Introduction

- Pipelining became a universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”
    - The potential overlap among instructions
- Beyond this, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMP processors
  - Compiler-based static approaches
    - Not as successful outside of scientific applications

Pipelining

- Multiple instructions are overlapped in execution
- Takes advantage of parallelism
- Increases instruction throughput, but does not reduce the time to complete an instruction
- Ideal case:
  - The time per instruction: Time per instruction on unpipelined machine / number of pipe stages
  - Speedup: the number of pipe stages
- Intel Pentium 4: 20 stages; Cedar Mill: 31 stages
- Intel Core i7: 14 stages. Why?

The Basics of a RISC Instruction Set

- **Key Proprieties:**
  - **All operations on data apply to data in registers**
  - **The only operations that affect memory are load and store operations**
  - **The instruction formats are few in number, with all instructions typically being one size.**

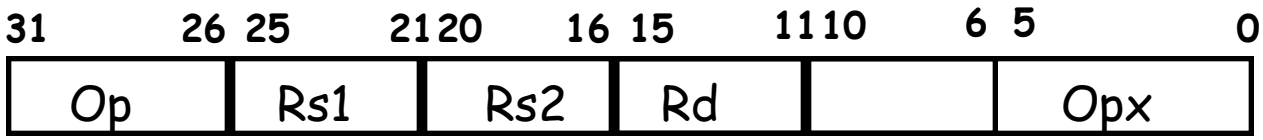
Basic Pipelining: A "Typical" RISC ISA

- 32-bit fixed format instruction (3 formats)
  - ALU instructions, Load/Store, Branches and Jumps
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions /Jump

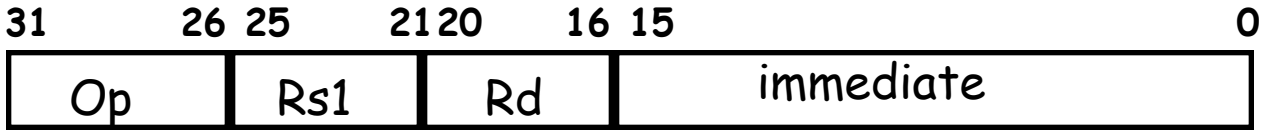
see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Basic Pipelining: e.g., MIPS (MIPS)

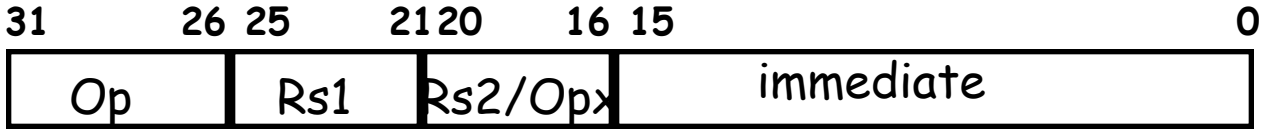
Register-Register



Register-Immediate



Branch



Jump / Call

A simple implementation of a RISC Instruction Set

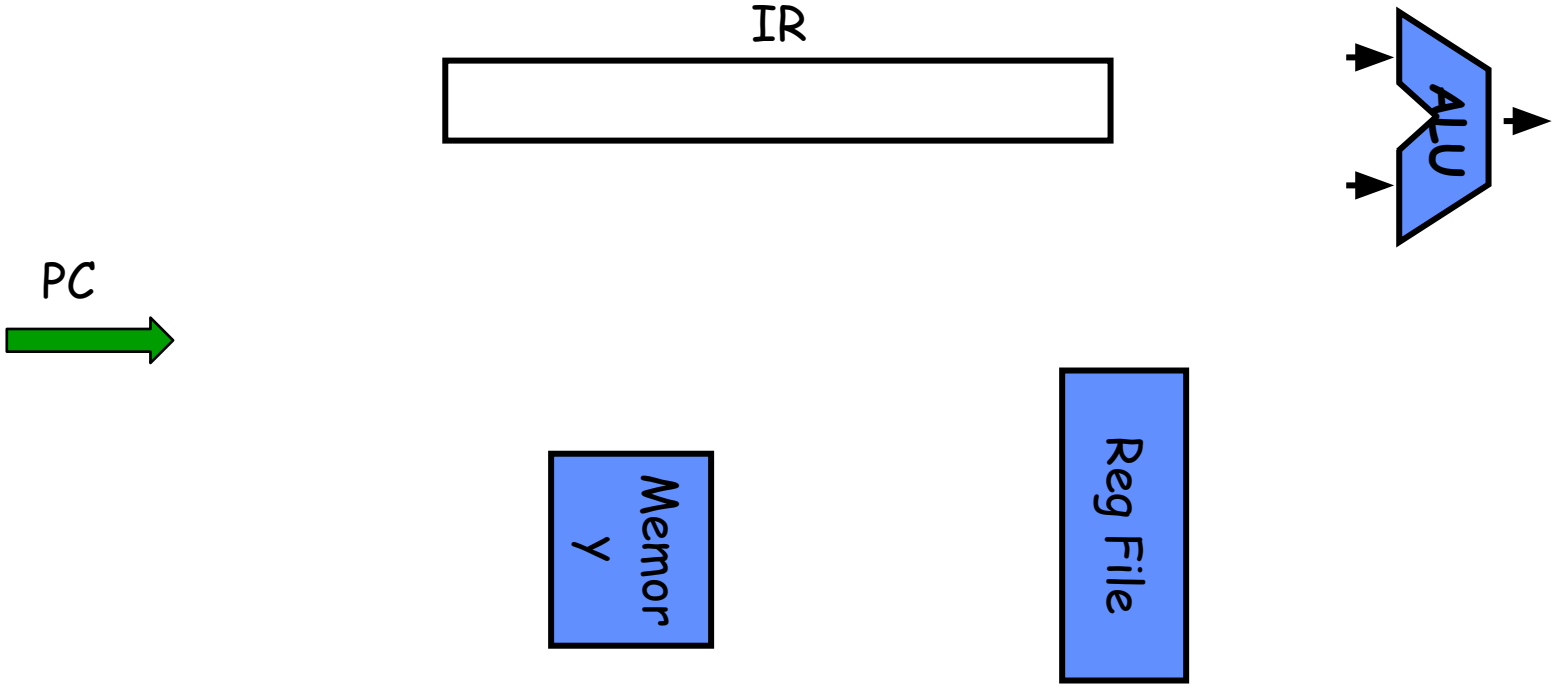
- **Every instruction can be implemented in at most 5 cycles.**
  - **Instruction fetch cycle (IF): send PC to memory and fetch the current instruction; Update the PC to the next PC by adding 4.**
  - **Instruction decode/register fetch cycle (ID): decode the instruction and read the registers. For a branch, do the equality test on the registers, and compute the target address**
  - **Execution/effective address cycle (EX): The ALU operates on the operands prepared in the prior cycle, performing:**
    - » **Memory reference: ALU adds the base register and the offset. LW 100(R1) => 100 + R1**
    - » **Register-Register ALU instruction: ALU performs the operation specified by the opcode on the values read from the register file. ADD R1,R2,R3 => R1 + R2**
    - » **Register-Immediate ALU instruction: ALU performs the operation specified by the opcode on the first values read from the register file and the immediate. ADD R1,3,R2 => R1 + 3**

A simple implementation of a RISC Instruction Set

- **Every instruction can be implemented in at most 5 cycles.**
  - **Memory access (MEM). Read the memory using the effective address computed in the previous cycle if the instruction is a load; If it is a store, write the data to the memory**
  - **Write-back cycle (WB). For Register-Register ALU instruction or load instruction, write the result into the register file, whether it comes from the memory (for a load) or from the ALU (for an ALU instruction)**
- **Branch instructions require 2 cycles, store instructions require 4 cycles, others require 5 cycles.**

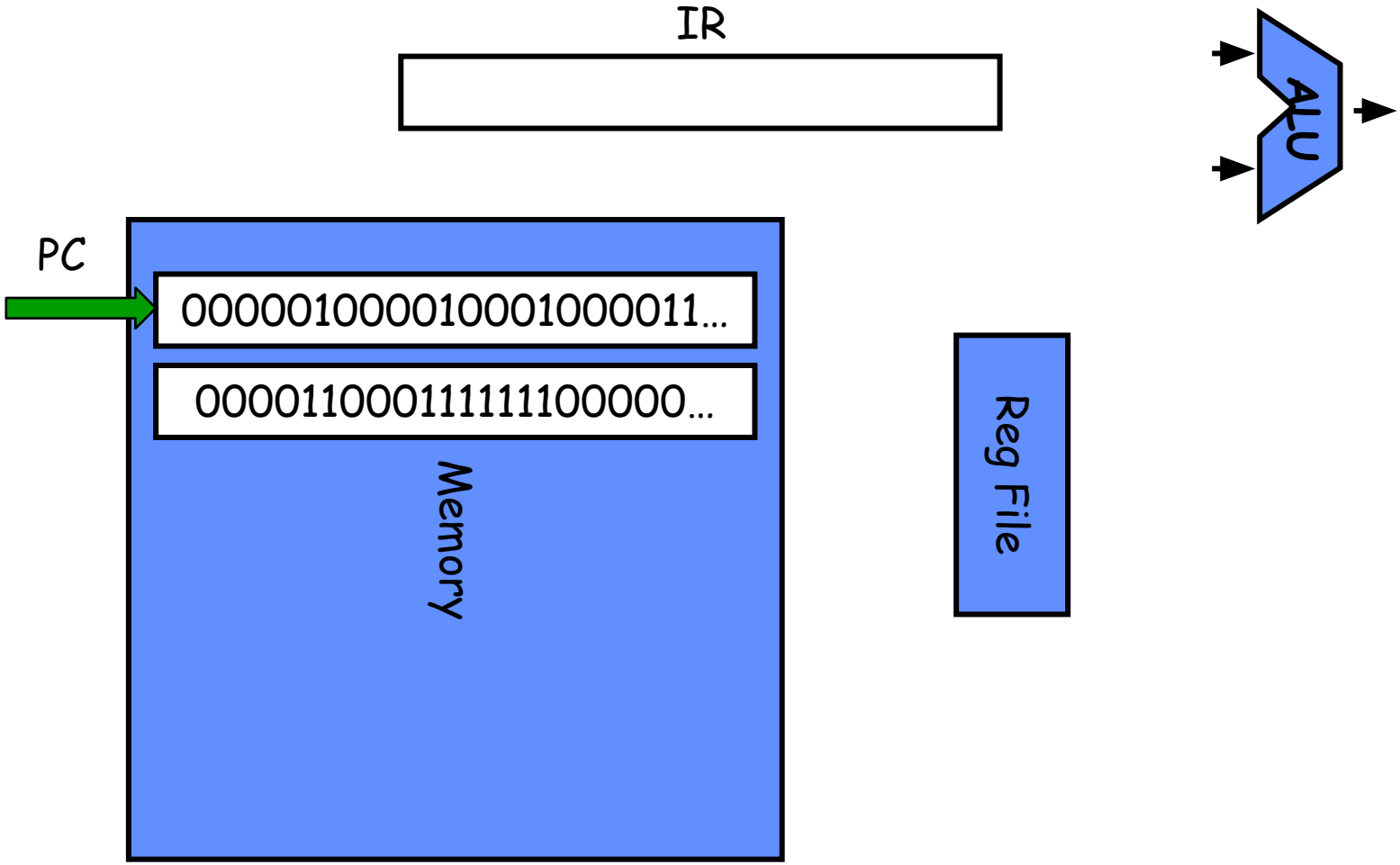
An example of an instruction execution

- **ADD R3, R1, R2 => R3 = R1 + R2;**



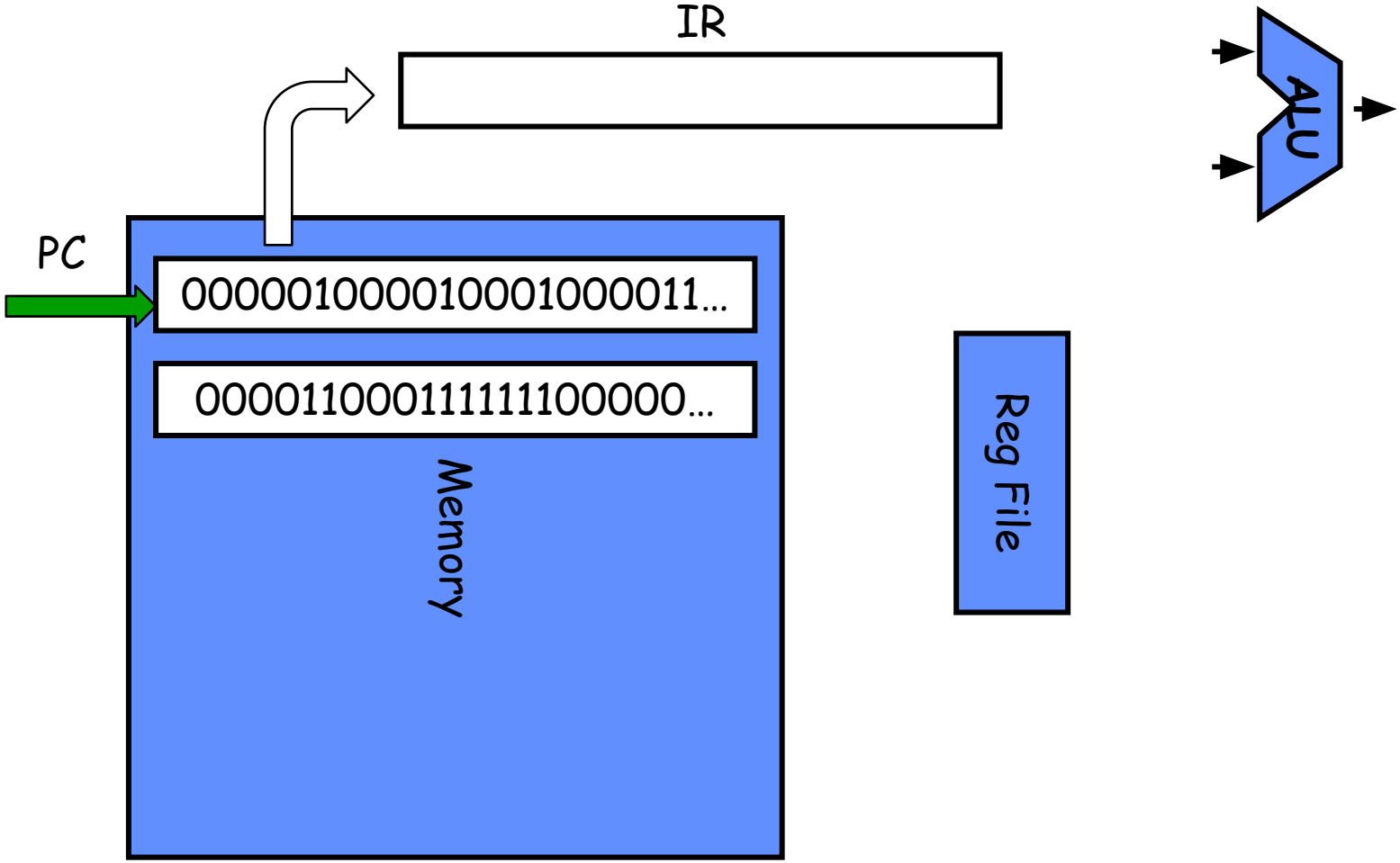
Instruction Fetch (IF)

- **ADD R3, R1, R2 => R3 = R1 + R2;**



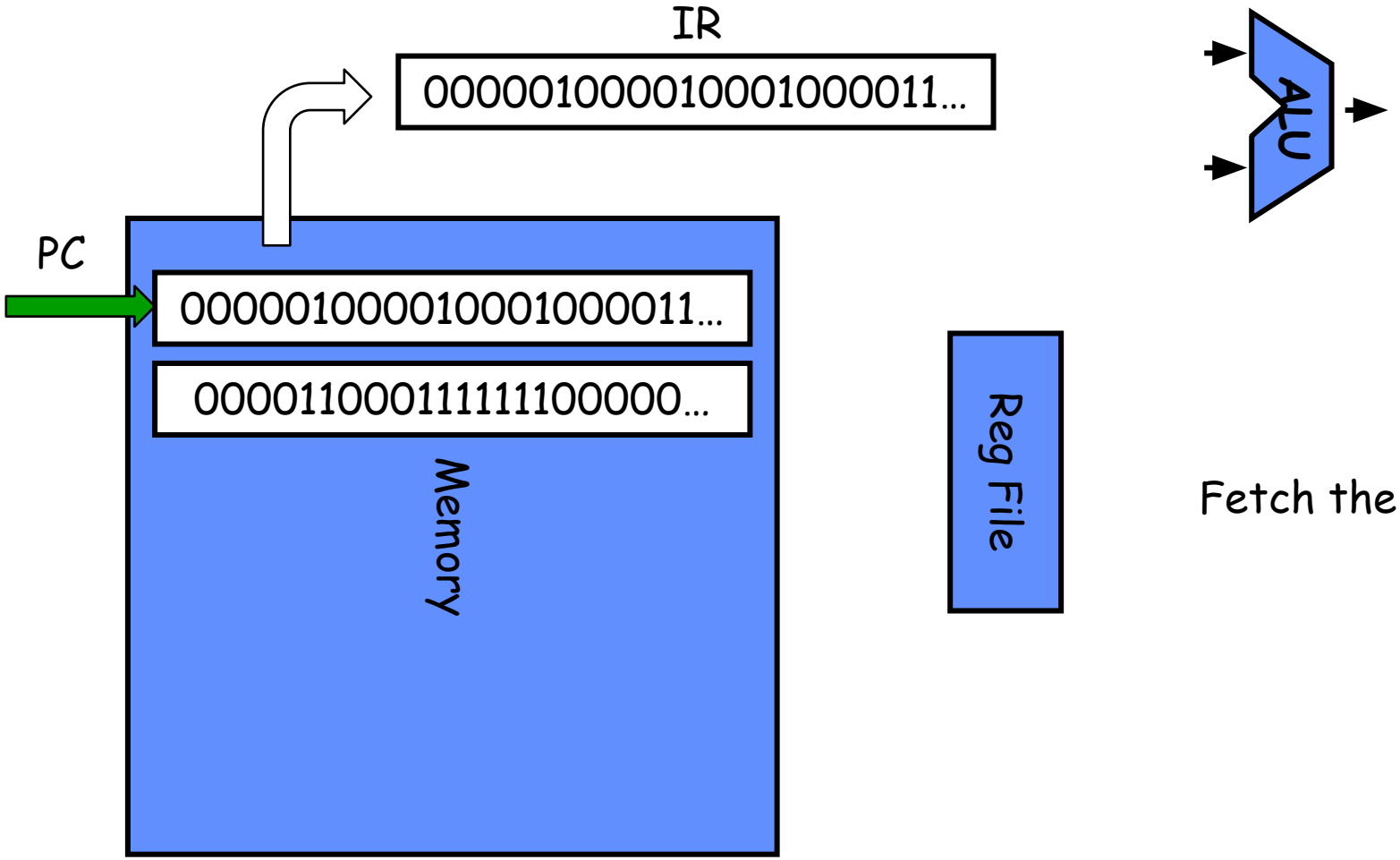
Instruction Fetch (IF)

- **ADD R3, R1, R2 => R3 = R1 + R2;**



Instruction Fetch (IF)

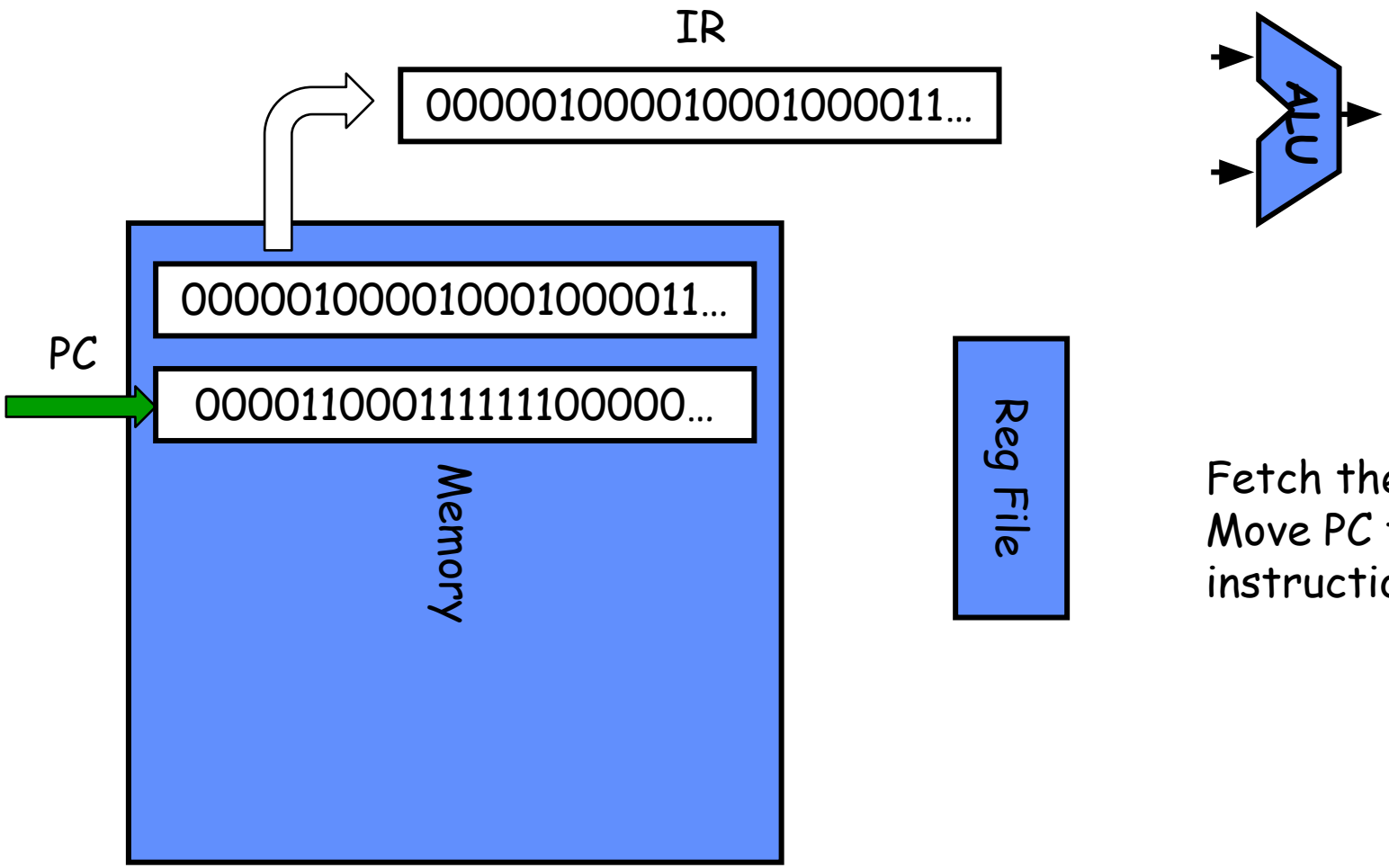
- **ADD R3, R1, R2 => R3 = R1 + R2;**



Fetch the instruction

Instruction Fetch (IF)

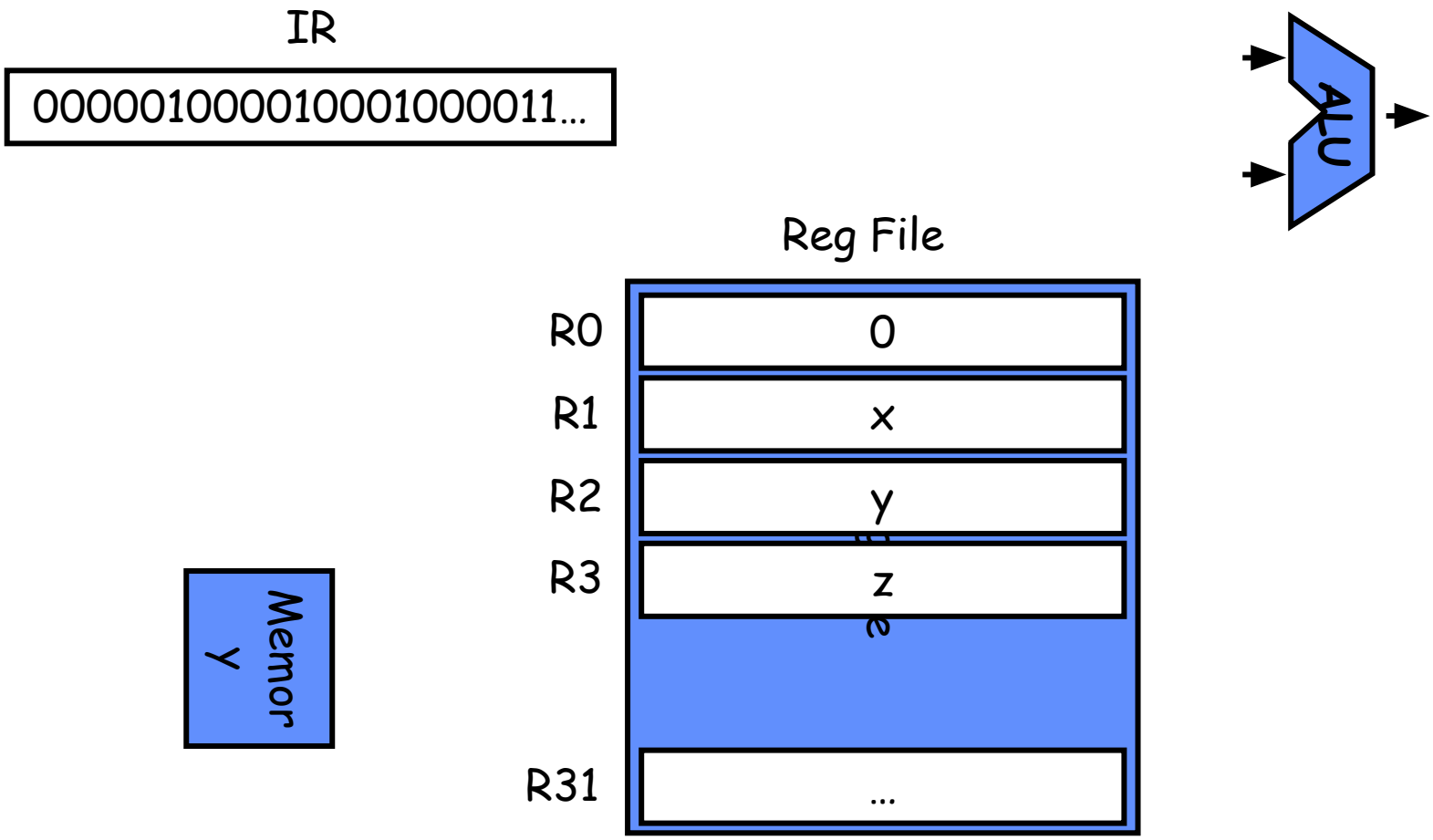
- **ADD R3, R1, R2 => R3 = R1 + R2;**



Fetch the instruction
Move PC to the next instruction

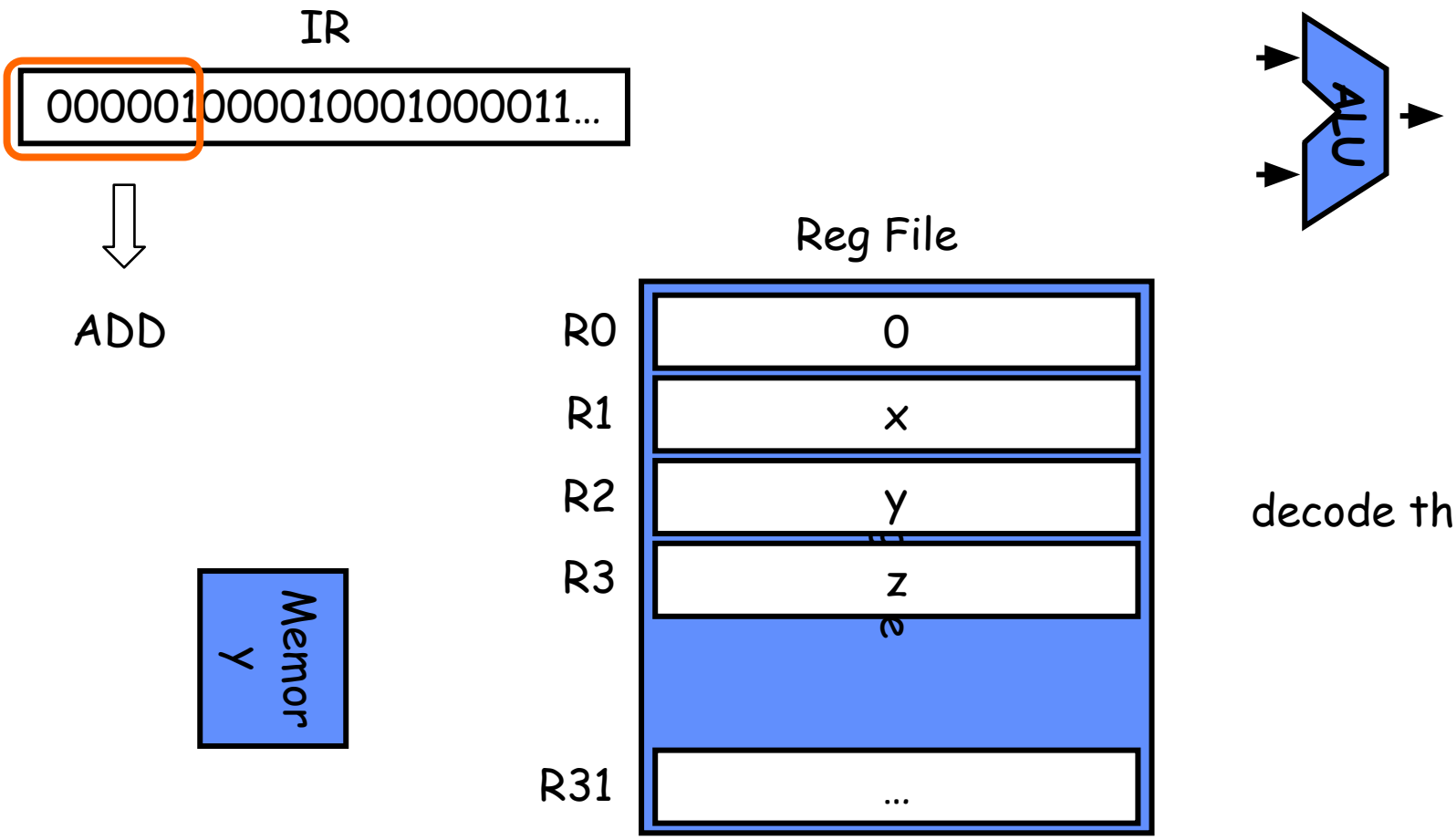
Instruction Decode (ID)

- **ADD R3, R1, R2 => R3 = R1 + R2;**



Instruction Decode (ID)

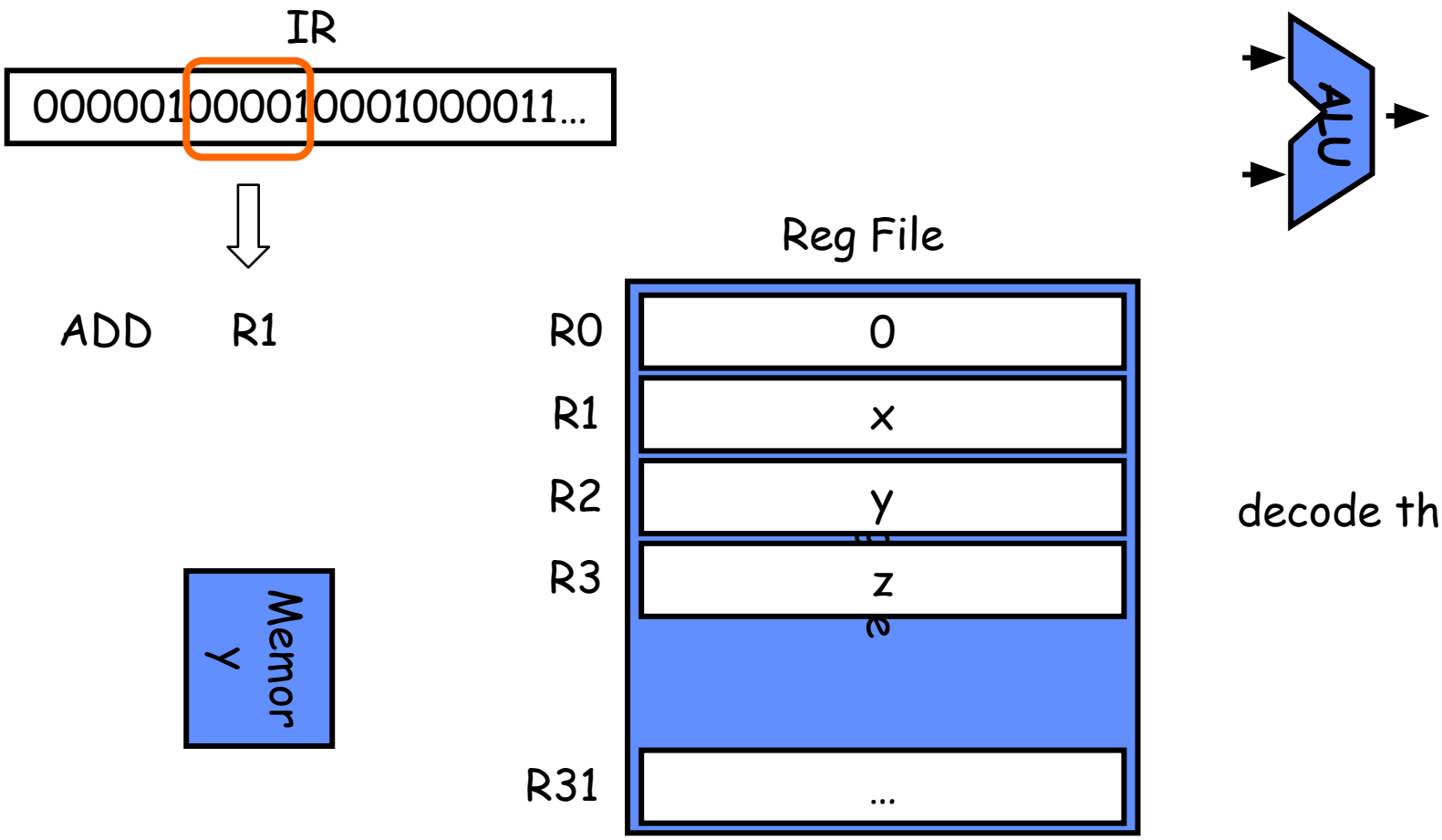
- **ADD R3, R1, R2 => R3 = R1 + R2;**



decode the instruction

Instruction Decode (ID)

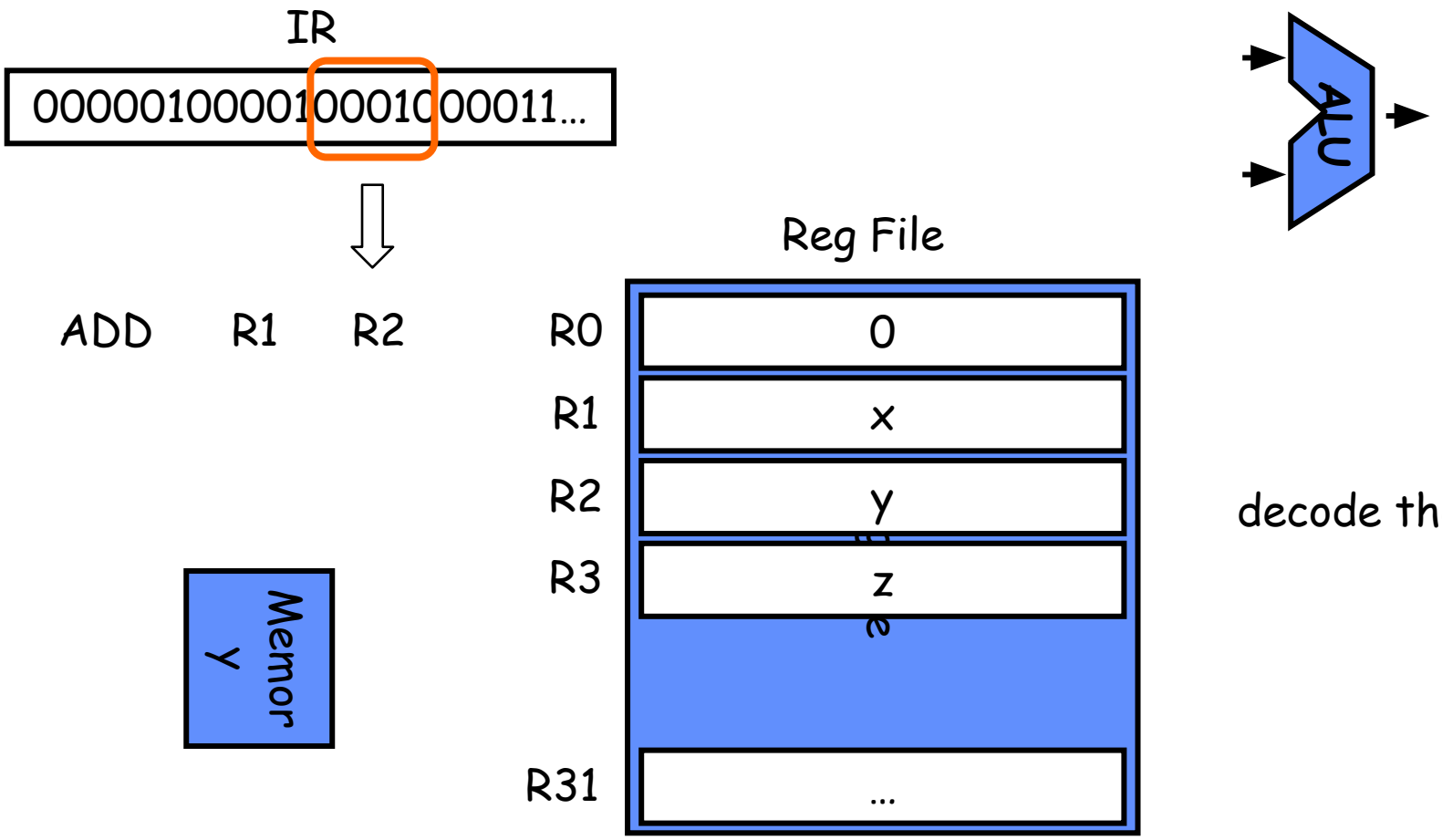
- **ADD R3, R1, R2 => R3 = R1 + R2;**



decode the instruction

Instruction Decode (ID)

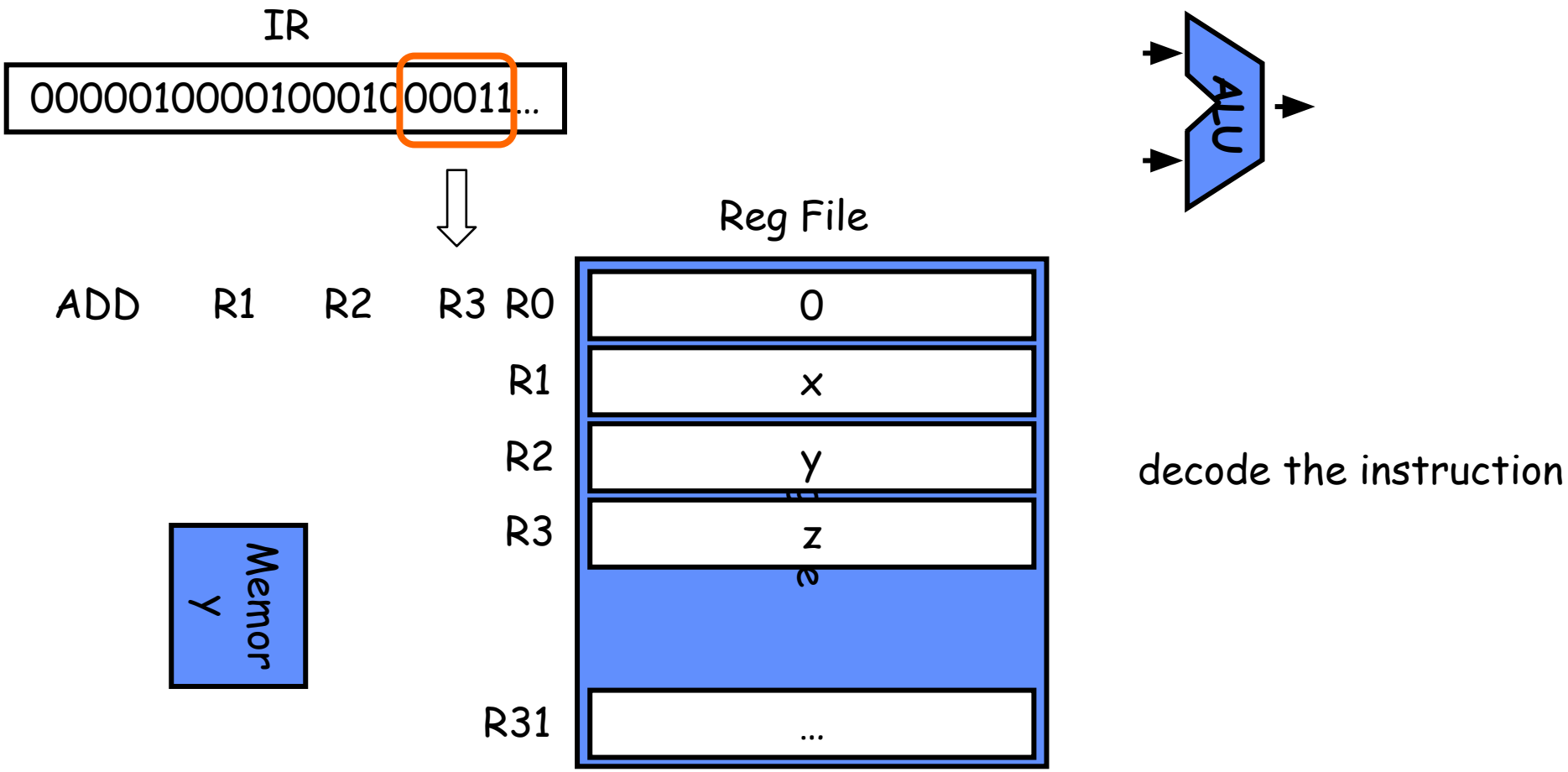
- **ADD R3, R1, R2 => R3 = R1 + R2;**



decode the instruction

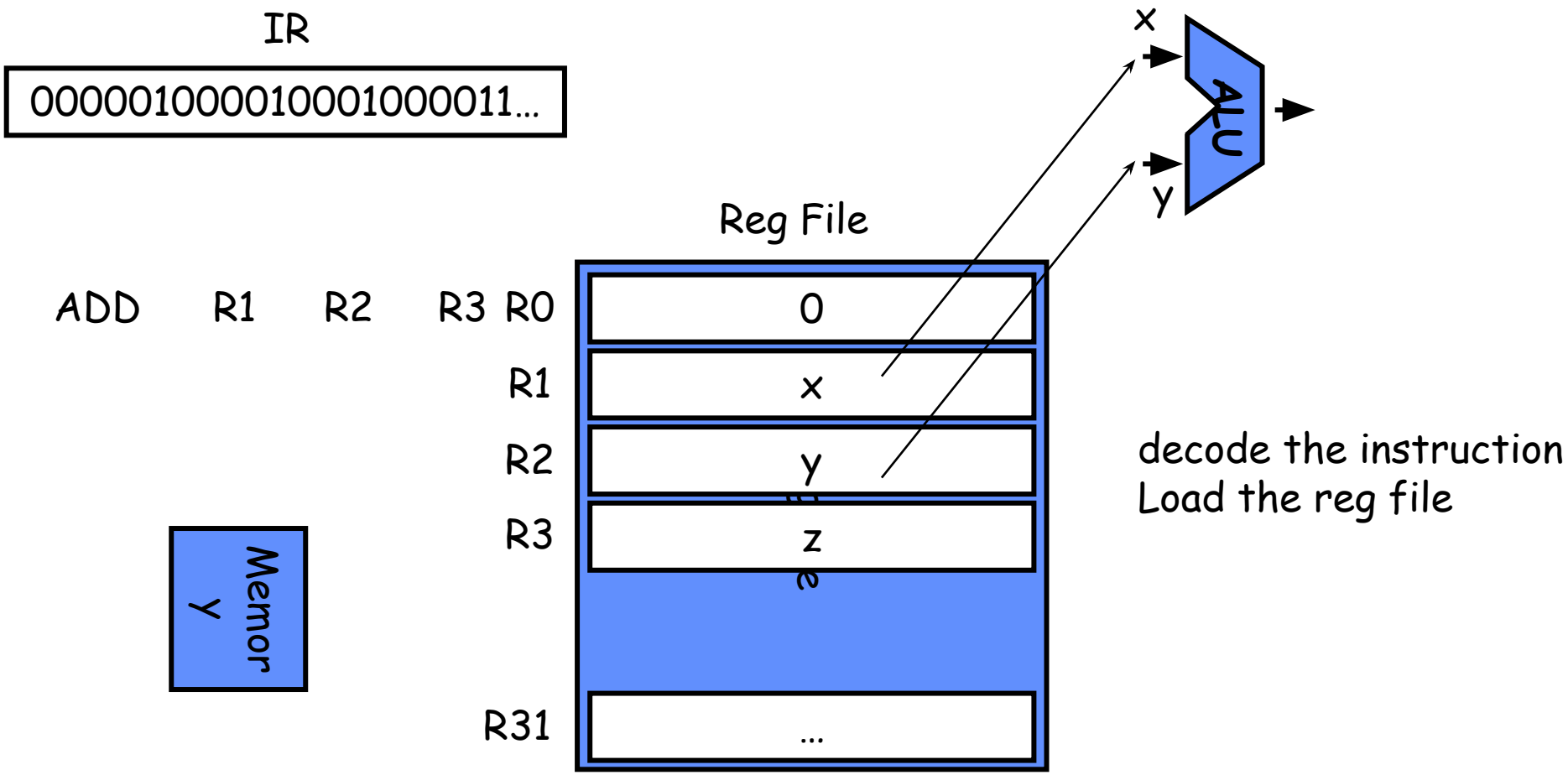
Instruction Decode (ID)

- **ADD R3, R1, R2 => R3 = R1 + R2;**



Instruction Decode (ID)

- **ADD R3, R1, R2 => R3 = R1 + R2;**

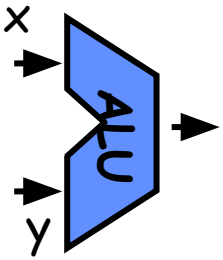


Execution (EX)

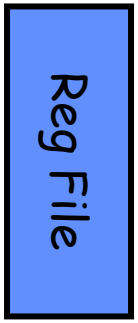
- **ADD R3, R1, R2 => R3 = R1 + R2;**

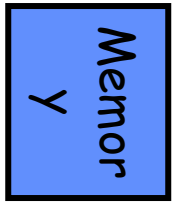
IR

00000100001000100001 1...



Execute the operation



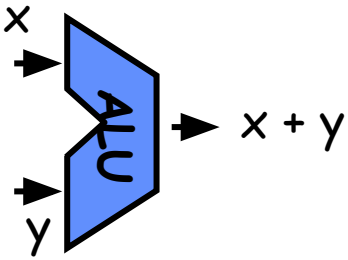


Execution (EX)

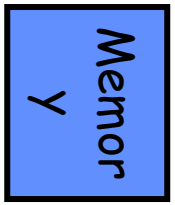
- **ADD R3, R1, R2 => R3 = R1 + R2;**

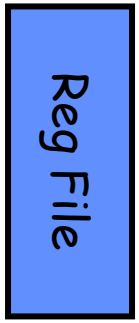
IR

00000100001000100001 1...



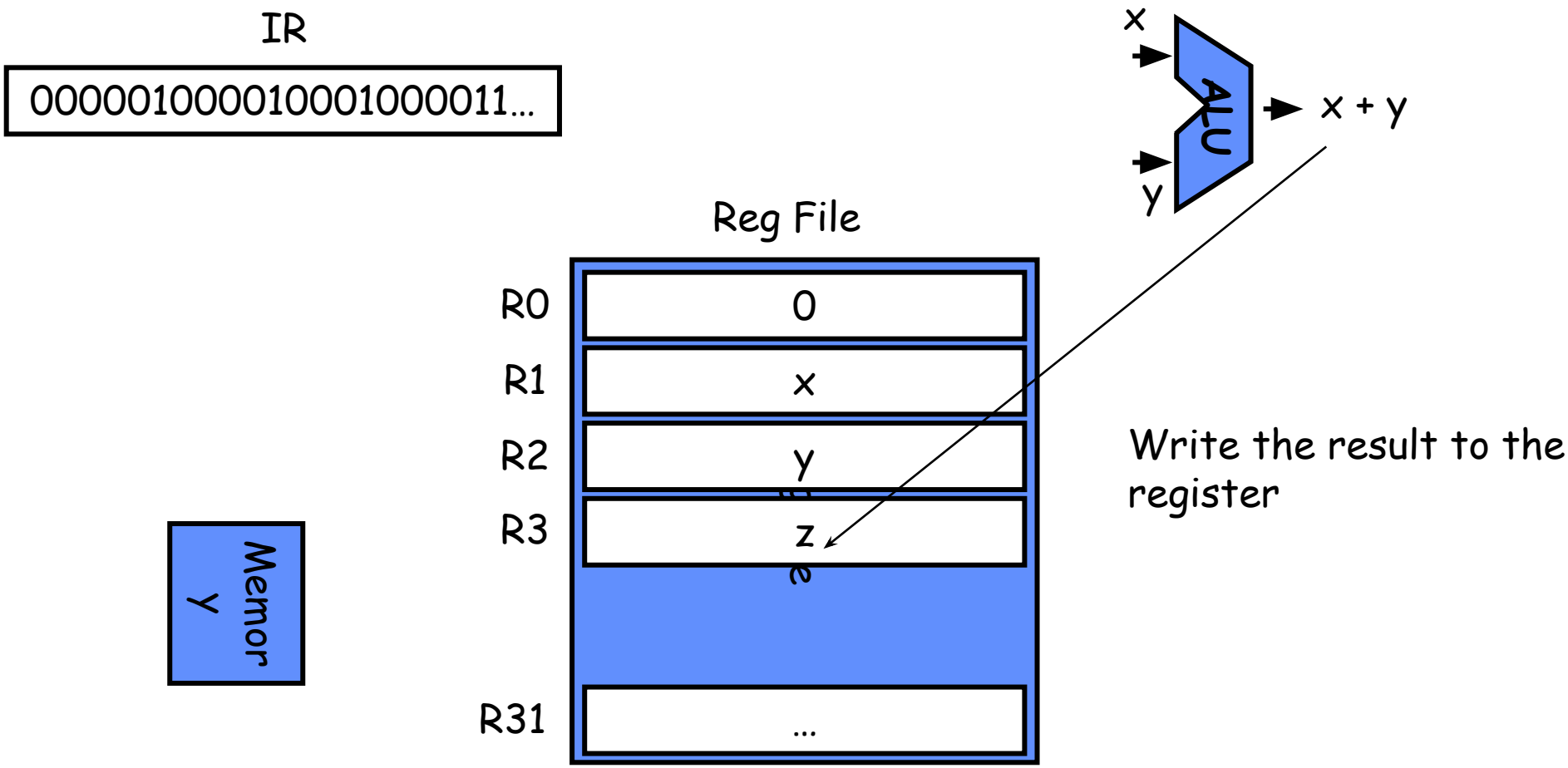
Execute the operation





Writeback (WB)

- **ADD R3, R1, R2 => R3 = R1 + R2;**

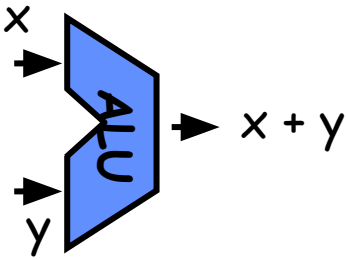


Writeback (WB)

- **ADD R3, R1, R2 => R3 = R1 + R2;**

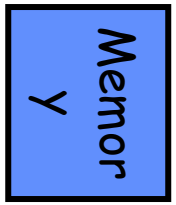
IR

00000100001000100001 1...



Reg File

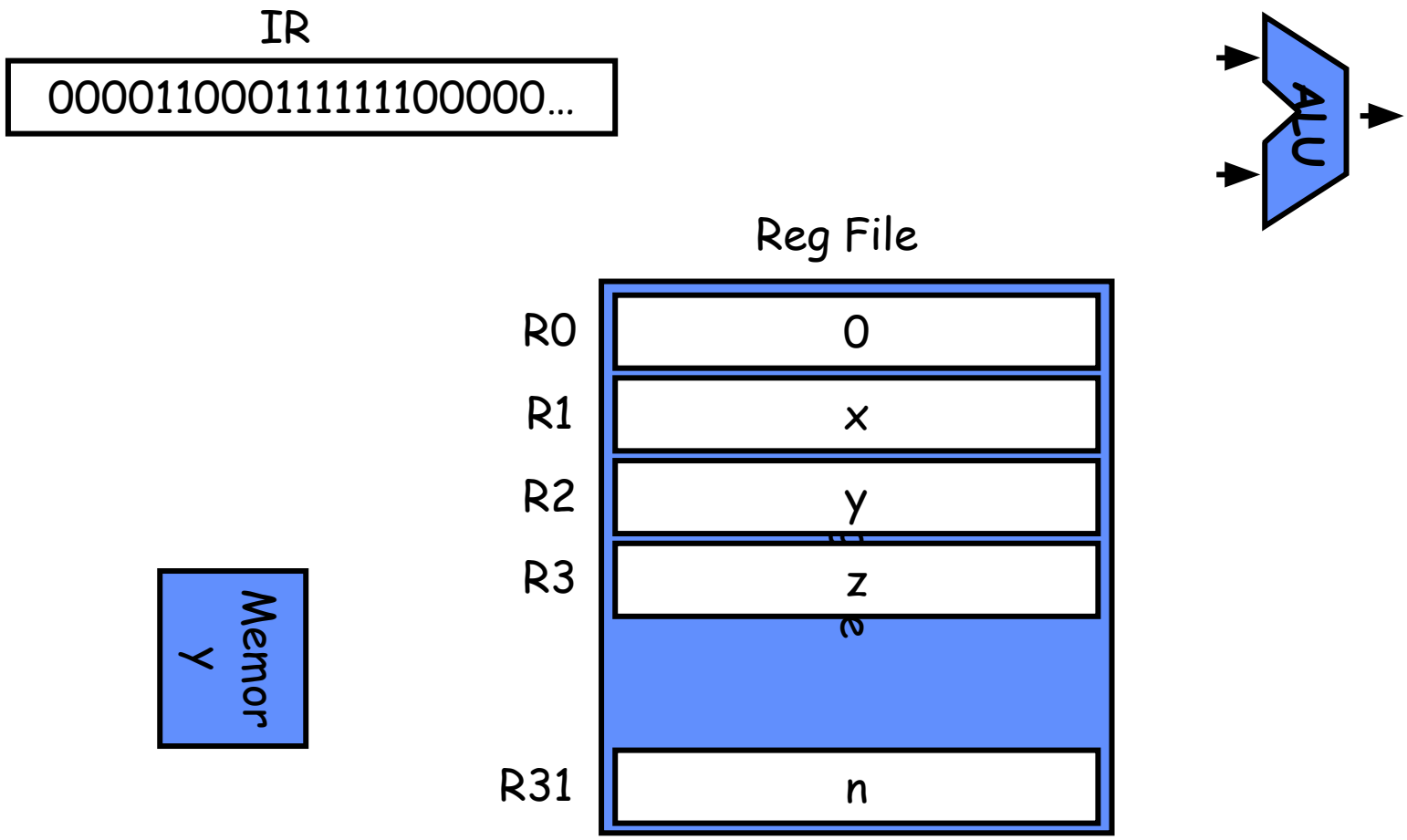
R0	0
R1	x
R2	y
R3	x + y
R31	...



Write the result to the register

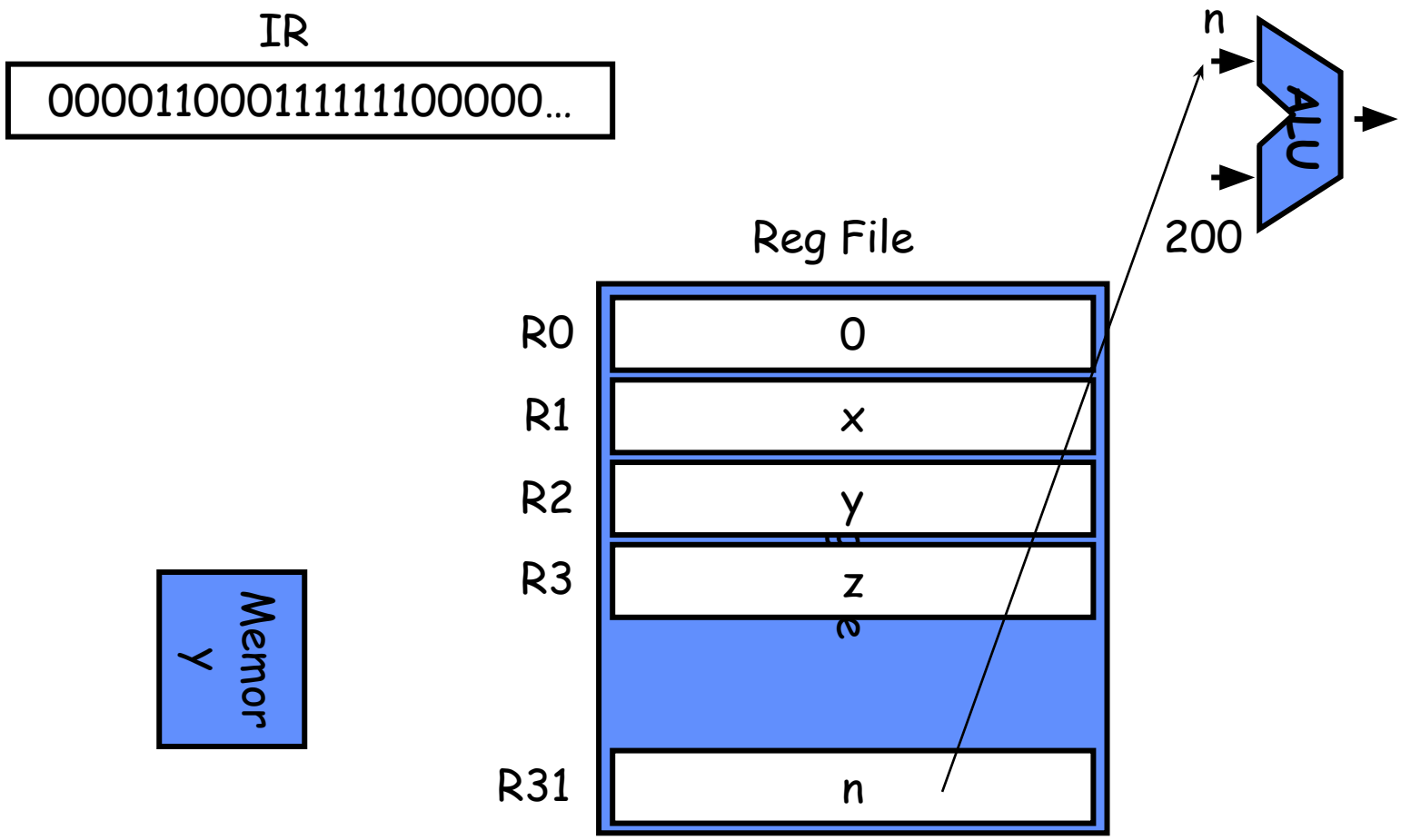
Another example: Load

- **LW R3, R31, 200 => R3 = mem[R31 + 200];**



Before the Execution (EX) stage

- **LW R3, R31, 200 => R3 = mem[R31 + 200];**

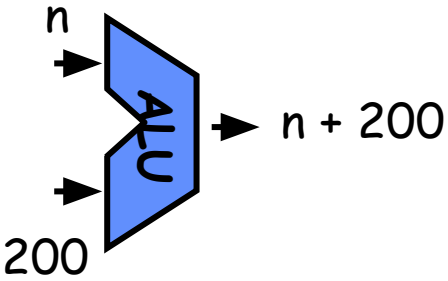


Execution (EX)

- **LW R3, R31, 200 => R3 = mem[R31 + 200];**

IR

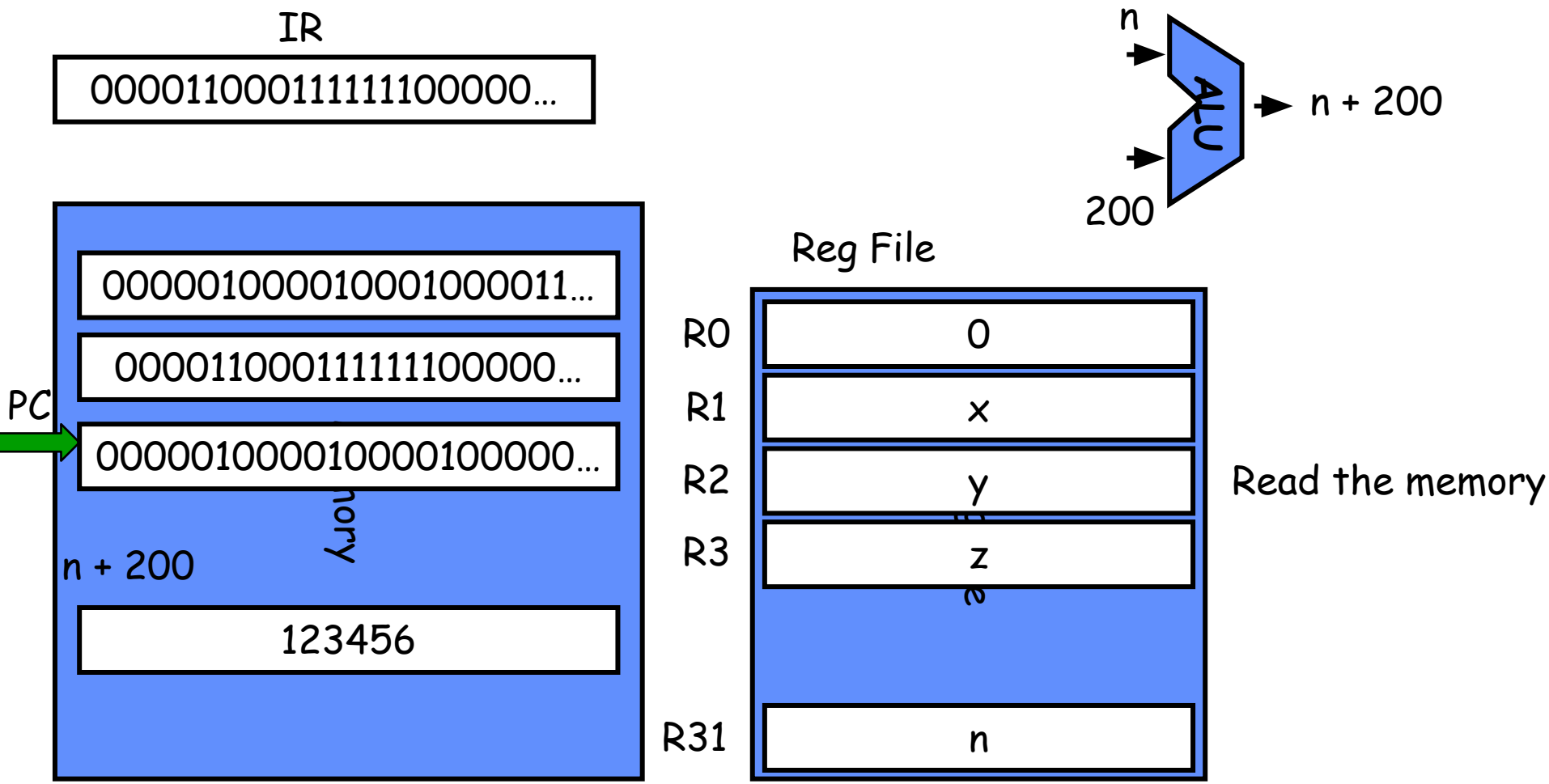
00001100011111100000…



Execute the operation

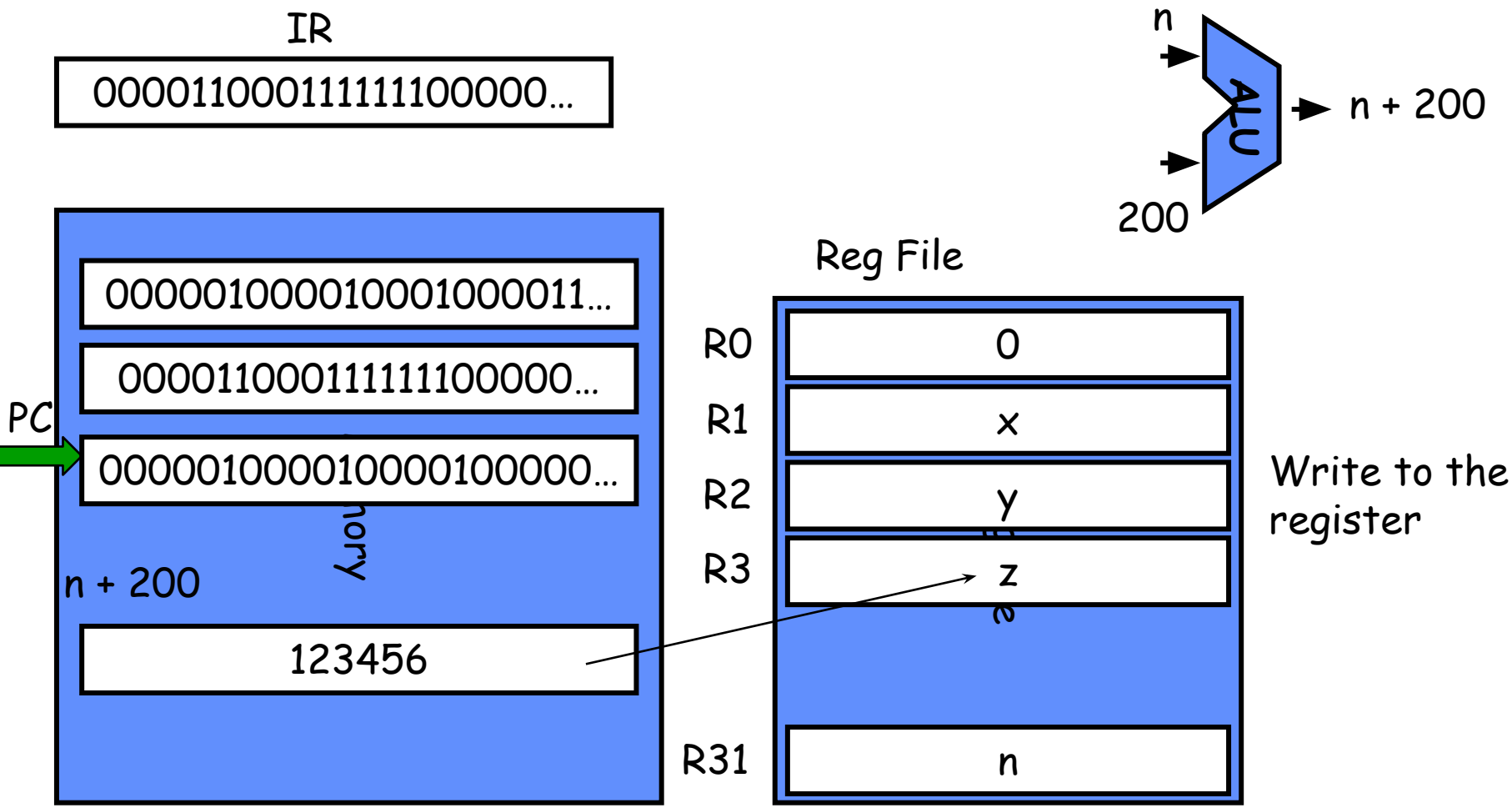
Memory Access (MEM)

- **LW R3, R31, 200 => R3 = mem[R31 + 200];**



Writeback (WB)

- **LW R3, R31, 200 => R3 = mem[R31 + 200];**

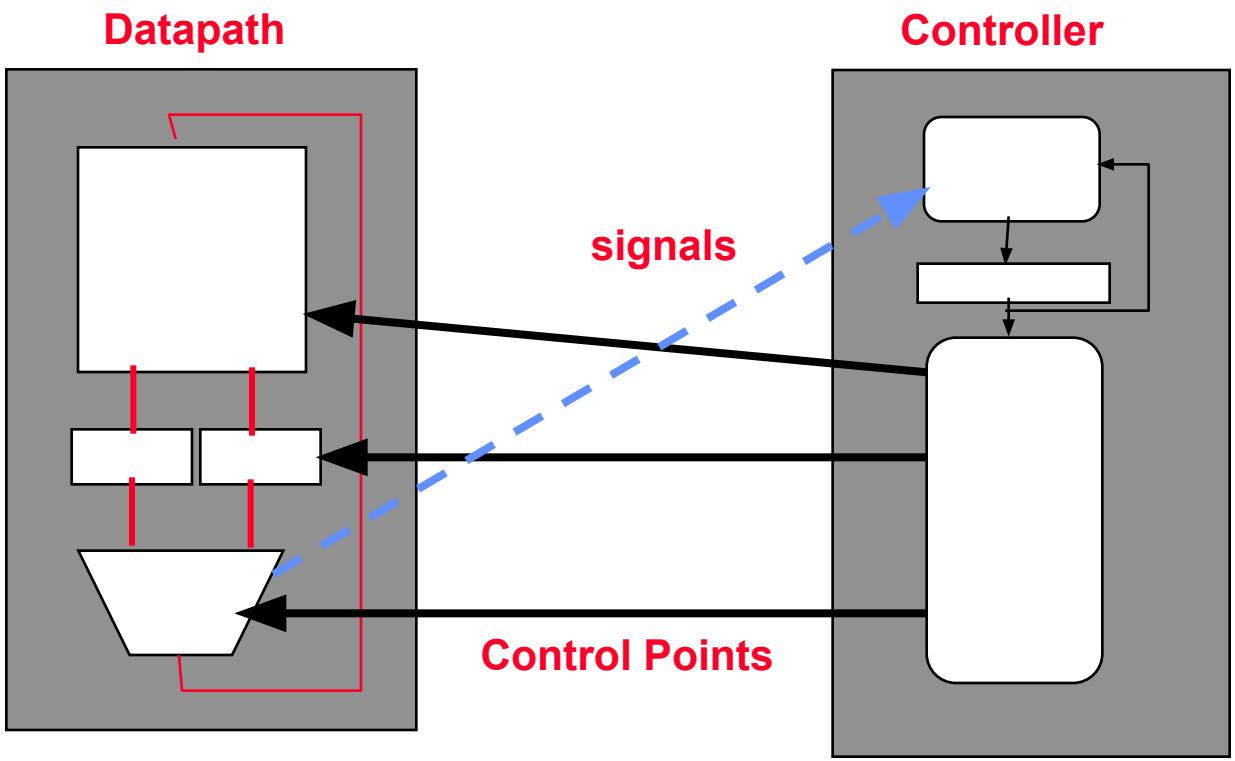


5-stage pipeline

- IF: mem, alu, load/store, branch
- ID: reg file, alu, load/store, branch
- EX: ALU, alu, load/store
- MEM: mem, load/store
- WB: reg file, alu, load

- Can we change the order of MEM and WB? Why?

Datapath vs Control



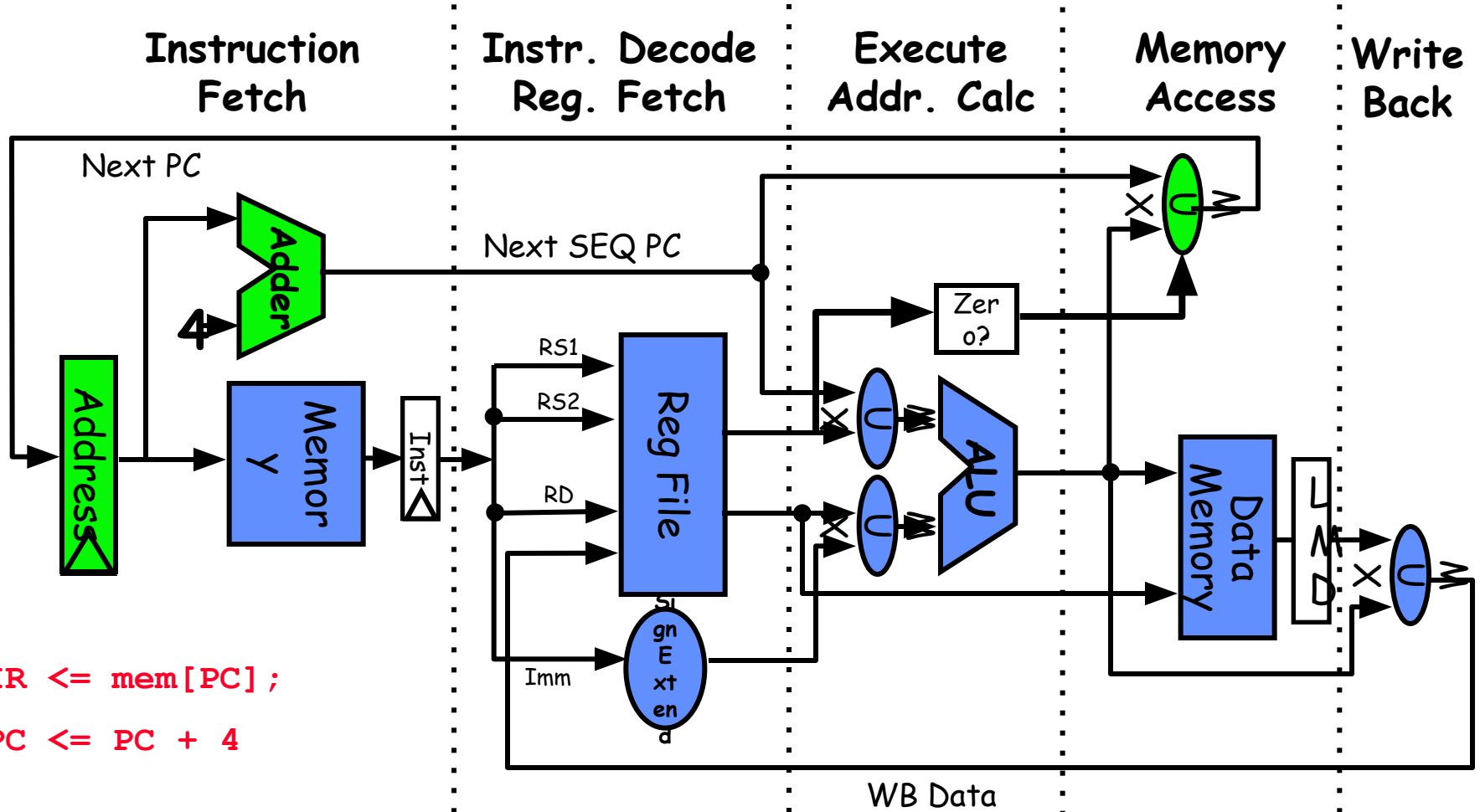
- Datapath: Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals
- Controller: State machine to orchestrate operation on the data path
  - Based on desired function and signals

Approaching an ISA

- Instruction Set Architecture
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- Meaning of each instruction is described by RTL (Register Transfer Language) on *architected registers* and memory
- Given technology constraints assemble adequate datapath
  - Architected storage mapped to actual storage
  - Function units to do all the required operations
  - Possible additional storage (eg. MAR, MBR, …)
  - Interconnect to move information among regs and FUs
- Map each instruction to sequence of RTLs
- Collate sequences into symbolic controller state transition diagram (STD)
- Lower symbolic STD to control points
- Implement controller

5 Steps of MIPS Datapath

Figure A.2, Page A-8




```
IR <= mem[PC];
PC <= PC + 4
```

$Reg[IR_{rd}] <= Reg[IR_{rs1}]\ op_{IRop}\ Reg[IR_{rs2}]$

Classic RISC Pipeline

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Source: http://en.wikipedia.org/wiki/Classic_RISC_pipeline

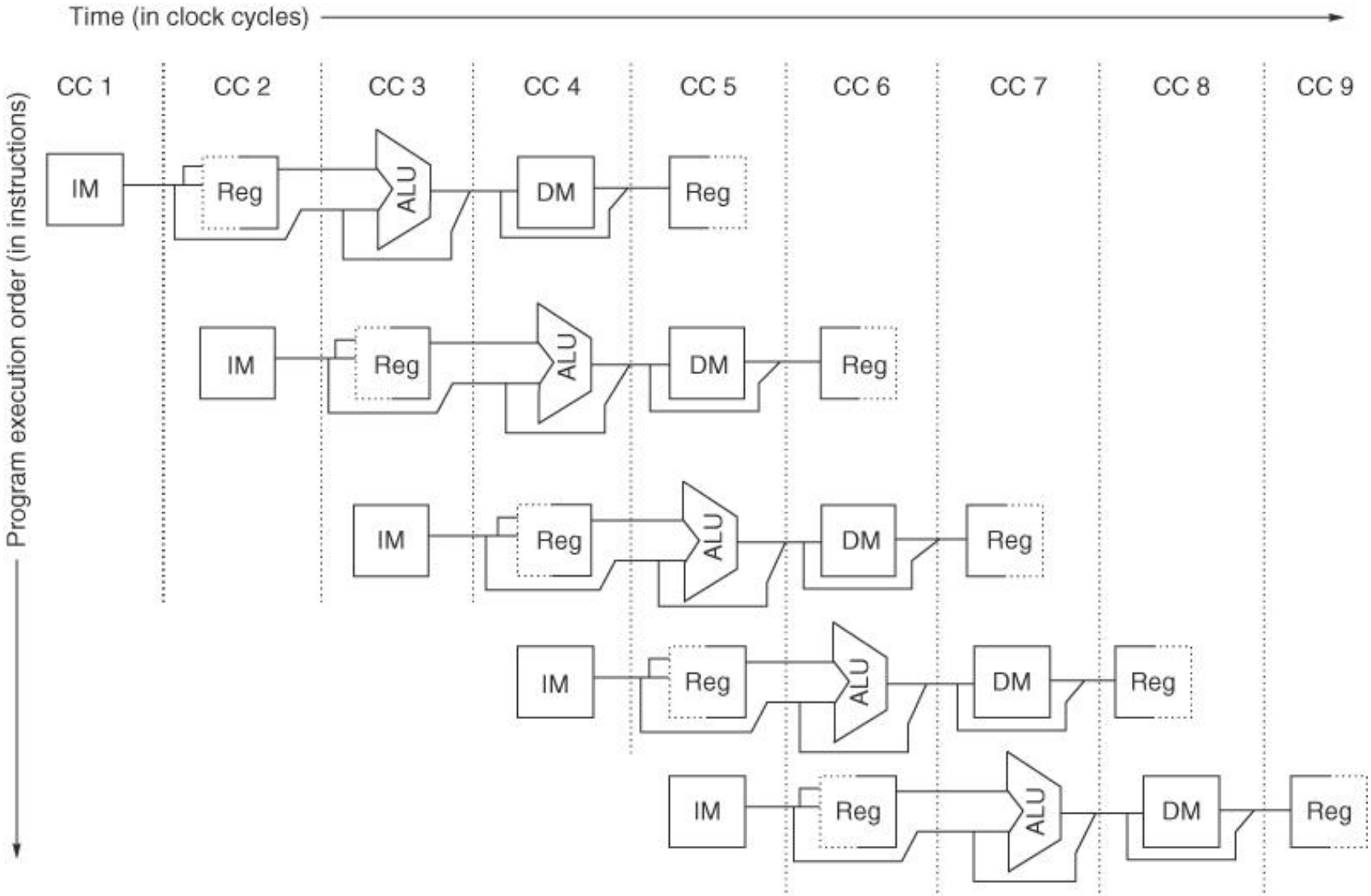
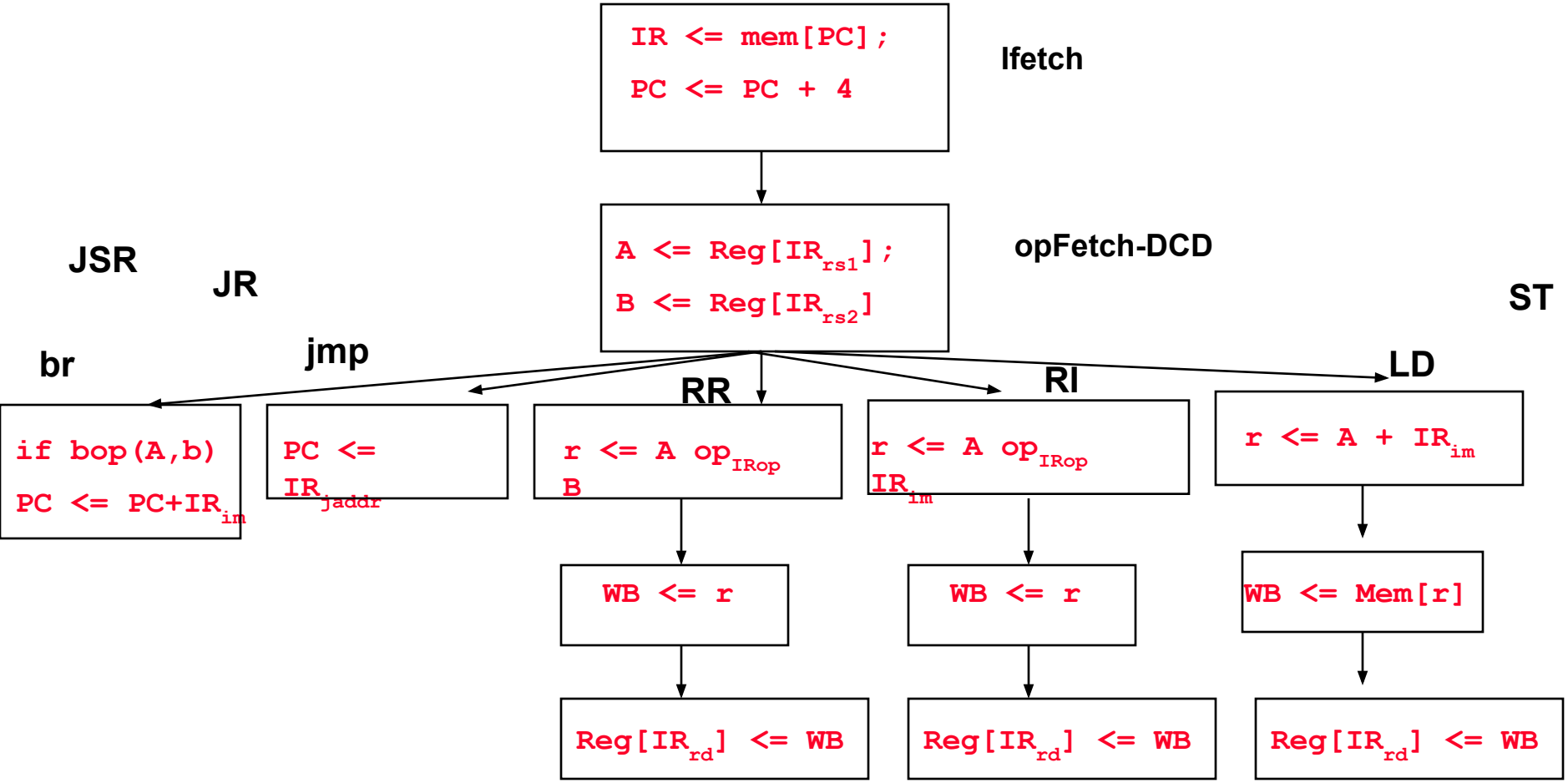


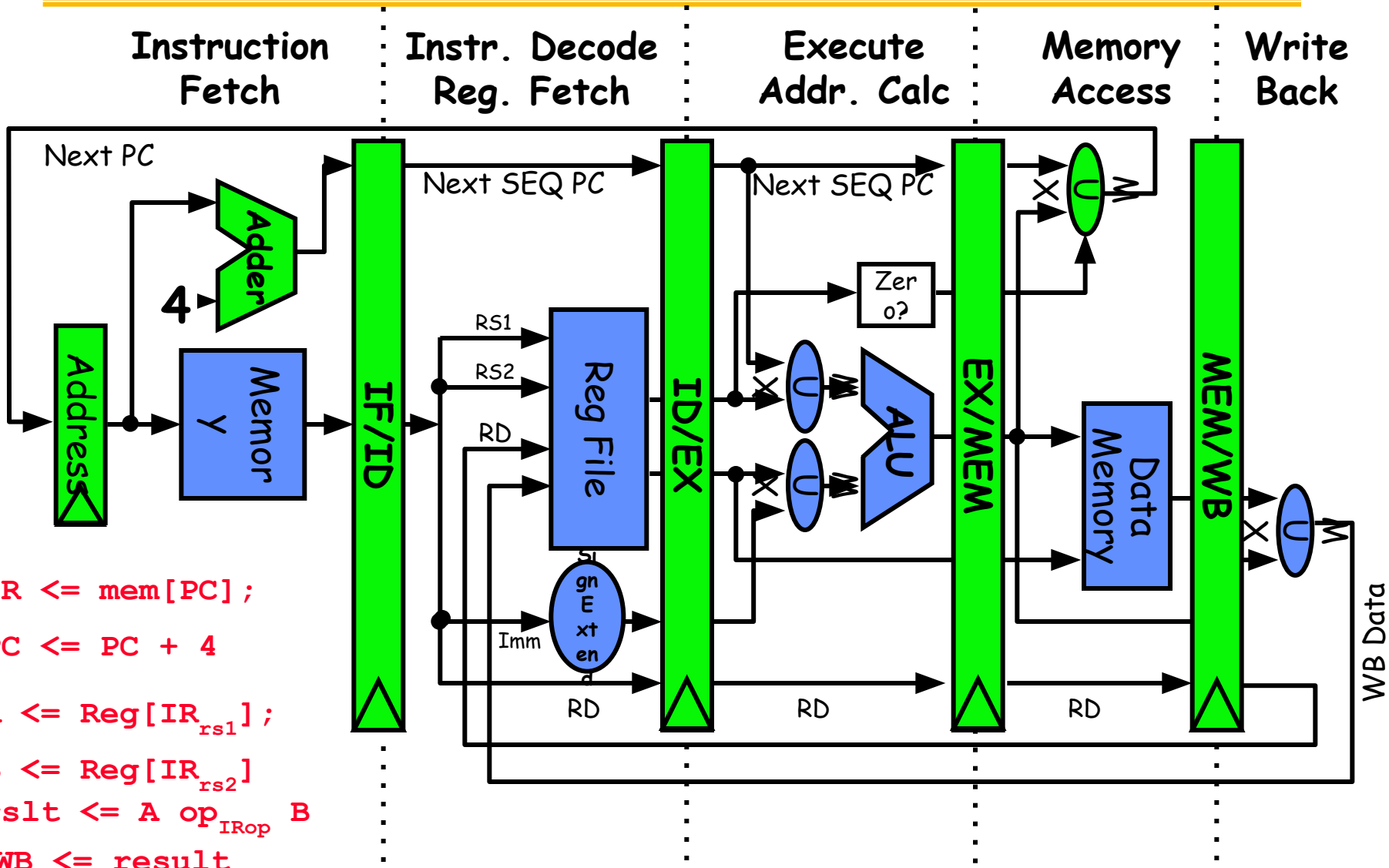
Figure C.2 The pipeline can be thought of as a series of data paths shifted in time. This shows the overlap among the parts of the data path, with clock cycle 5 (CC 5) showing the steady-state situation. Because the register file is used as a source in the ID stage and as a destination in the WB stage, it appears twice. We show that it is read in one part of the stage and written in another by using a solid line, on the right or left, respectively, and a dashed line on the other side. The abbreviation IM is used for instruction memory, DM for data memory, and CC for clock cycle.

Inst. Set Processor Controller



5 Steps of MIPS Datapath

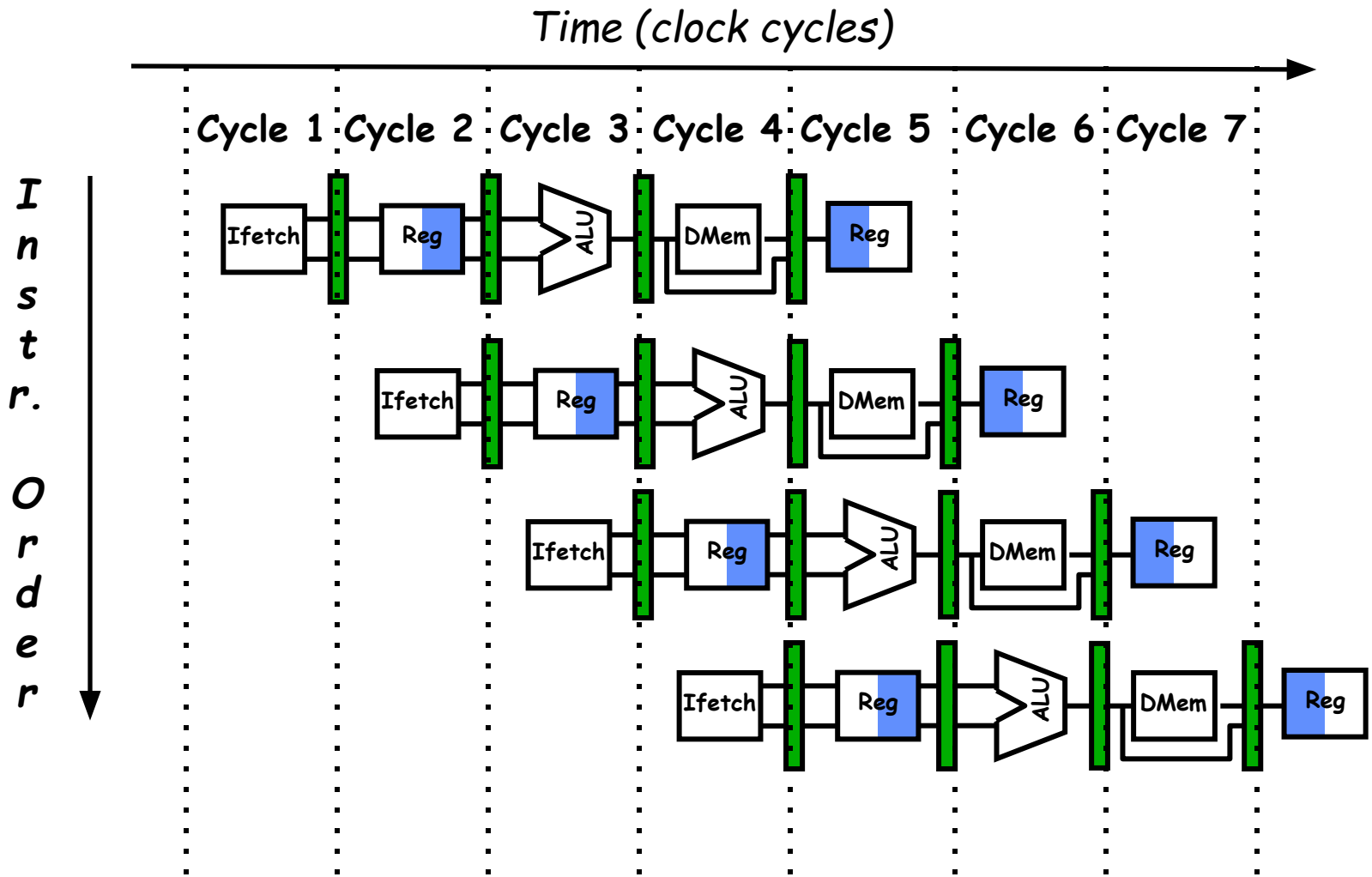
Figure A.3, Page A-9

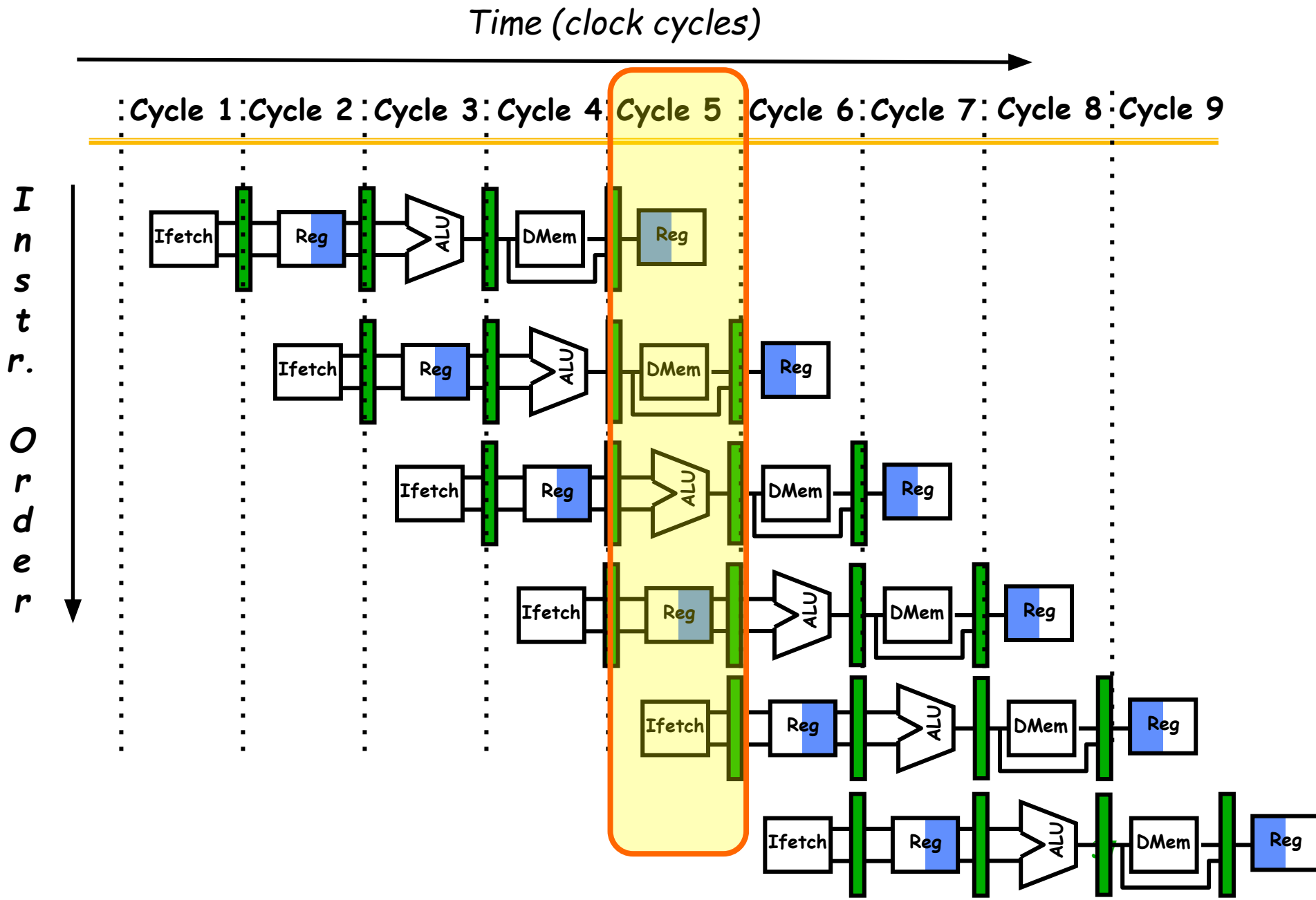


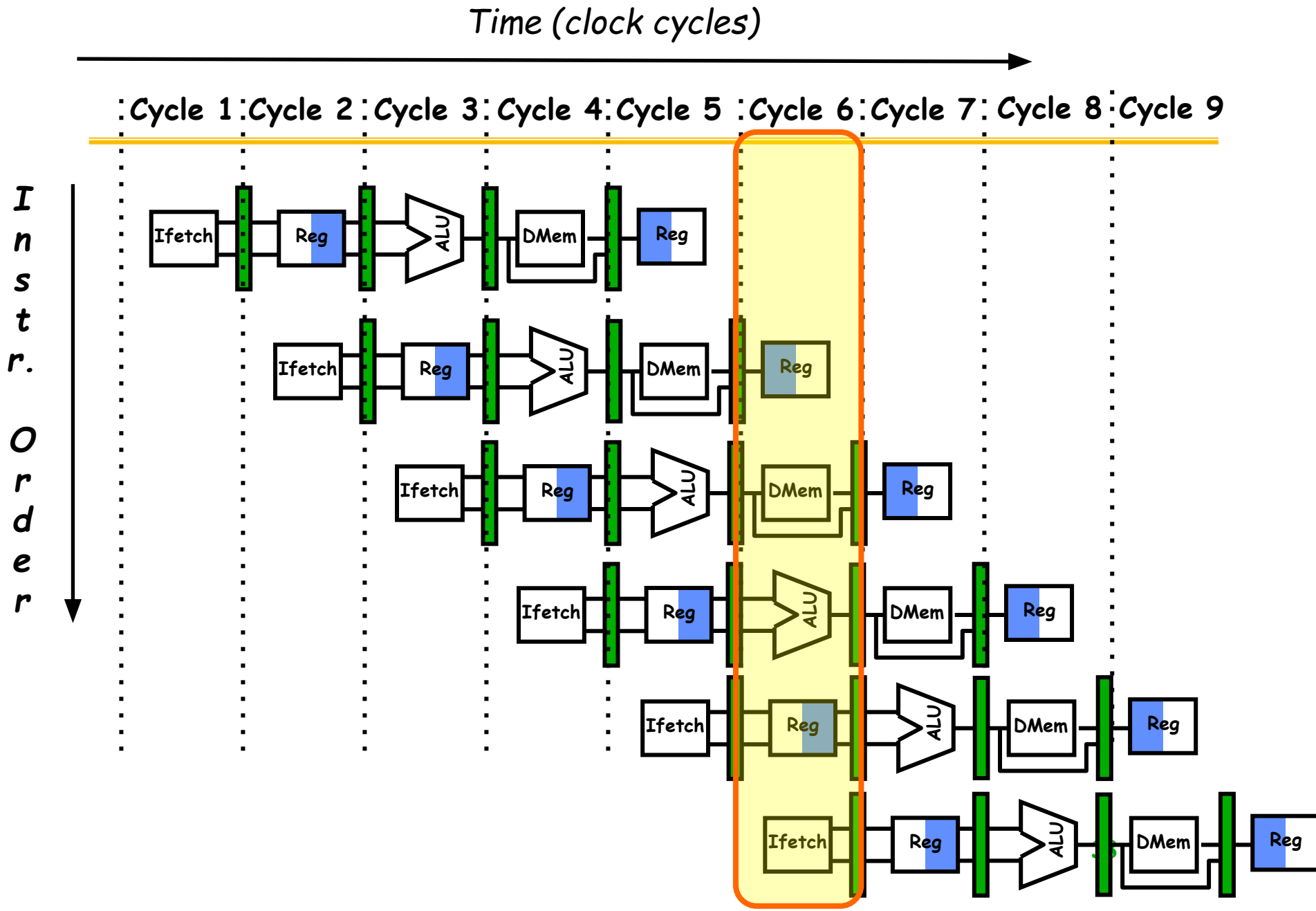

```
IR <= mem[PC];
PC <= PC + 4
A <= Reg[IR_rs1];
B <= Reg[IR_rs2]
rslt <= A op_IRop B
WB <= result
Reg[IR_rd] <= WB
```

Visualizing Pipelining

Figure A.2, Page A-8




Time (clock cycles)
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Cycle 8
Cycle 9
Instr. Order
Ifetch
Reg
ALU
DMem
Reg

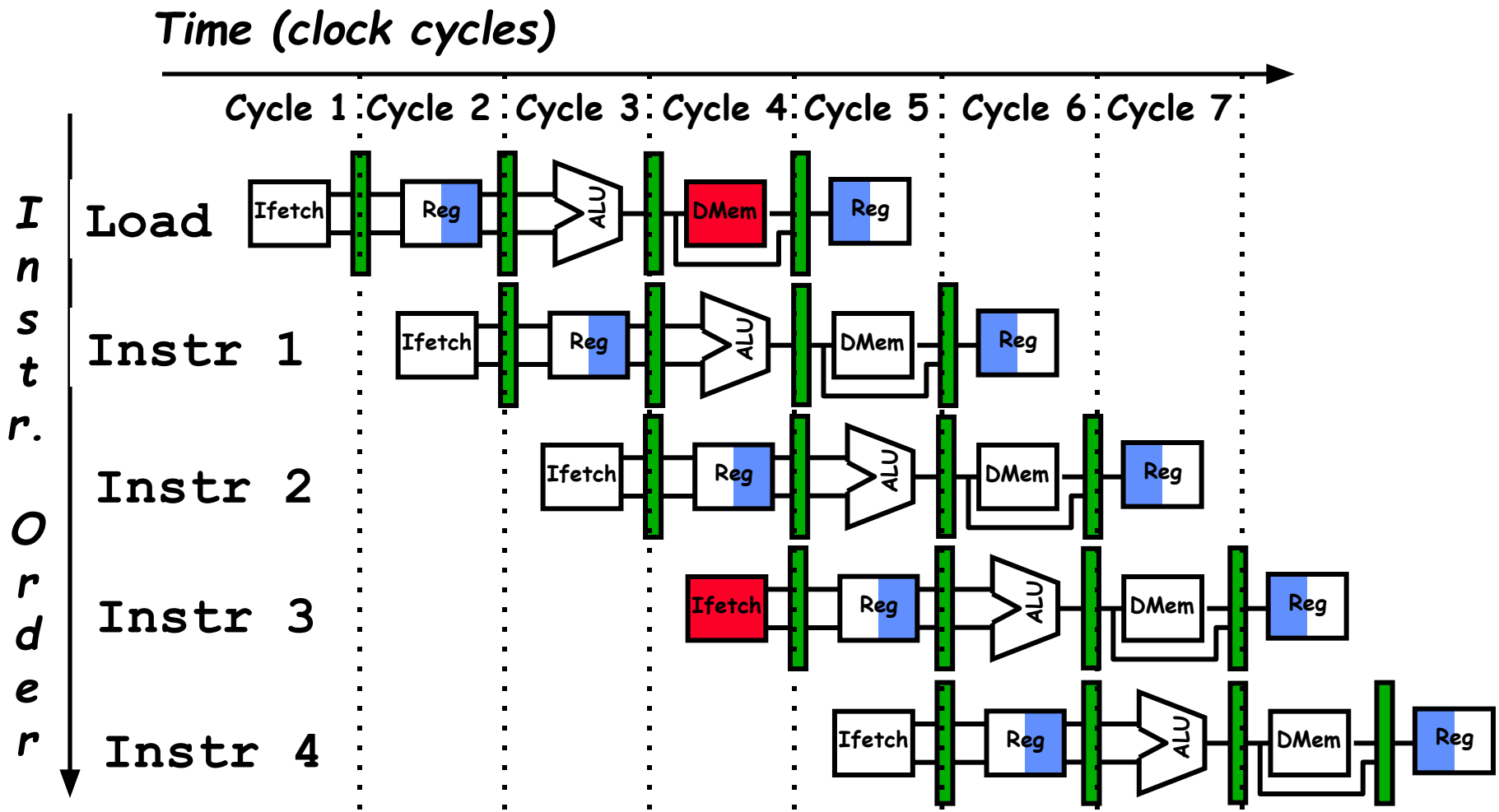

Time (clock cycles)
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Cycle 8
Cycle 9
Instr. Order
Ifetch
Reg
ALU
DMem
Reg

Pipelining is not quite that easy!

- **Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle**
  - **Structural hazards: HW cannot support this combination of instructions**
  - **Data hazards: Instruction depends on result of prior instruction still in the pipeline**
  - **Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).**

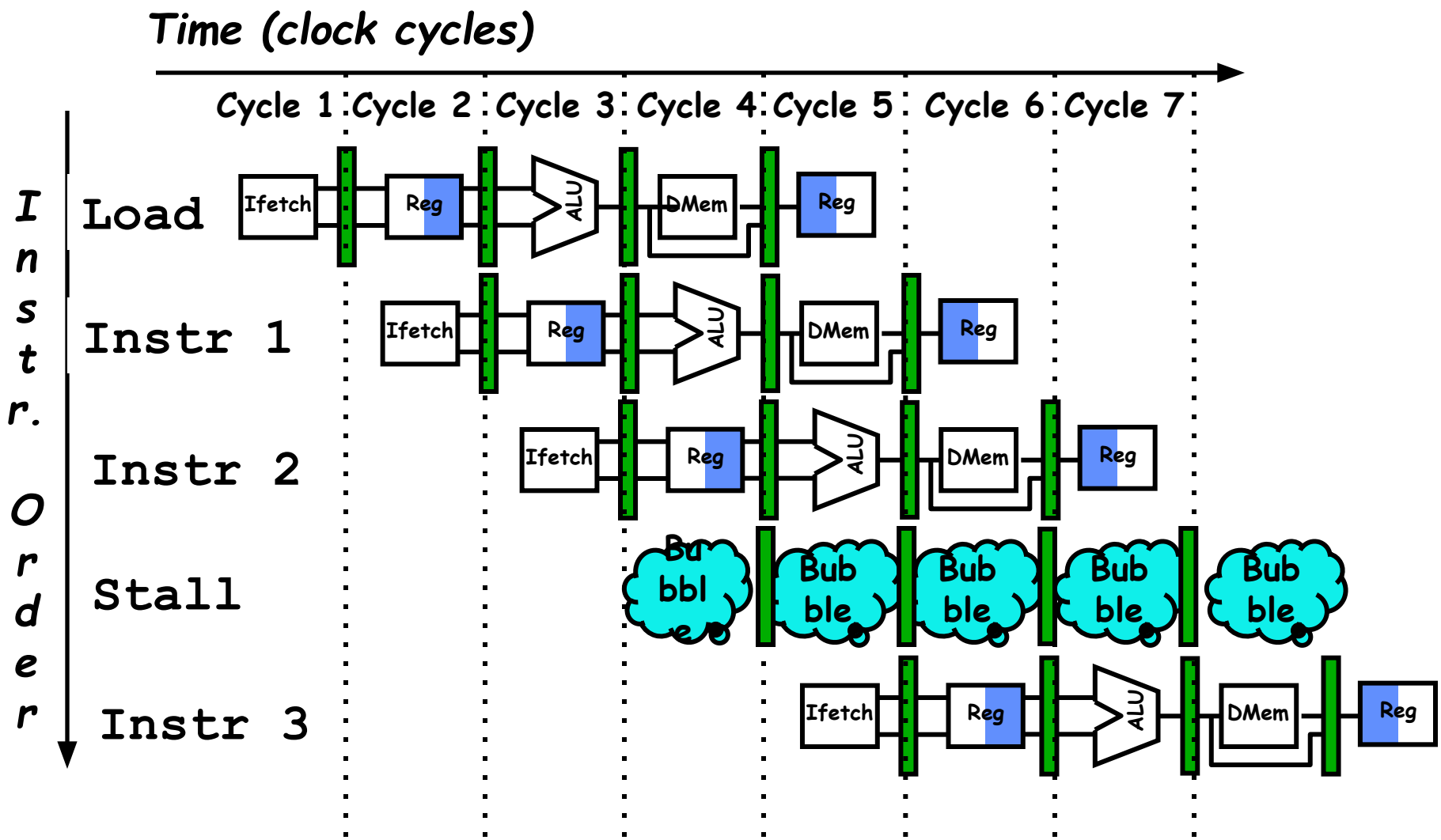
One Memory Port/Structural Hazards

Figure A.4, Page A-14



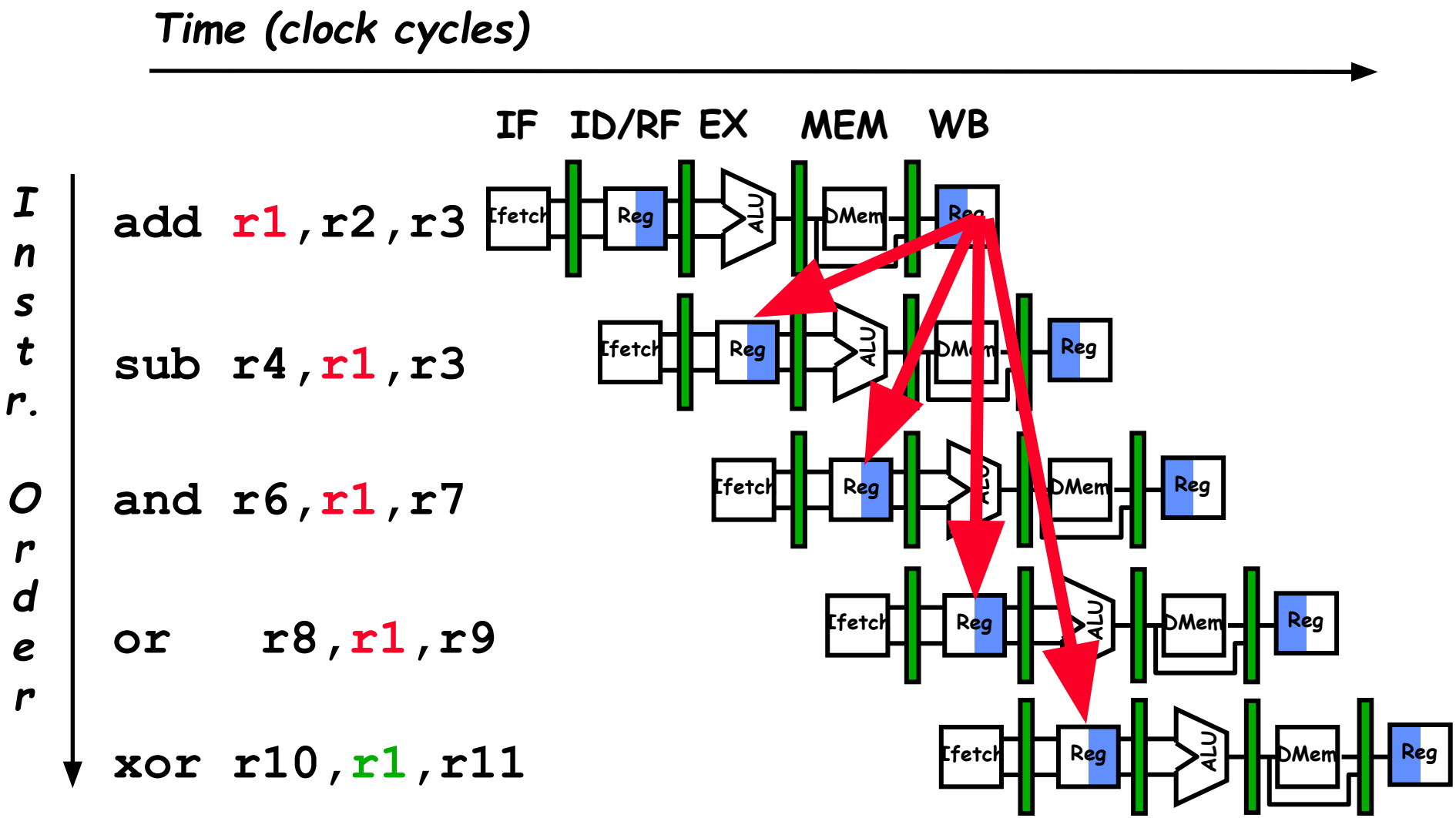
One Memory Port/Structural Hazards

(Similar to Figure A.5, Page A-15)



Data Hazard on R1

Figure A.6, Page A-17



Three Generic Data Hazards

- **Read After Write (RAW)**
  $Instr_J$ tries to read operand before $Instr_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- **Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.**

Three Generic Data Hazards

- **Write After Read (WAR)**
  $Instr_J$ writes operand *before* $Instr_I$ reads it

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- **Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1".**

- **Can it happen in MIPS 5 stage pipeline?**
  - **All instructions take 5 stages, and**
  - **Reads are always in stage 2, and**
  - **Writes are always in stage 5**

Three Generic Data Hazards

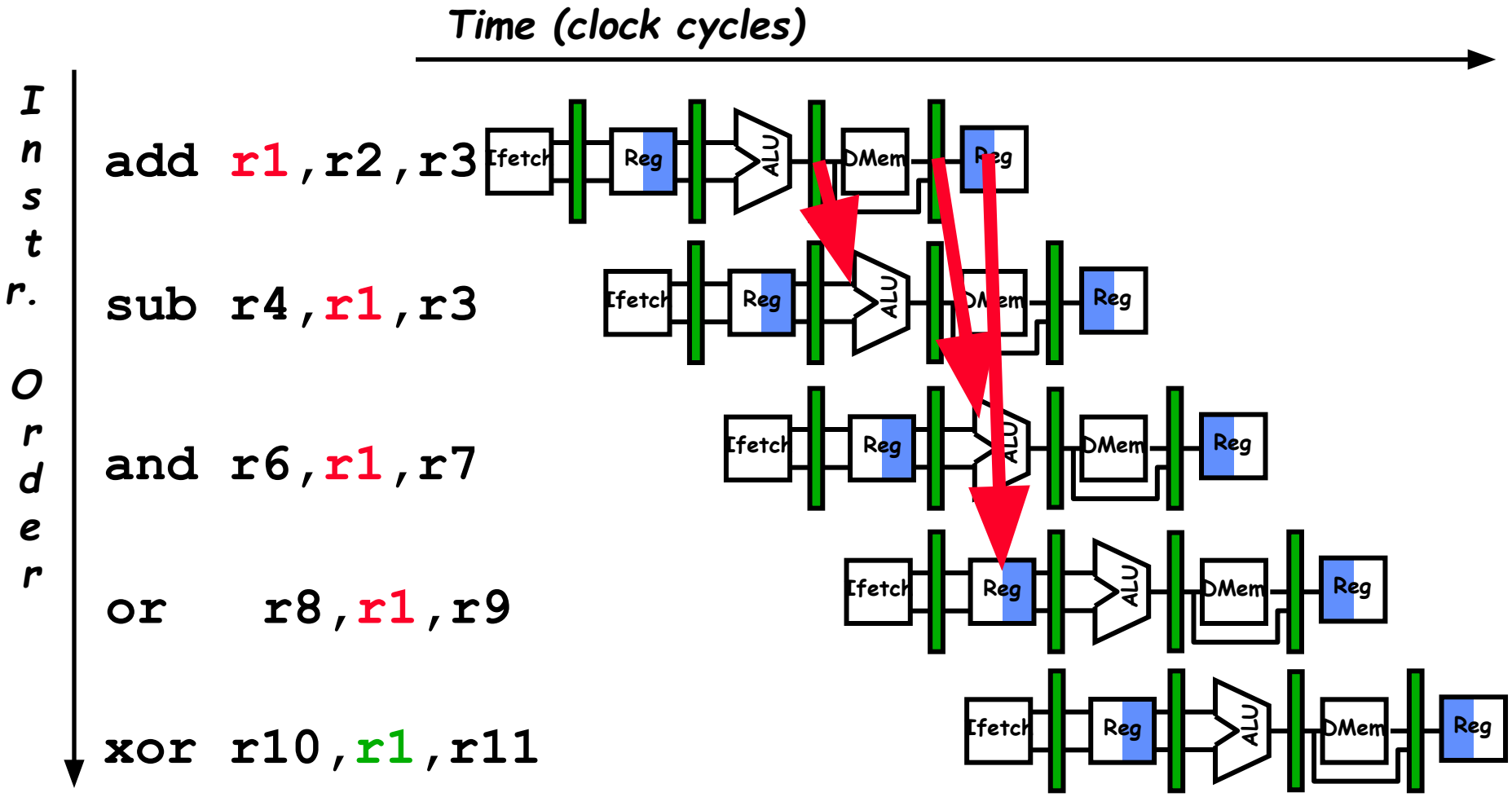
- **Write After Write (WAW)**
  $Instr_J$ writes operand *before* $Instr_I$ writes it.

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- **Called an "output dependence" by compiler writers**
  This also results from the reuse of name "r1".
- **Can't happen in MIPS 5 stage pipeline because:**
  - **All instructions take 5 stages, and**
  - **Writes are always in stage 5**
- **Will see WAR and WAW in more complicated pipes**

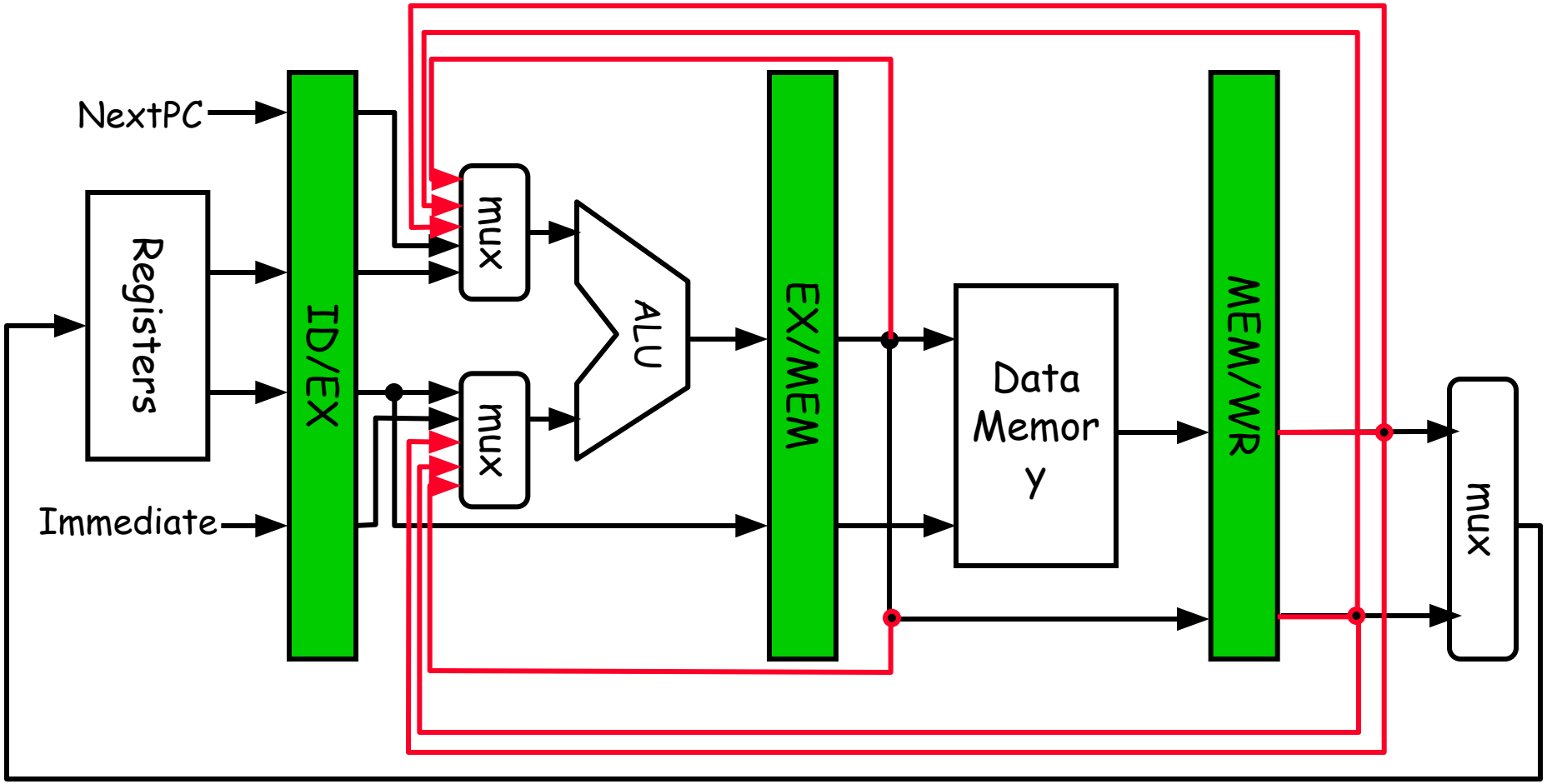
Forwarding to Avoid Data Hazard

Figure A.7, Page A-19



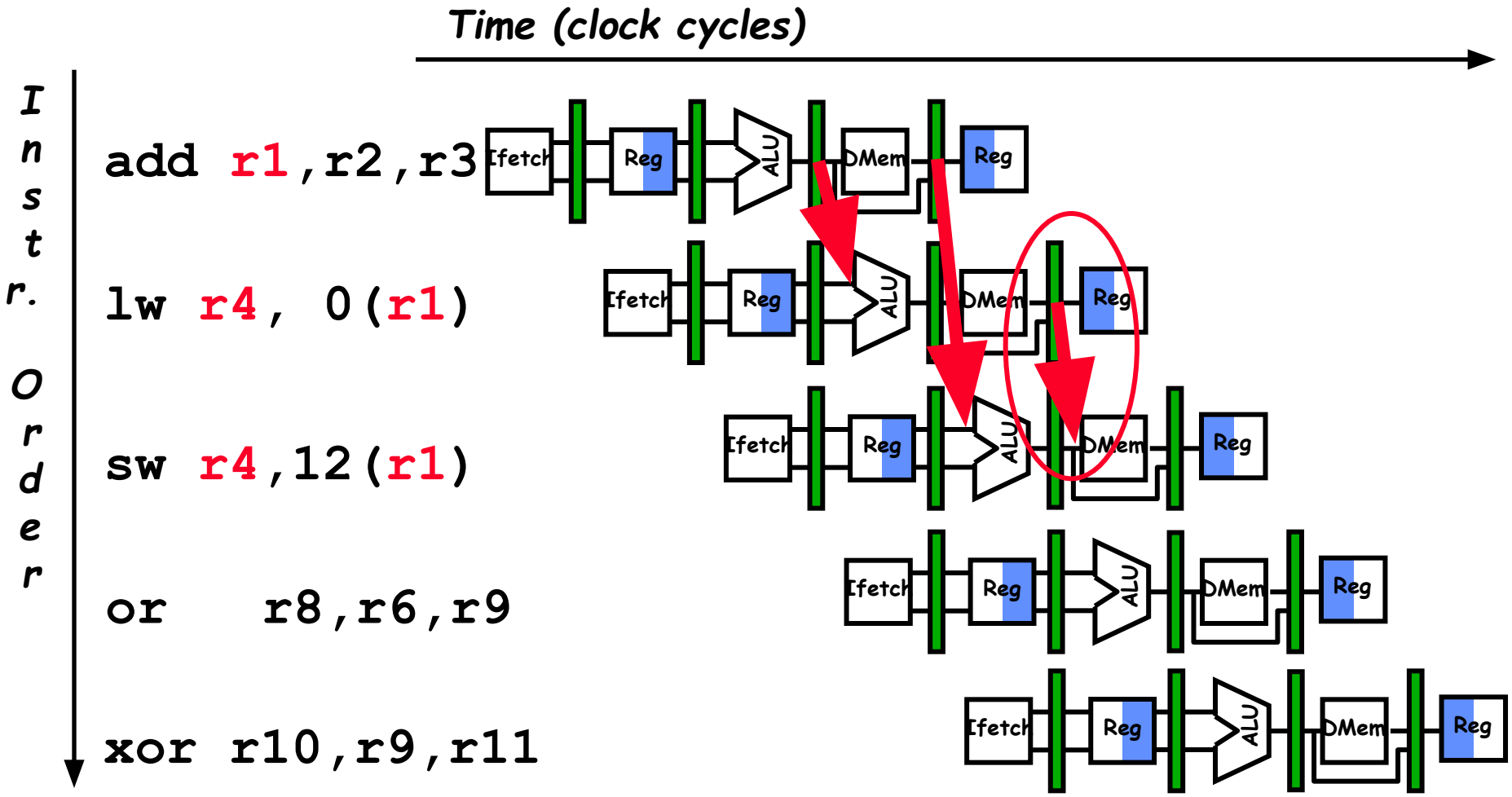
HW Change for Forwarding

Figure A.23, Page A-37



Forwarding to Avoid LW-SW Data Hazard

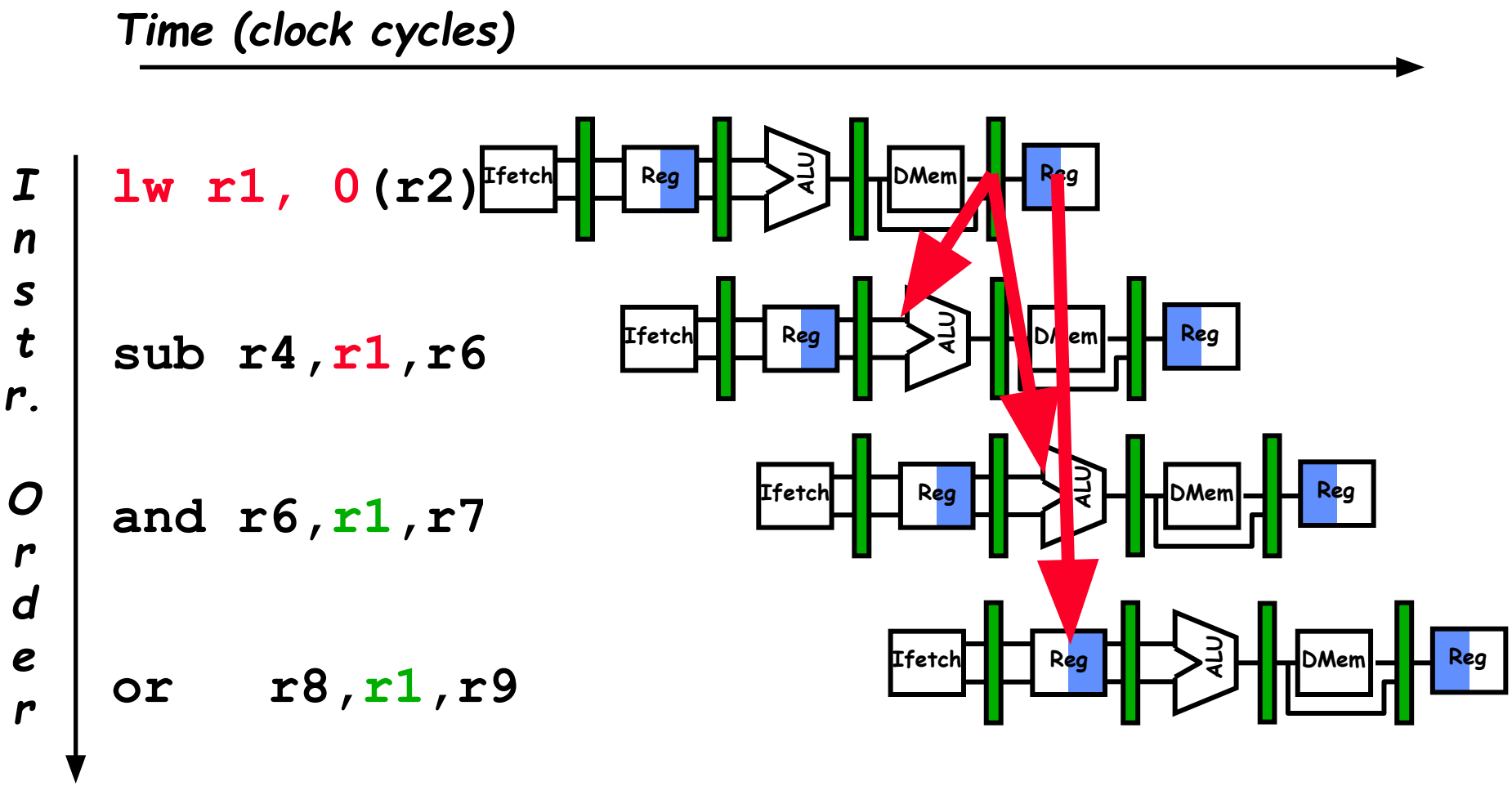
Figure A.8, Page A-20



Any hazard that cannot be avoided with forwarding?

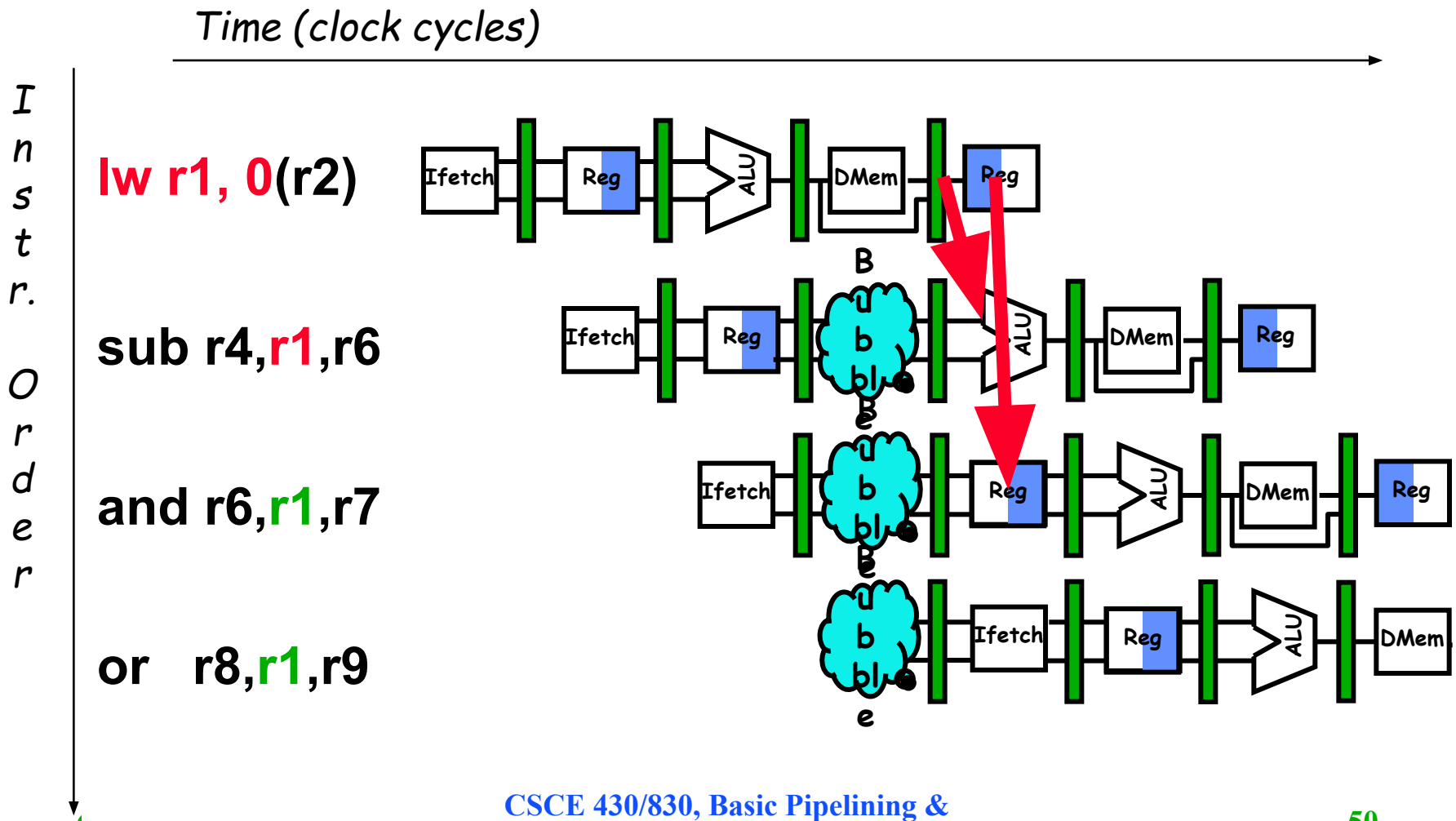
Data Hazard Even with Forwarding

Figure A.9, Page A-21



Data Hazard Even with Forwarding

(Similar to Figure A.10, Page A-21)



Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

d = e – f;

assuming a, b, c, d ,e, and f in memory.

Slow code:

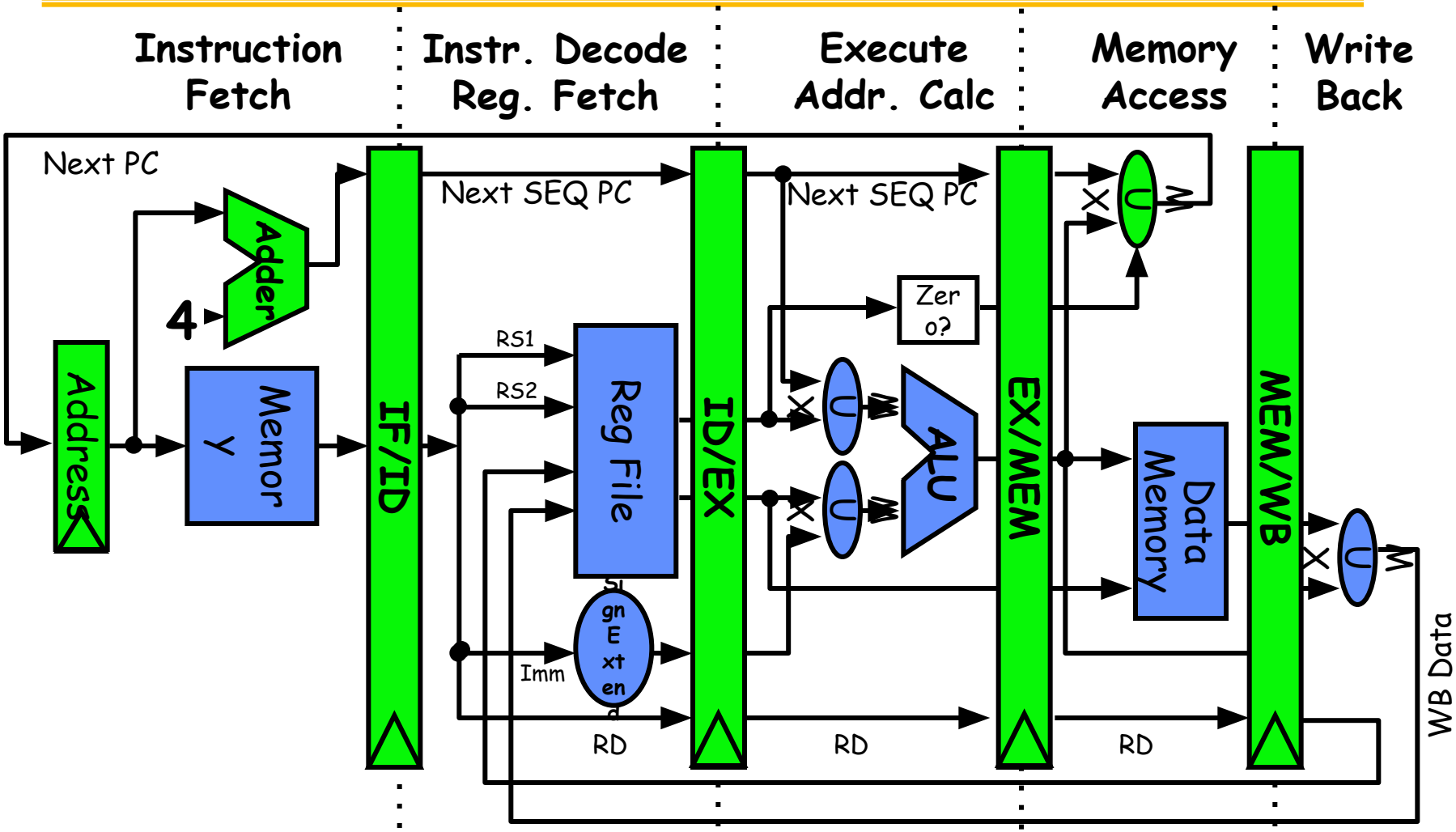
```
LW  Rb,b
LW  Rc,c
ADD     Ra,Rb,Rc
SW      a,Ra
LW  Re,e
LW  Rf,f
SUB     Rd,Re,Rf
SW  d,Rd
```

Fast code:

```
LW  Rb,b
LW  Rc,c
LW  Re,e
ADD     Ra,Rb,Rc
LW  Rf,f
SW      a,Ra
SUB     Rd,Re,Rf
SW  d,Rd
```

Compiler optimizes for performance. Hardware checks for safety.

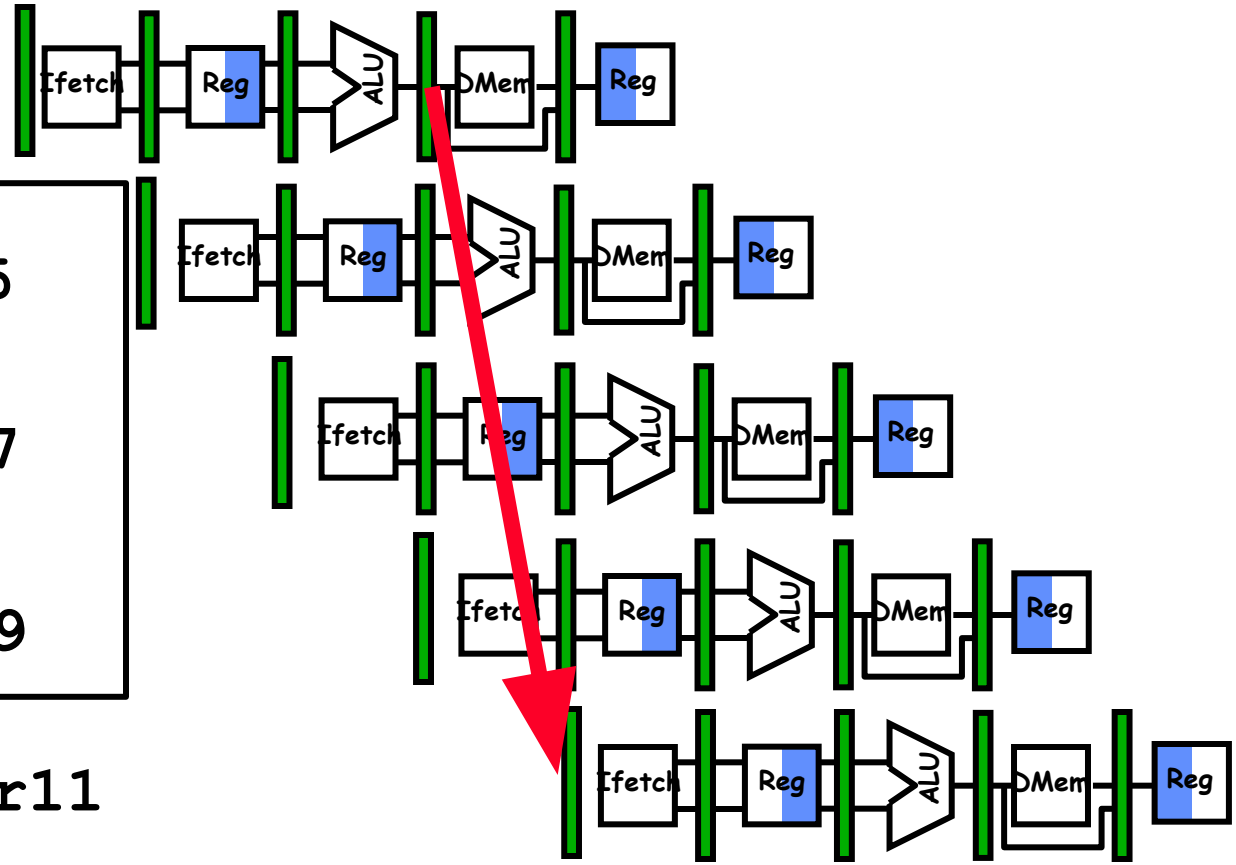
Control Hazard:



- Branch condition determined @ end of EX stage
- Target instruction address ready @ end of Mem stage

Control Hazard on Branches
Three Stage Stall

```
10: beq r1,r3,36
14: and r2,r3,r5
18: or  r6,r1,r7
22: add r8,r1,r9
36: xor r10,r1,r11
```



What do you do with the 3 instructions in between?

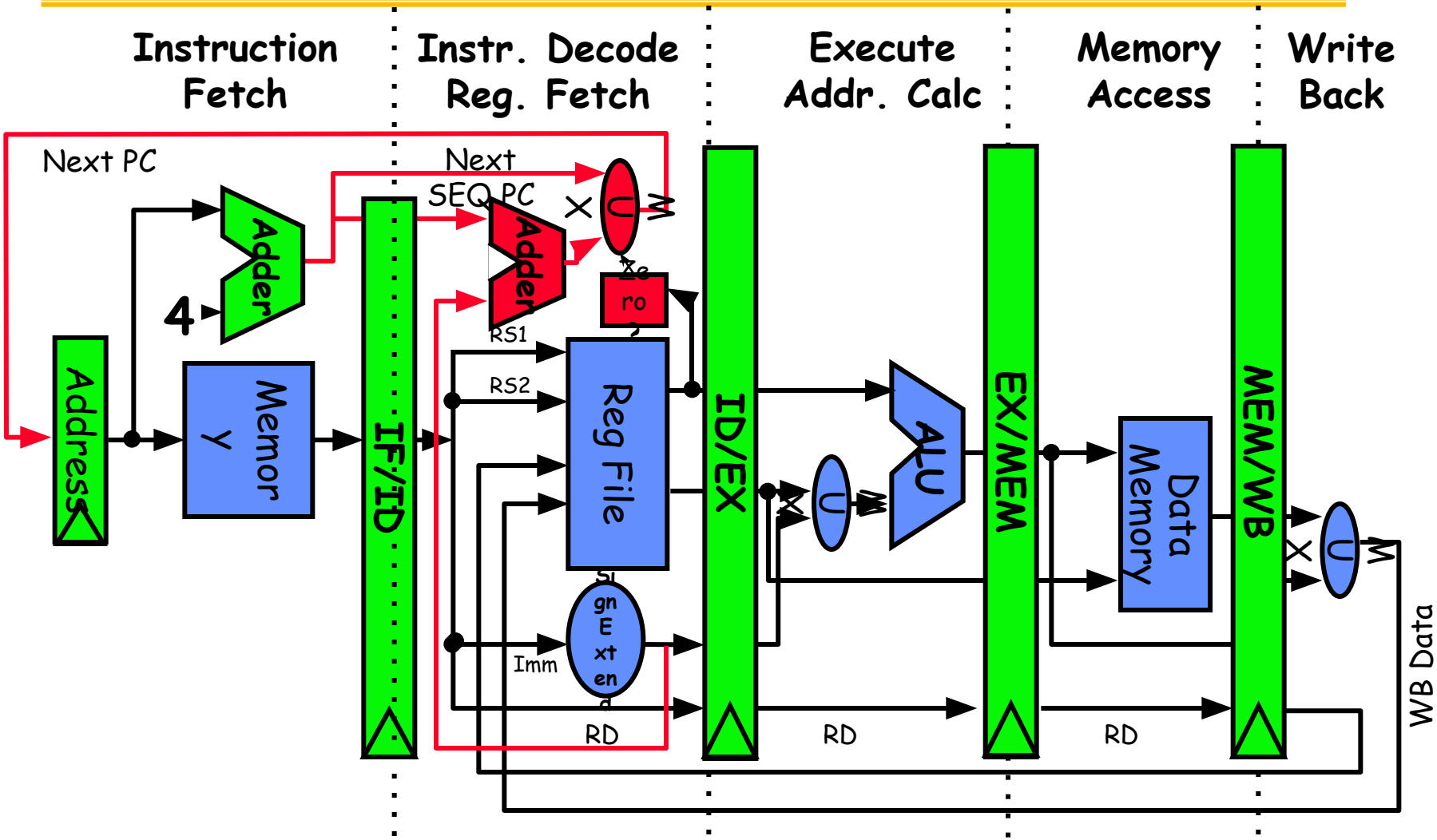
How do you do it?

Where is the "commit"?

Branch Stall Impact

- If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch outcome sooner, AND
  - Compute taken branch (target) address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Add an adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Two-part solution to the long stalls:



1. Determine the condition earlier @ ID
2. Calculate the target instruction address earlier @ ID

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- 53% MIPS branches taken on average
- <u>But haven't calculated branch target address in MIPS</u>
  - » MIPS still incurs 1 cycle branch penalty
  - » Other machines: branch target known before branch condition determination

Four Branch Hazard Alternatives

#4: Delayed Branch

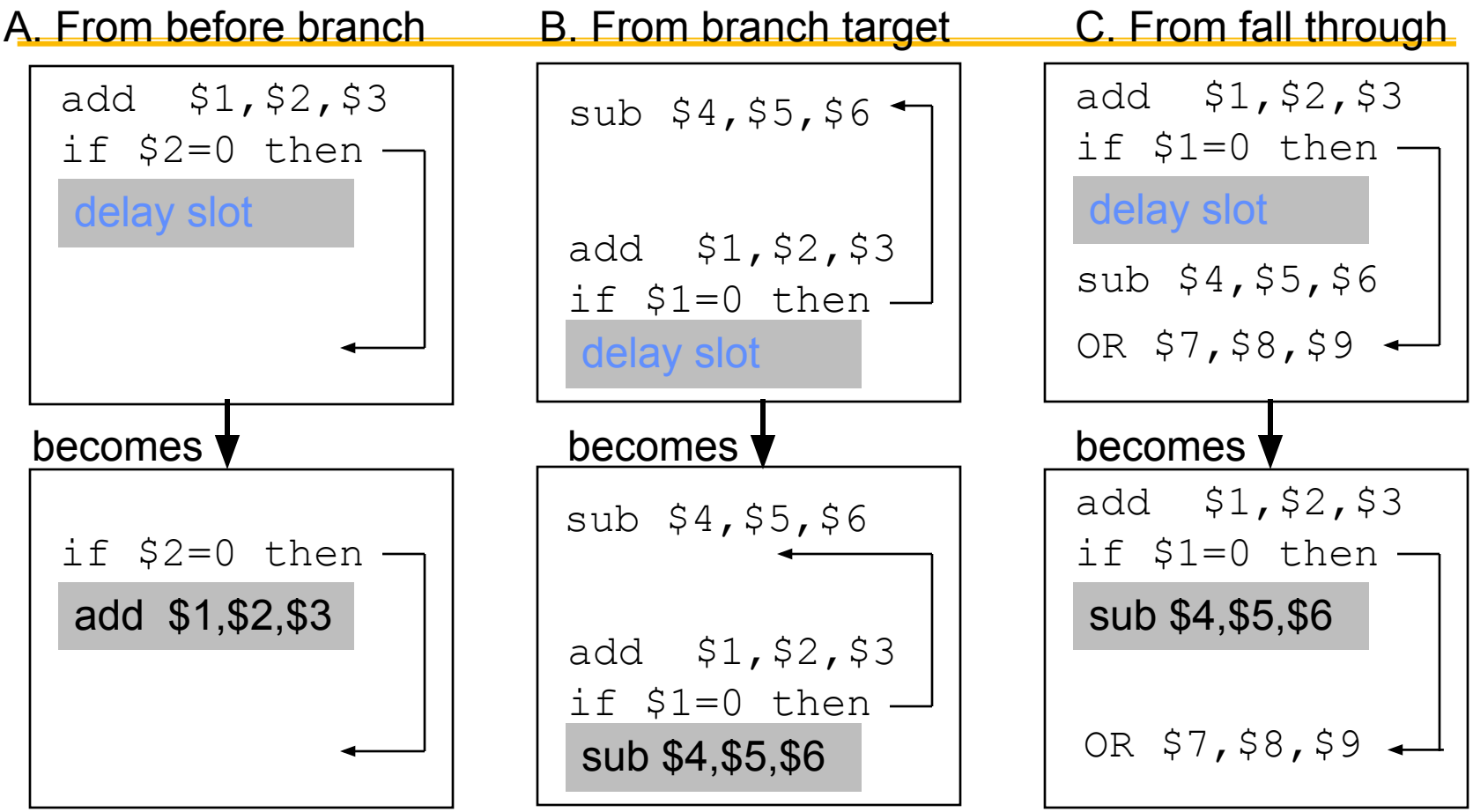
- Define branch to take place **AFTER** a following instruction

```
branch instruction
  sequential successor1
  sequential successor2
  ........
  sequential successorn
branch target if taken
```

Branch delay of length *n*

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Scheduling Branch Delay Slots (Fig A.14)



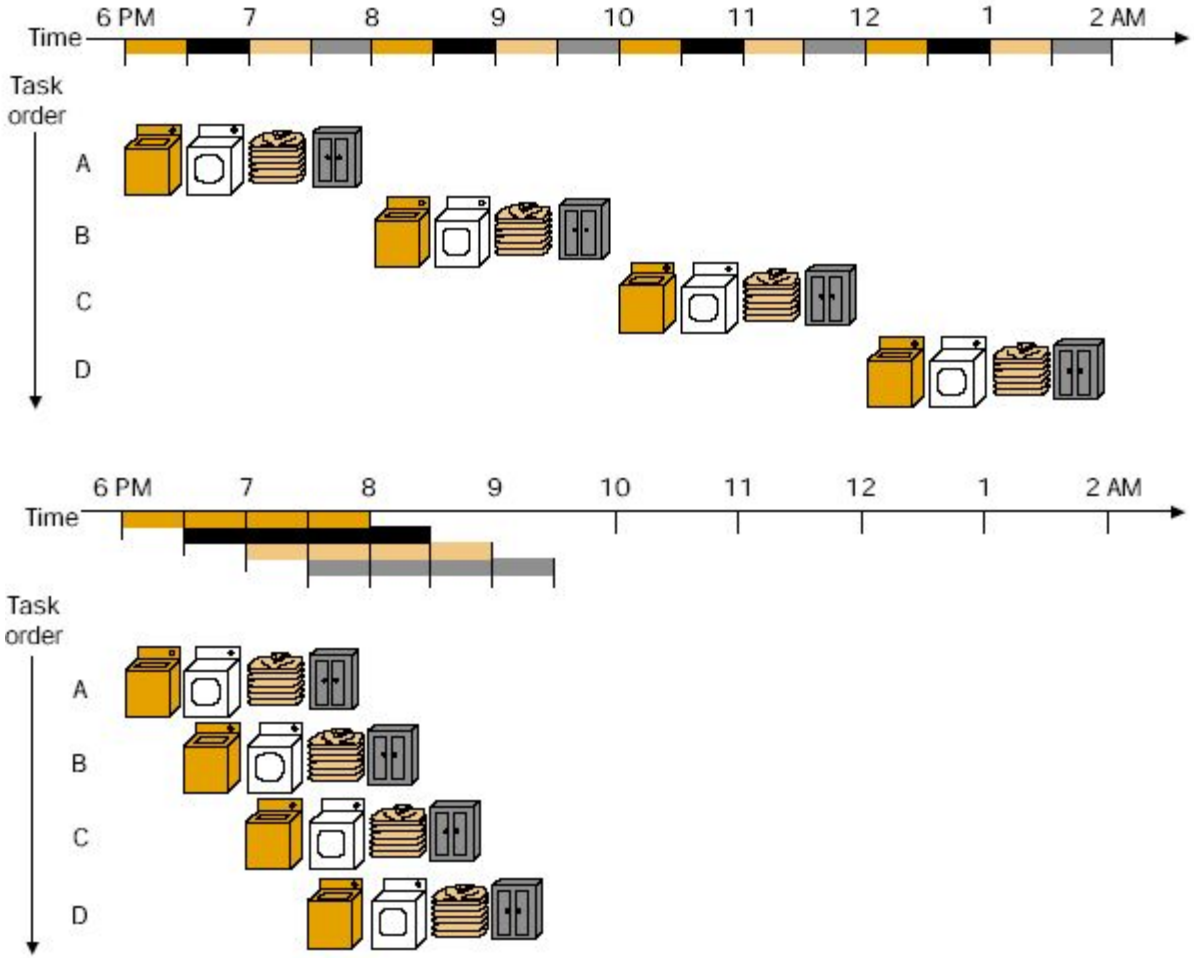
- **A is the best choice, fills delay slot**
- **In B, the `sub` instruction may need to be copied, increasing IC**
- **In B and C, must be okay to execute `sub` when branch fails**

Delayed Branch

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: As processors go to deeper pipelines and multiple issues, the branch delay grows and needs more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

Instruction-Level Parallelism

- Benefit of Pipelining (the laundry analogy)



Speed Up Equation for Pipelining

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

For simple RISC pipeline, Ideal CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

Example: Dual-port vs. Single-port

- Machine A: Dual ported memory ("Harvard Architecture")
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

$$\text{SpeedUp}_A = \text{Pipeline Depth}/(1 + 0) \times (\text{clock}_{unpipe}/\text{clock}_{pipe})$$
$$= \text{Pipeline Depth}$$
$$\text{SpeedUp}_B = \text{Pipeline Depth}/(1 + 0.4 \times 1) \times (\text{clock}_{unpipe}/(\text{clock}_{unpipe} / 1.05)$$
$$= (\text{Pipeline Depth}/1.4) \times 1.05$$
$$= 0.75 \times \text{Pipeline Depth}$$
$$\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth}/(0.75 \times \text{Pipeline Depth}) = 1.33$$

- Machine A is 1.33 times faster

Evaluating Branch Alternatives

Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken; Deeper pipeline: target & condition known @ end of 3rd & 4th stage respectively

What's the addition to CPI?

Scheduling scheme	*Uncond penalty*	*Untaken penalty*	*Taken. penalty*
Stall pipeline	2	3	3
Predict taken	2	3	2
Predict not taken	2	0	3

Scheduling scheme	*Uncond branches*	*Untaken branches*	*Taken branches*	*All branches*
	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.3	0.56
Predict taken	0.08	0.18	0.2	0.46
Predict not taken	0.08	0.00	0.3	0.38

More branch evaluations

- Suppose the branch frequencies (as percentage of all instructions) of 15% cond. Branches, 1% jumps and calls, and 60% cond. Branches are taken. Consider a 4-stage pipeline where branch is resolved at the end of the 2nd cycle for uncond. Branches and at the end of the 3rd cycle for cond. Branches. How much faster would the machine be without any branch hazards, ignoring other pipeline stalls?

$$\text{Pipeline speedup}_{\text{ideal}} = \text{Pipeline depth}/(1+\text{Pipeline stalls})$$
$$= 4/(1+0) = 4$$

$$\text{Pipeline stalls}_{\text{real}} = (1\text{x}1\%) + (2\text{x}9\%) + (0\text{x}6\%) = 0.19$$

$$\text{Pipeline speedup}_{\text{real}} = 4/(1+0.19) = 3.36$$

$$\text{Pipeline speedup}_{\text{without control hazards}} = 4/3.36 = 1.19$$

More branch question

- A reduced hardware implementation of the classic 5-stage RISC pipeline might use the EX stage hardware to perform a branch instruction comparison and then not actually deliver the branch target PC to the IF stage until the clock cycle in which the branch reaches the MEM stage. Control hazard stalls can be reduced by resolving branch instructions in ID, but improving performance in one aspect may reduce performance in other circumstances.
- How does determining branch outcome in the ID stage have the potential to increase data hazard stall cycles?

Problems with Pipelining

- **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)
- Problem: It must appear that the exception or interrupt must appear between 2 instructions (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 can take place
- The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}

Homework

- **C.1.(e)**
  - **10-stage pipeline, full forwarding & bypassing, predicted taken**
  - **Branch outcomes and targets are obtained at the end of the ID stage.**

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LD R1,0(R2)	IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2				
DADDI R1, R1, #1		IF1	IF2	……										
SD R1, 0(R2)			IF1	……										

Homework

- **C.1.(e)**
  - **"For example, data are forwarded from the output of the second EX stage to the input of the first EX stage, still causing 1-cycle delay"**
  - **Example: (not as the same as the problem)**

Data Dependence: Read After Write

	1	2	3	4	5	6	7	8	9	10	11	12	13
ADD R1, R2, R3	IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2			
SUB R4, R5, R1		IF1	IF2	ID1	ID2	stall	EX1	EX2	MEM1	MEM2	WB1	WB2	

No Data Dependence

	1	2	3	4	5	6	7	8	9	10	11	12	13
ADD R1, R2, R3	IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2			
SUB R4, R5, R6		IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2		

Homework

- **C.1.(g)**
  - **CPI = The total number of clock cycles / the total number of instructions of code segment**

Homework

- C.2.(a) & C.2.(b)
  - "Assuming that only the first pipe stage can always be done independent of whether the branch goes"
  - An indicator of "predicted not taken"
  - The first stage of the instruction after the branch instruction could be done
  - Example: 4-stage pipeline
  - Unconditional branch: "at the end of the second cycle"

	1	2	3	4	5	6
J loop	IF	ID	EX	WB		
Next instruction		IF				
Target instruction			IF	ID	EX	WB

Homework

- C.2.(a) & C.2.(b)
  - Conditional Branch: "at the end of the third cycle"
  - If the branch is not taken, penalty = ?

	1	2	3	4	5	6	
BNEZ R4, loop	IF	ID	EX	WB			
Next instruction		IF	stall	ID	EX	WB	
Next + 1 instruction				IF	ID	EX	WB

  - If the branch is taken, penalty = ?

	1	2	3	4	5	6	
BNEZ R4, loop	IF	ID	EX	WB			
Next instruction		IF	stall	ID	EX	WB	
Target instruction				IF	ID	EX	WB