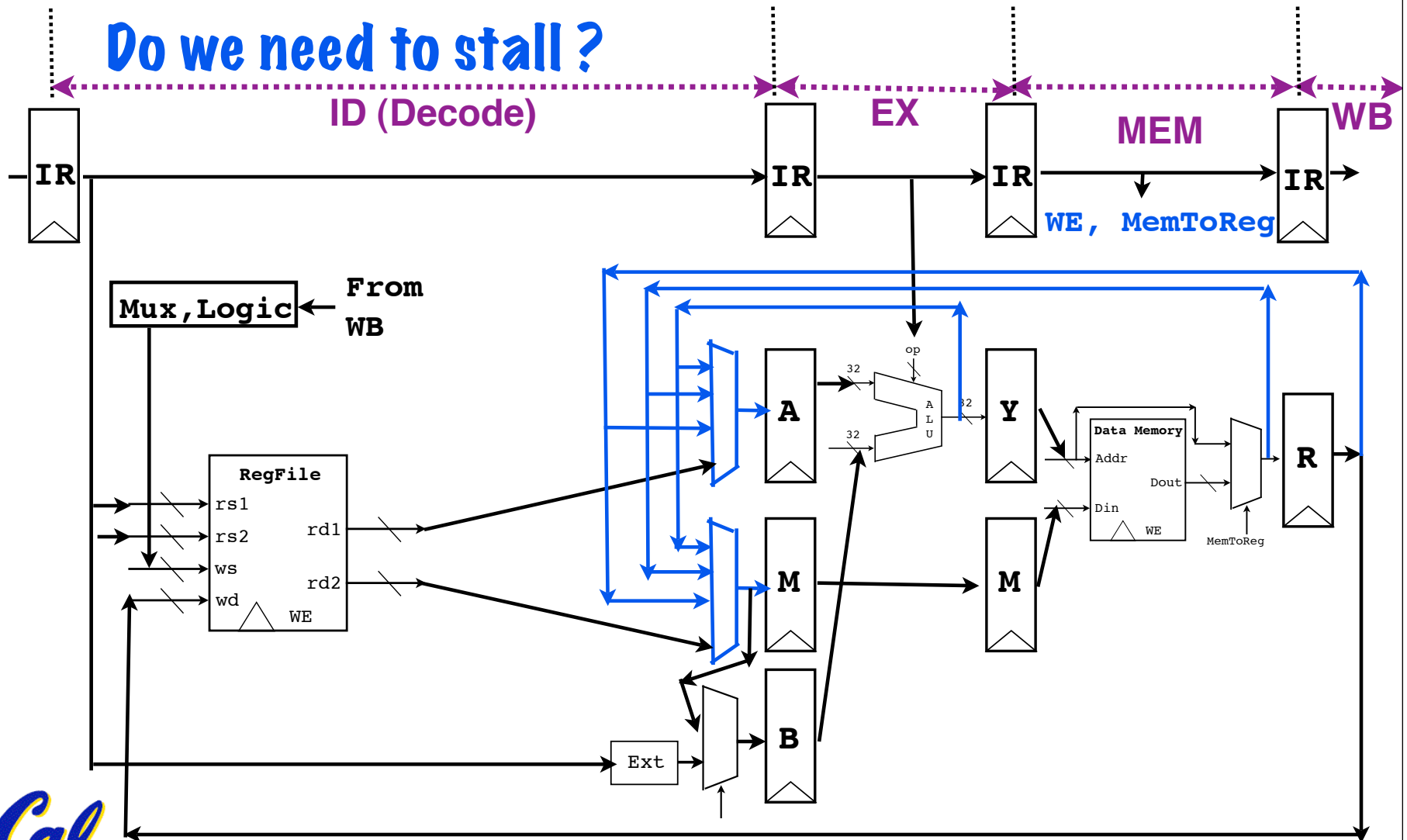


Questions about LW and forwarding

ADDIU R1 R1 24

LW R1 128(R29) OR R1,R3,R1

Do we need to stall?

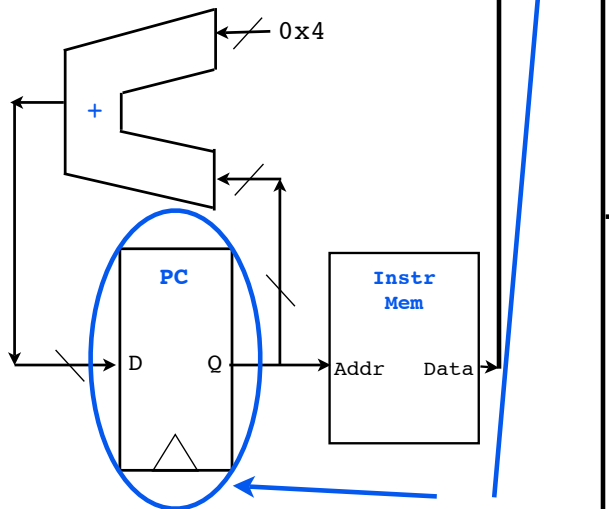


Resolving a RAW hazard by stalling



Sample program

ADD R4, R3, R2
OR R5, R4, R2



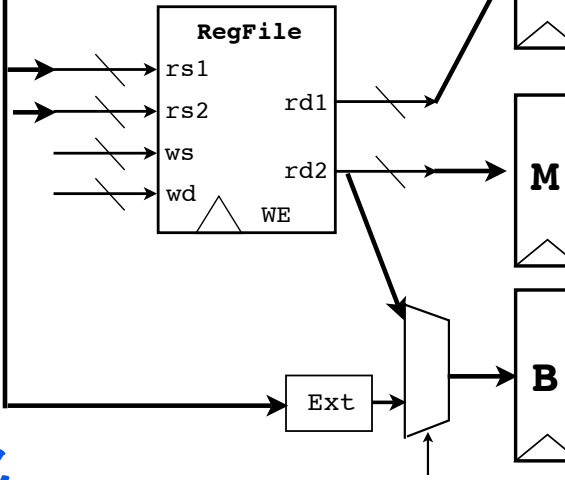
Freeze PC and IR until stall is over.

OR R5, R4, R2

Keep executing OR instruction until R4 is ready. Until then, send NOPS to IR 2/3.

ADD R4, R3, R2

Let ADD proceed to WB stage, so that R4 is written to regfile.



New datapath hardware

(1) Mux into IR 2/3 to feed in NOP.

(2) Write enable on PC and IR 1/2



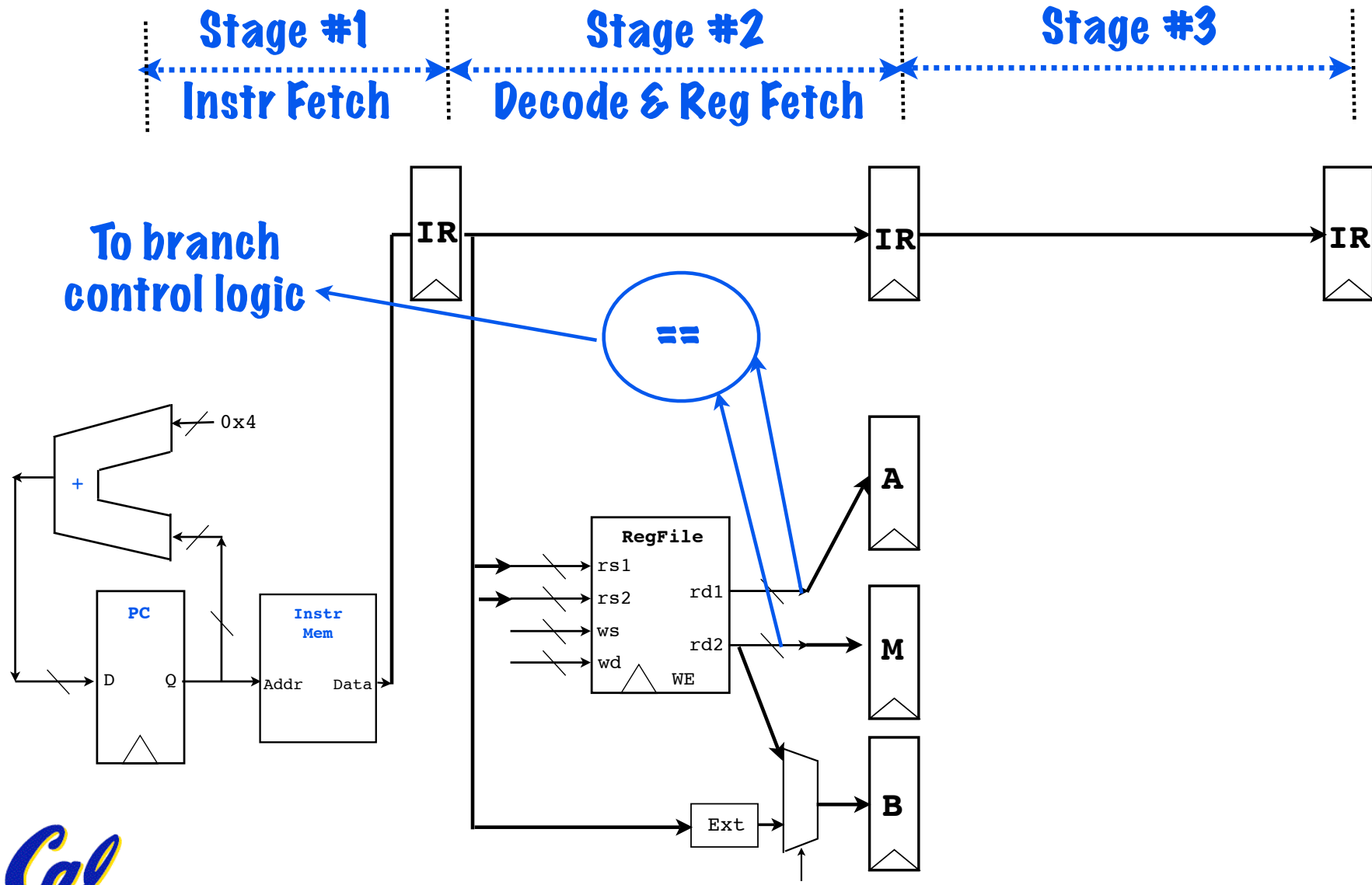
Branches and Hazards

Single
"delay slot"

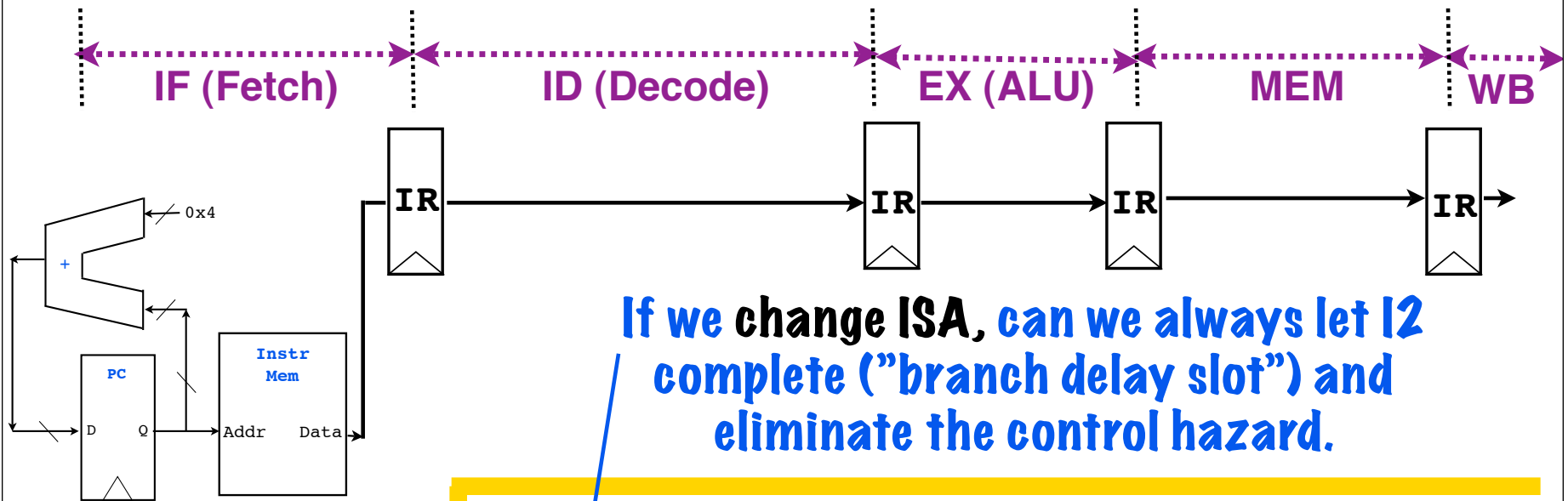


Type	Instructions
arithmetic	addu, subu, addiu
logical	and, andi, or, ori, xor, xori, lui
shift	sll, sra, srl
compare	slt, slti, sltu, sltui
control	beq, bne, bgez, bltz, j, jr, jal
data transfer	lw, sw
Other:	break

Recall: Control hazard and hardware



Recall: After more hardware, change ISA



If we change ISA, can we always let I2 complete ("branch delay slot") and eliminate the control hazard.

Sample Program
(ISA w/o branch delay slot)

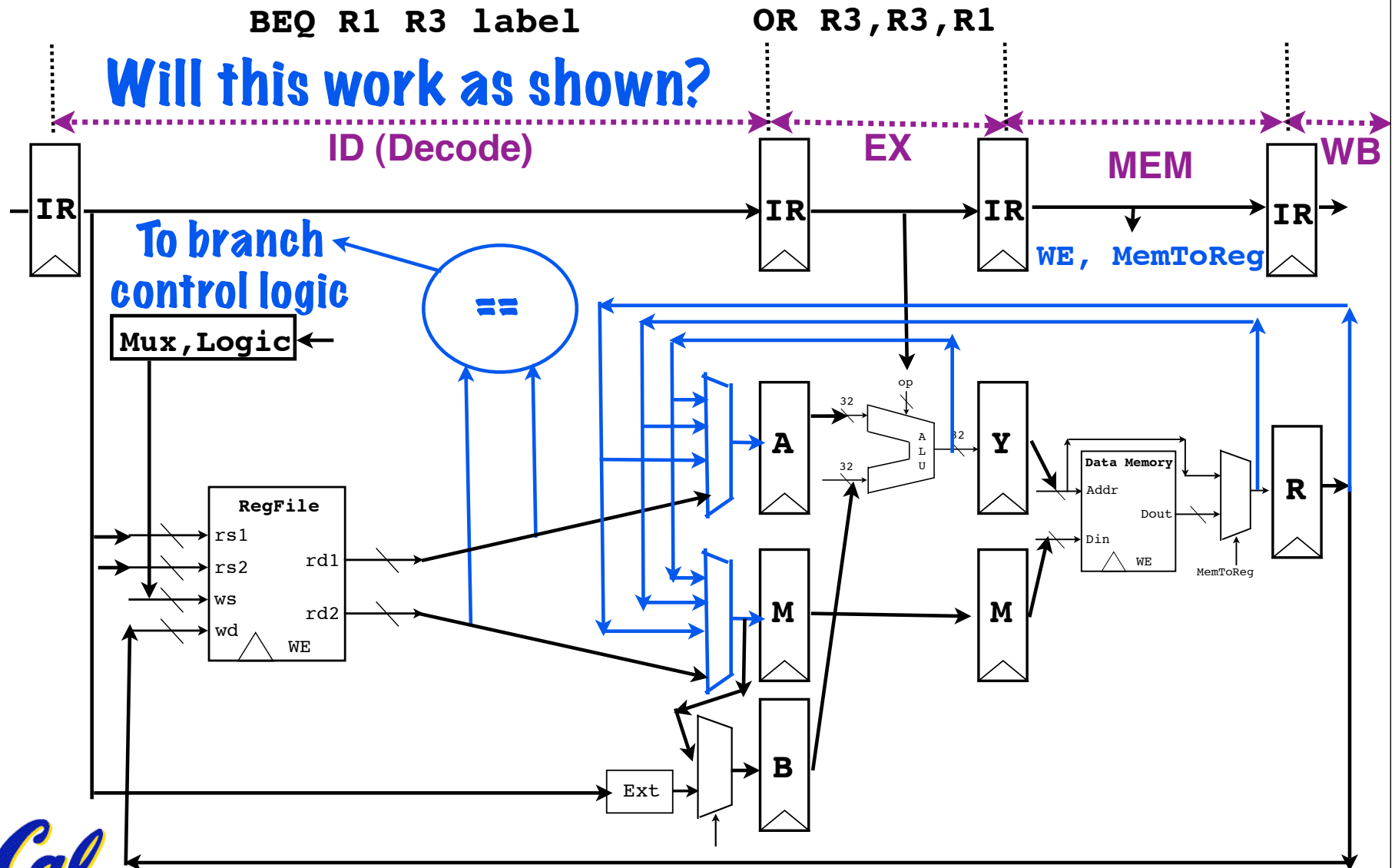
I1: BEQ R4, R3, 25
I2: AND R6, R5, R4
I3: SUB R1, R9, R8

Time:	t1	t2	t3	t4	t5	t6	t7	t8
Inst								
I1:	IF	ID	EX	MEM	WB			
I2:		IF						
I3:								
I4:								
I5:								
I6:								

Annotations:
 - A blue circle highlights the ID stage of I1 at t2.
 - A blue circle highlights the IF stage of I2 at t2.
 - A blue arrow points from the text "ID stage computes if branch is taken" to the ID stage of I1.
 - A blue arrow points from the text "If branch is taken, this instruction MUST NOT complete!" to the IF stage of I2.



Question about branch and forwards:



Lessons learned

- * **Pipelining is hard**
- * **Study every instruction**
- * **Write test code in advance**
- * **Think about interactions ...**



Lessons learned

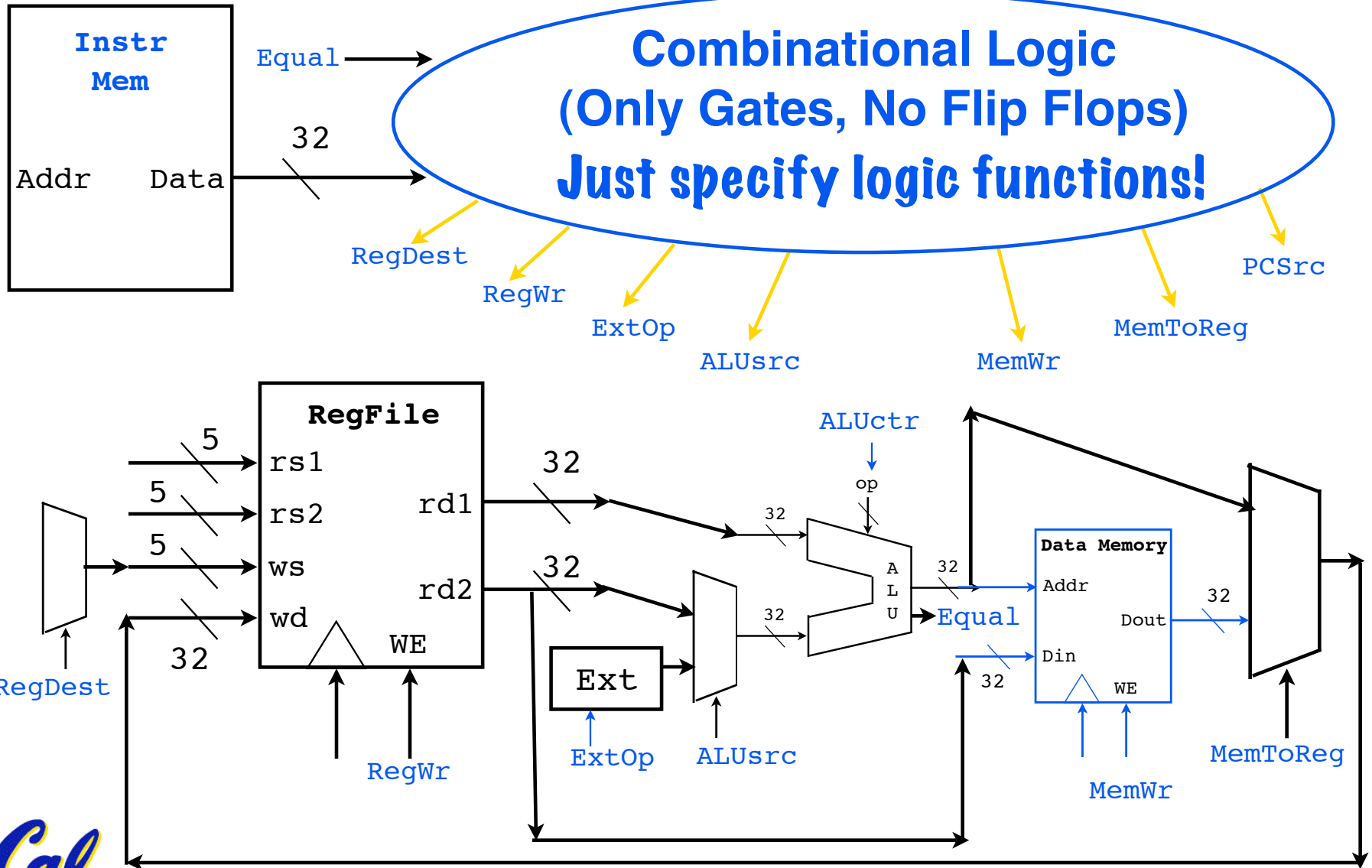
- * **Pipelining is hard**
- * **Study every instruction**
- * **Write test code in advance**
- * **Think about interactions ...
between forwarding, branch and
jump delay slots, R0 issues
LW issues ... a long list!**



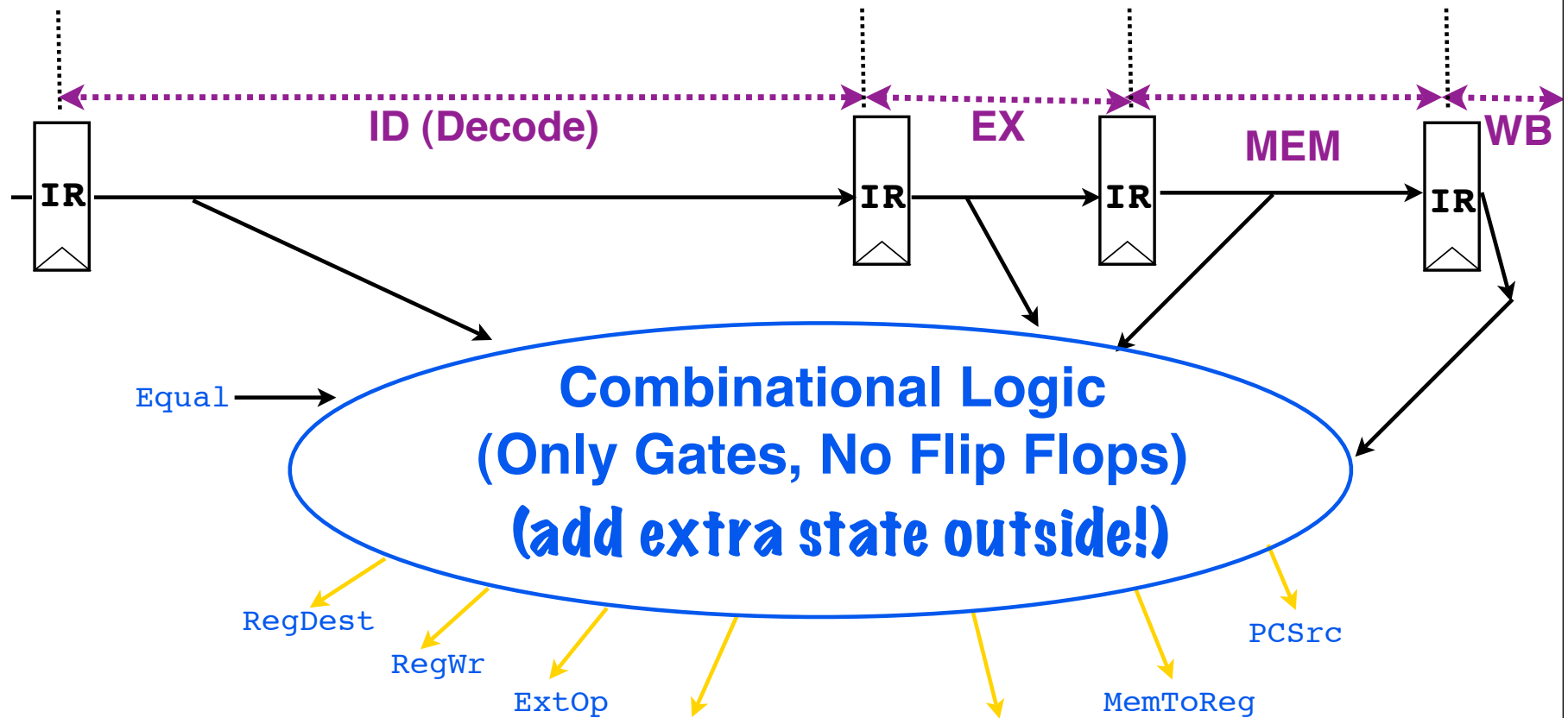
Control Implementation



Recall: What is single cycle control?



In pipelines, all IR registers are used



A “conceptual” design -- for shortest critical path, IR registers may hold decoded info, not the complete 32-bit instruction



Exceptions and Interrupts

Exception: An unusual event happens to an instruction during its execution. **Examples:** divide by zero, undefined opcode.

Interrupt: Hardware signal to switch the processor to a new instruction stream. **Example:** a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting).



Challenge: Precise Interrupt / Exception

Definition:

(or exception)

It must appear as if an interrupt is taken between two instructions (say I_i and I_{i+1})

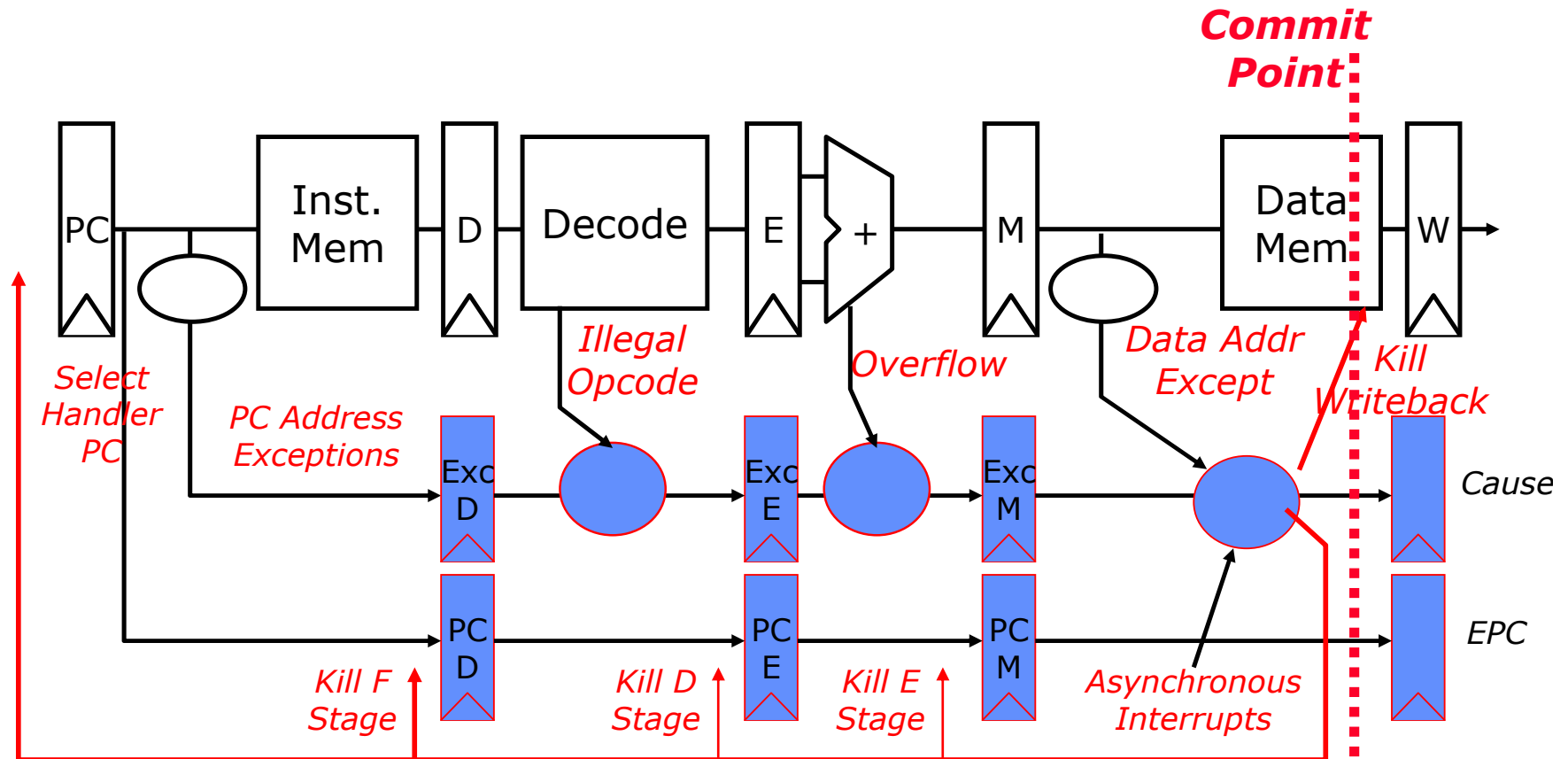
- the effect of all instructions up to and including I_i is totally complete
- no effect of any instruction after I_i has taken place

The interrupt handler either aborts the program or restarts it at I_{i+1} .

Follows from the “contract” between the architect and the programmer ...



Precise Exceptions in Static Pipelines



Key observation: architected state only change in memory and register write stages.



Thursday:

Thr 2/25

Lec #12: Project Introduction: Memory Blocks, Project Specification:
Reading: Pages 111 thru 137 of the [Virtex-5 User's Guide](#)

