

Instruction Set Architecture (ISA)

- The Instruction Set Architecture (ISA) is the part of the processor that is visible to the programmer or compiler writer. The ISA serves as the boundary between software and hardware. We will briefly describe the instruction sets found in many of the microprocessors used today. The ISA of a processor can be described using 5 categories:

1. Operand Storage in the CPU

Where are the operands kept other than in memory?

2. Number of explicit named operands

How many operands are named in a typical instruction.

3. Operand location

Can any ALU instruction operand be located in memory? Or must all operands be kept internally in the CPU?

4. Operations

What operations are provided in the ISA.

5. Type and size of operands

What is the type and size of each operand and how is it specified?
Of all the above the most distinguishing factor is the first.

The 3 most common types of ISAs are:

Stack - The operands are implicitly on top of the stack.

Accumulator - One operand is implicitly the accumulator.

General Purpose Register (GPR) - All operands are explicitly mentioned, they are either registers or memory locations.

Lets look at the assembly code of

$C = A + B$;

in all 3 architectures:

Stack	Accumulator	GPR
PUSH A	LOAD A	LOAD R1,A
PUSH B	ADD B	ADD R1,B
ADD	STORE C	STORE R1,C
POP C -	-	

Not all processors can be neatly tagged into one of the above categories. The i8086 has many instructions that use implicit operands although it has a general register set. The i8051 is another example, it has 4 banks of GPRs but most instructions must have the A register as one of its operands.

What are the advantages and disadvantages of each of these approach's?

Stack

Advantages: Simple Model of expression evaluation (reverse polish). Short instructions.

Disadvantages: A stack can't be randomly accessed This makes it hard to generate efficient code. The stack itself is accessed every operation and becomes a bottleneck.

Accumulator

Advantages: Short instructions.

Disadvantages: The accumulator is only temporary storage so memory traffic is the highest for this approach.

GPR

Advantages: Makes code generation easy. Data can be stored for long periods in registers.

Disadvantages: All operands must be named leading to longer instructions.

Earlier CPUs were of the first 2 types but in the last 15 years all CPUs made are GPR processors. The 2 major reasons are that registers are faster than memory, the more data that can be kept internaly in the CPU the faster the program wil run. The other reason is that registers are easier for a compiler to use

A "Typical" RISC ISA

- 32-bit fixed format instruction (3 formats)
- 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
- Delayed branch

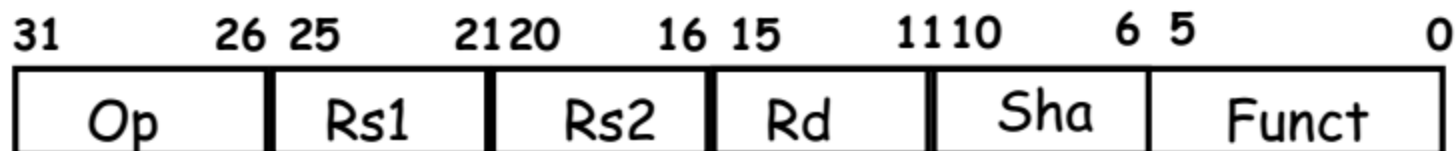
see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC,
CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Approaching an ISA

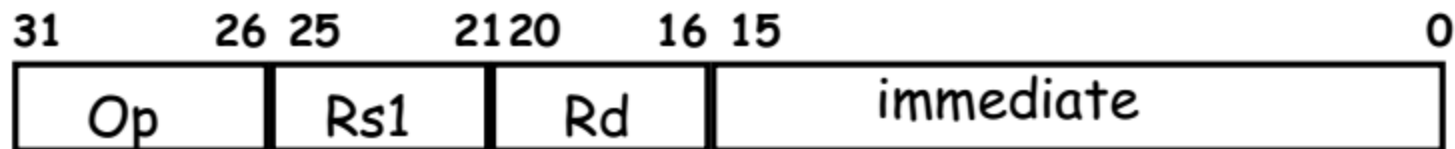
- **Instruction Set Architecture**
 - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- **Meaning of each instruction is described by register transfer logic (RTL) on *architected registers* and memory**
- **Given technology constraints 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**

Example: MIPS (- MIPS)

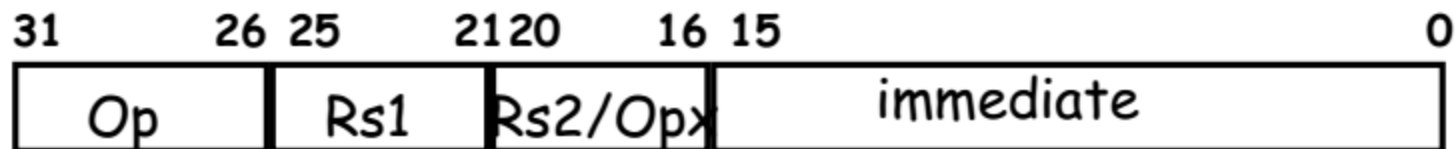
Register-Register



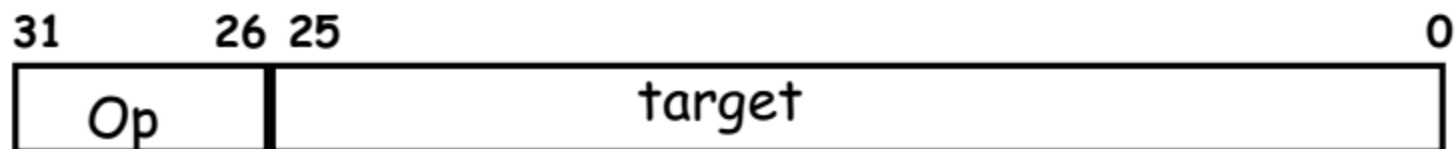
Register-Immediate



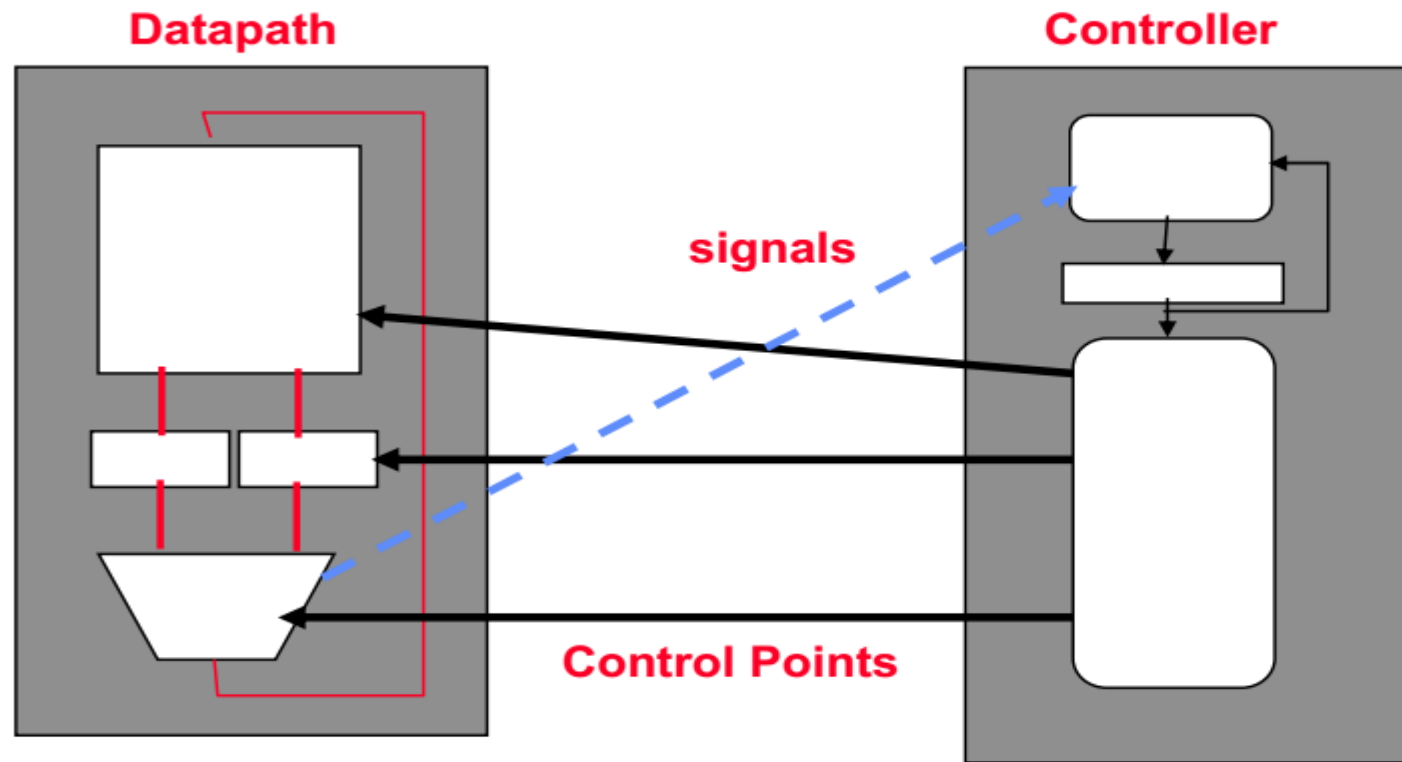
Branch



Jump / Call



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

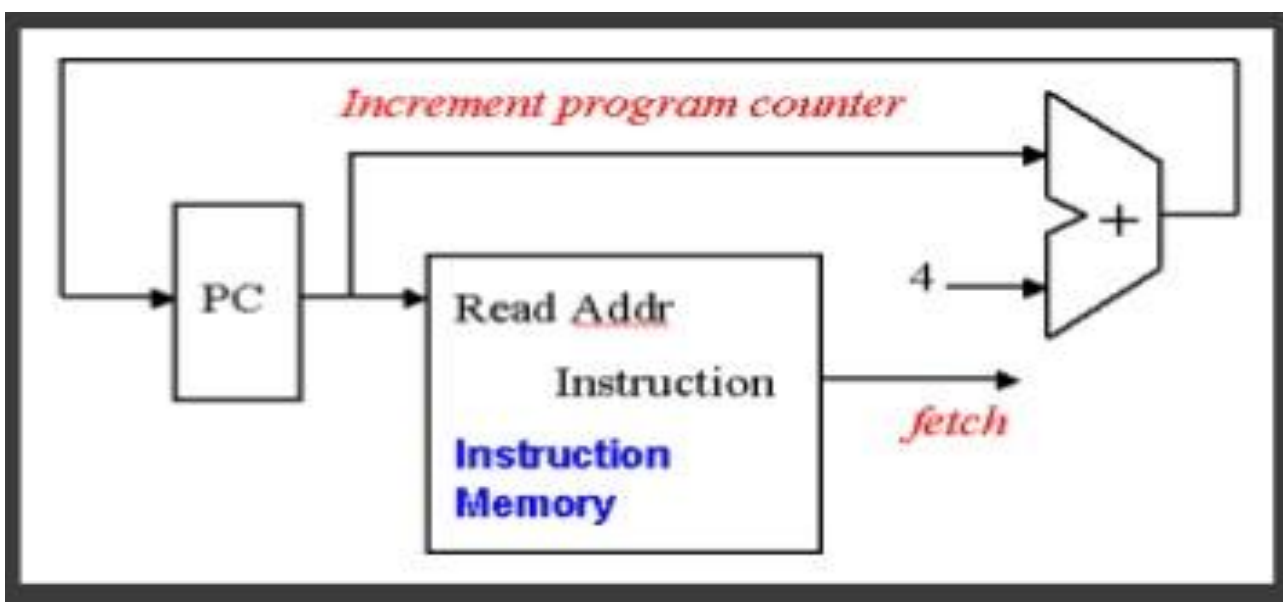
Based on desired function and signals

Datapath

- A **datapath** is a collection of functional units, such as arithmetic logic units or multipliers, that perform data processing operations, registers, and buses. Along with the control unit it composes the central processing unit (CPU).

General discipline for datapath design

- (1) determine the instruction classes and formats in the ISA,
- (2) design datapath components and interconnections for each instruction class or format, and
- (3) compose the datapath segments designed in Step 2) to yield a composite datapath



Load/Store Datapath

- **Load/Store Datapath:** The load/store datapath uses instructions where *offset* denotes a memory address offset applied to the base address in register .
- Fetch instruction from instruction memory and increment PC
- Read register value from the register file
- Result from ALU is applied as an address to the data memory

Example:

Register R=base address of an array of words, word=4 bytes and memory organized as location= byte
Load or store to reach an element of this array the physical address of this elements calculated as following

physical address= $R + \text{index of element} * 4$

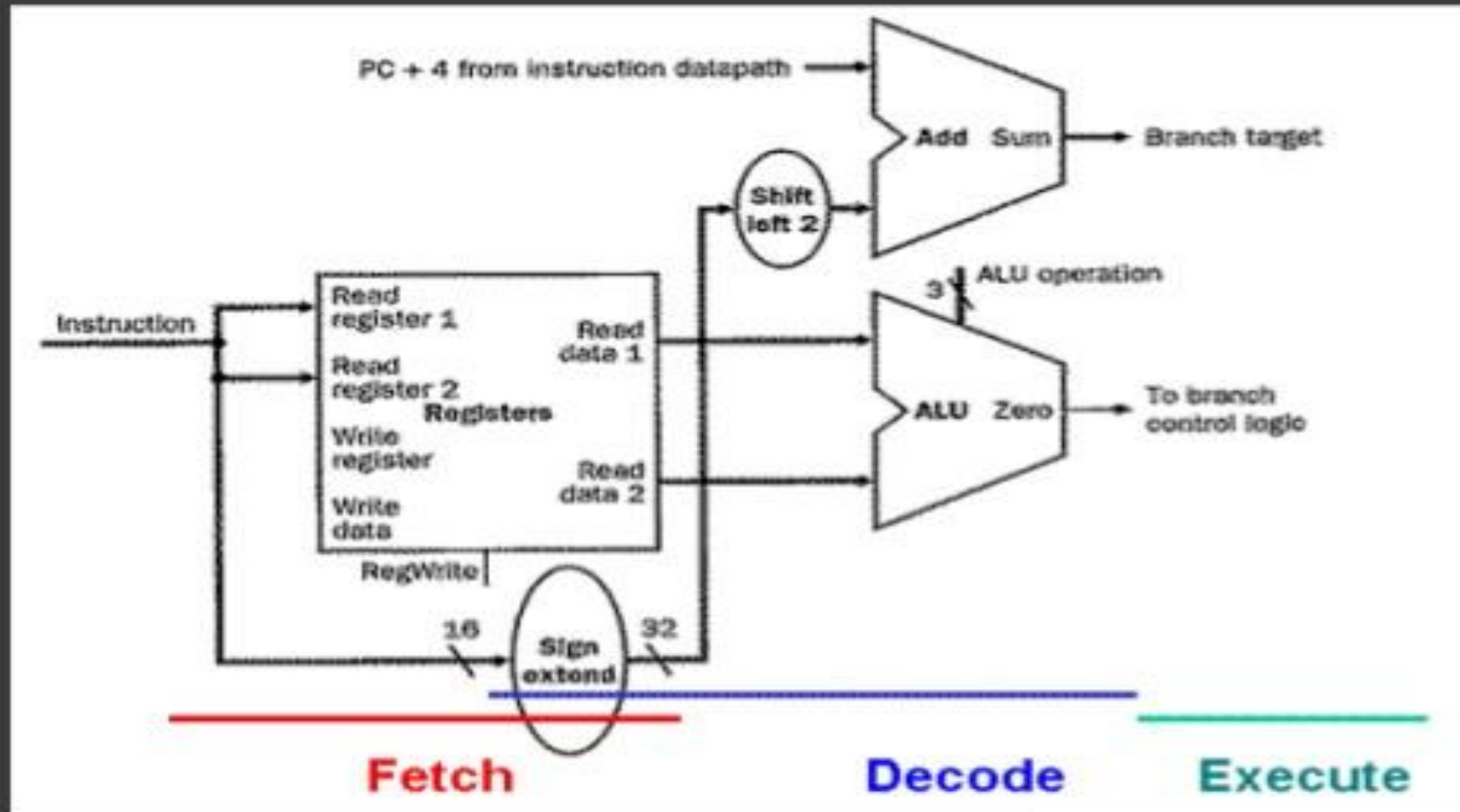
If $R=20000$, and $\text{index}=7$

Physical address of this element= $20000 + 7 * 4 = 20028$

Branch/Jump Datapath

- **Branch/Jump Datapath:** The branch datapath (jump is an unconditional branch) uses instructions such as **offset**, where *offset* is a 16-bit offset for computing the branch target address via PC-relative addressing.
- *Register Access* takes input from the register file, to implement the *instruction fetch* or *data fetch* step of the fetch-decode-execute cycle.

Schematic diagram of the Branch instruction datapath



ALU

- Arithmetic Logic Unit :
- »A and B are two n-bit inputs »FS is m-bit function select code »F is n-bit result »Status bits to provide more information about result F :
 - ► $Z = 1$ result is zero
 - ► $N = 1$ result is negative
 - ► $V = 1$ signed overflow
 - ► $C = 1$ carry out

ALU control codes

- Given the simple datapath shown in Figure:

ALU Control	Input Function
000	and
001	or
010	add
110	sub

Main Control Unit

- The first step in designing the main control unit is to identify the fields of each instruction and the required control lines to implement the datapath shown in Figure
- Recalling the three MIPS instruction formats (R, I, and J), shown as follows:

Main Control Unit

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	op	rs	rt	rd	shamt	funct
I:	op	rs	rt	address / immediate		

op: basic operation of the instruction (opcode)

rs: first source operand register

rt: second source operand register

rd: destination operand register

shamt: shift amount

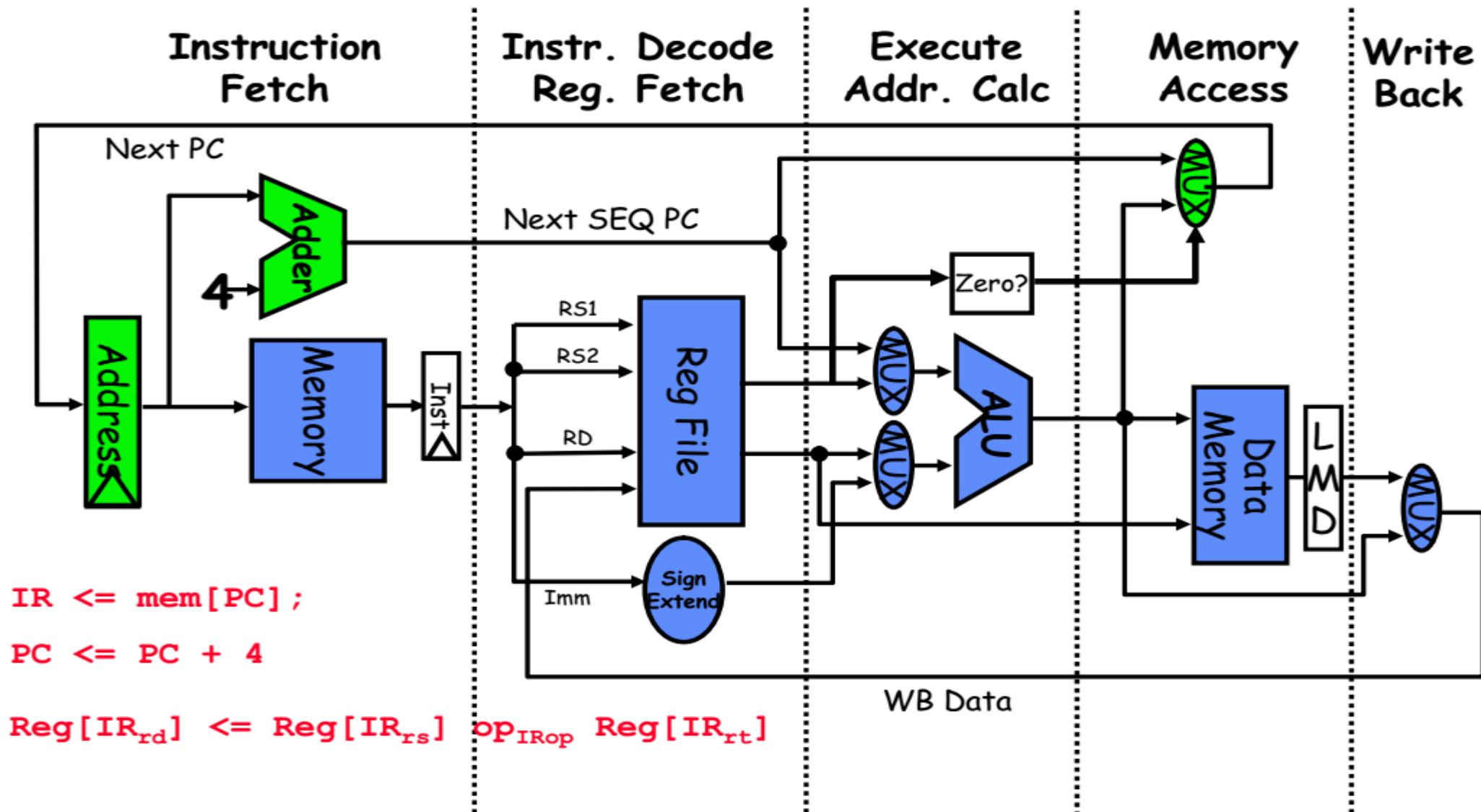
funct: selects the specific variant of the opcode (function code)

address: offset for load/store instructions ($\pm 2^{15}$)

immediate: constants for immediate instructions

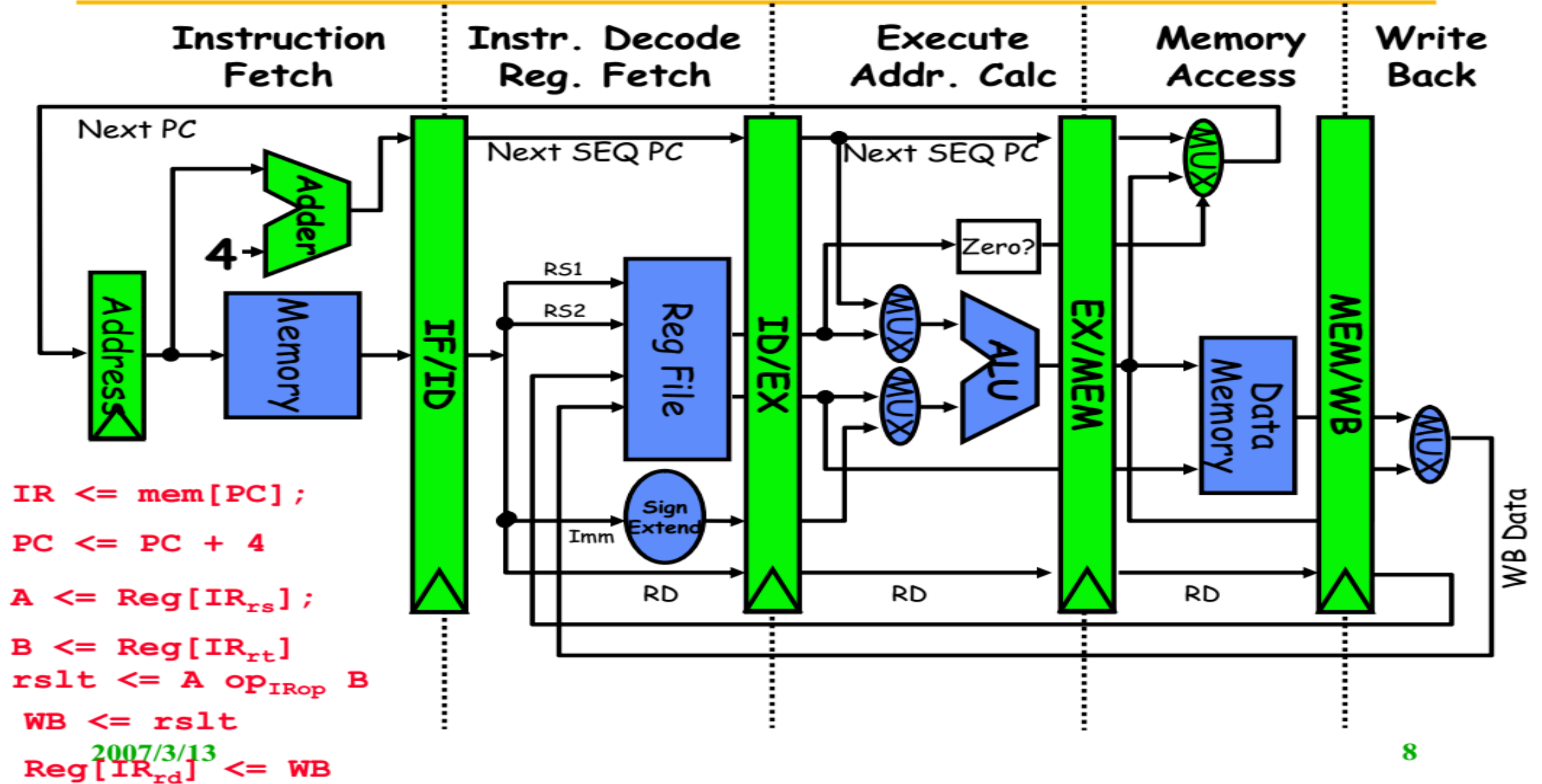
5 Steps of MIPS Datapath

Figure A.2, Page A-8

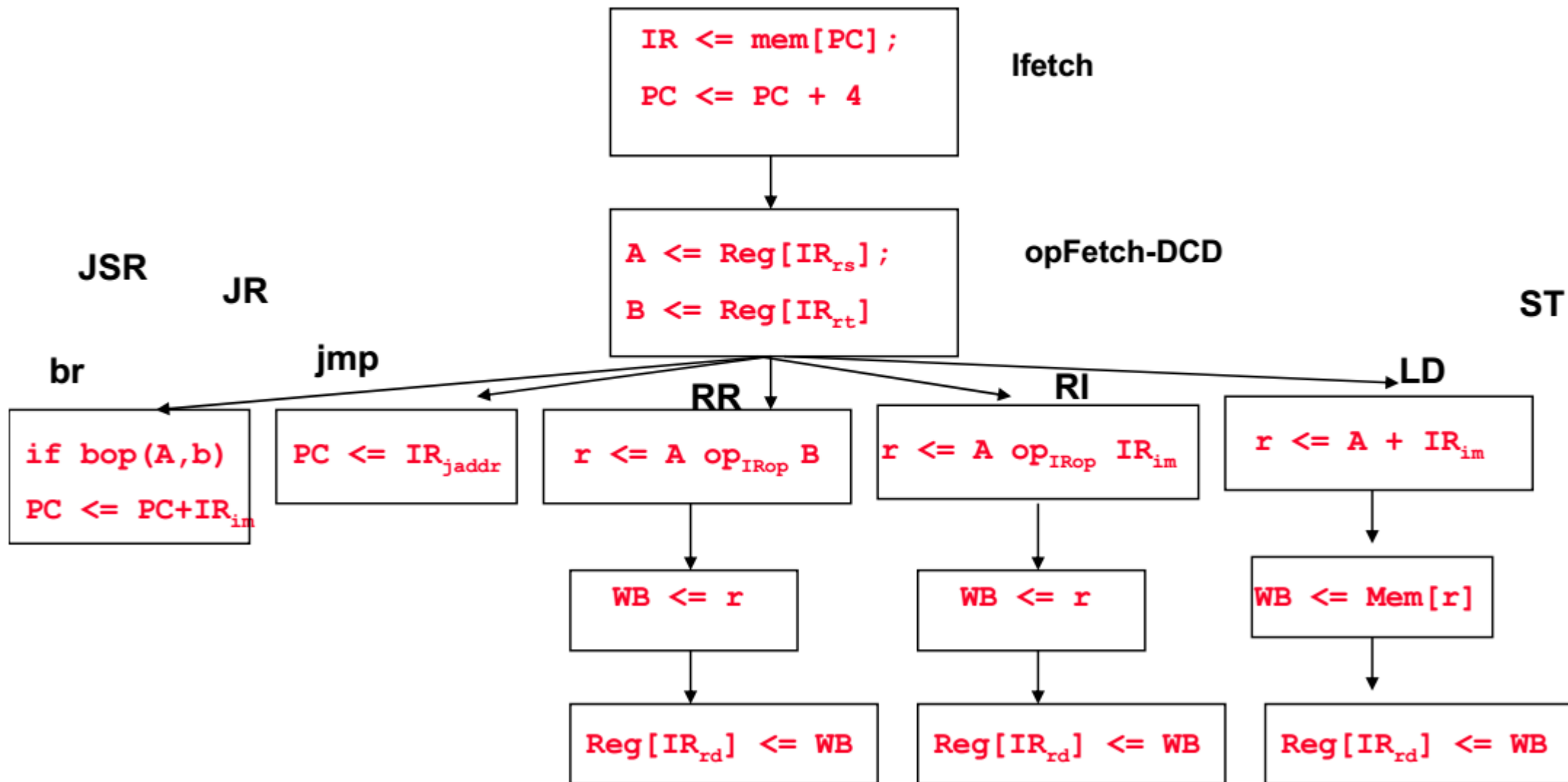


5 Steps of MIPS Datapath

Figure A.3, Page A-9

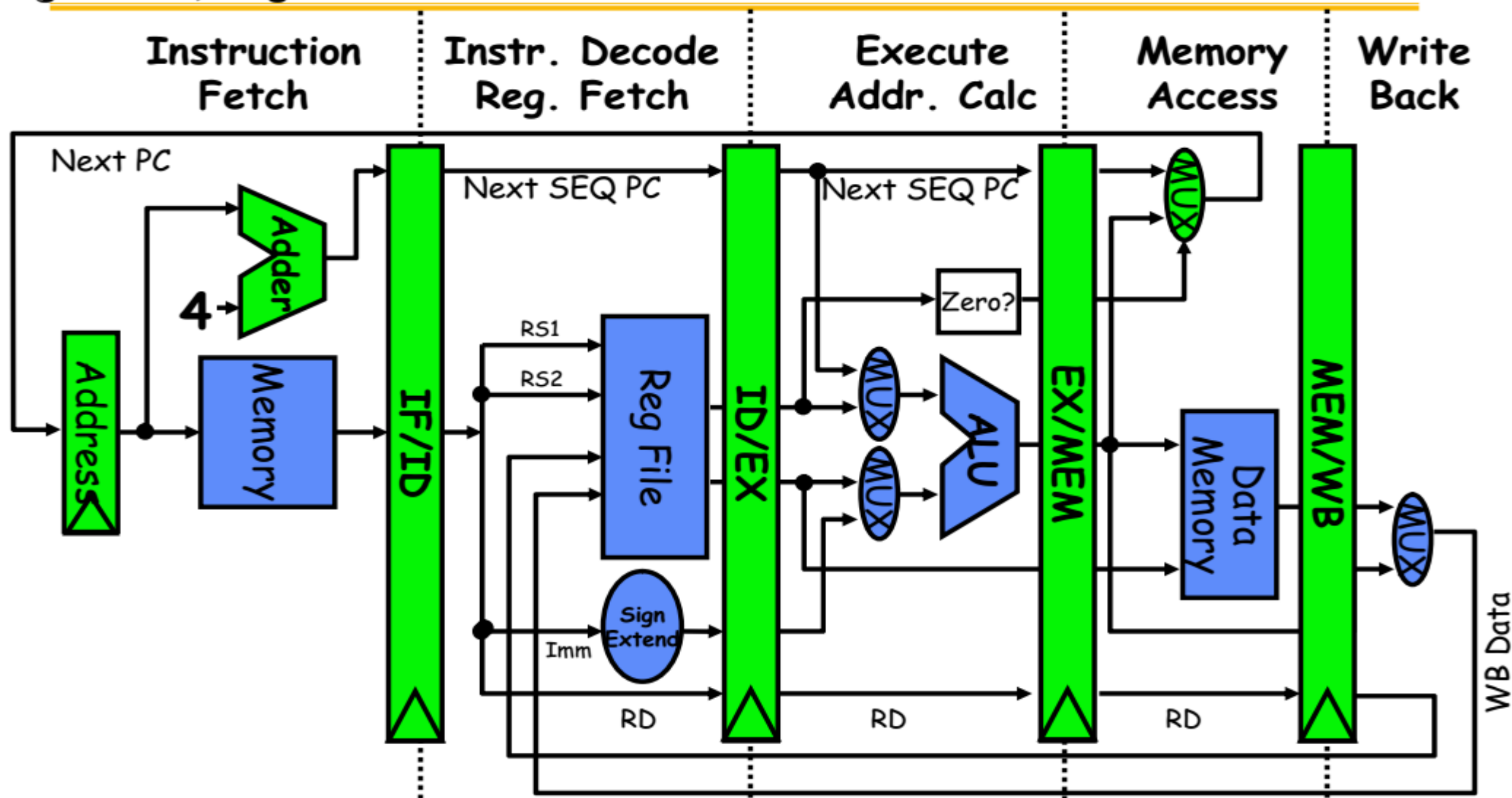


Inst. Set Processor Controller



5 Steps of MIPS Datapath

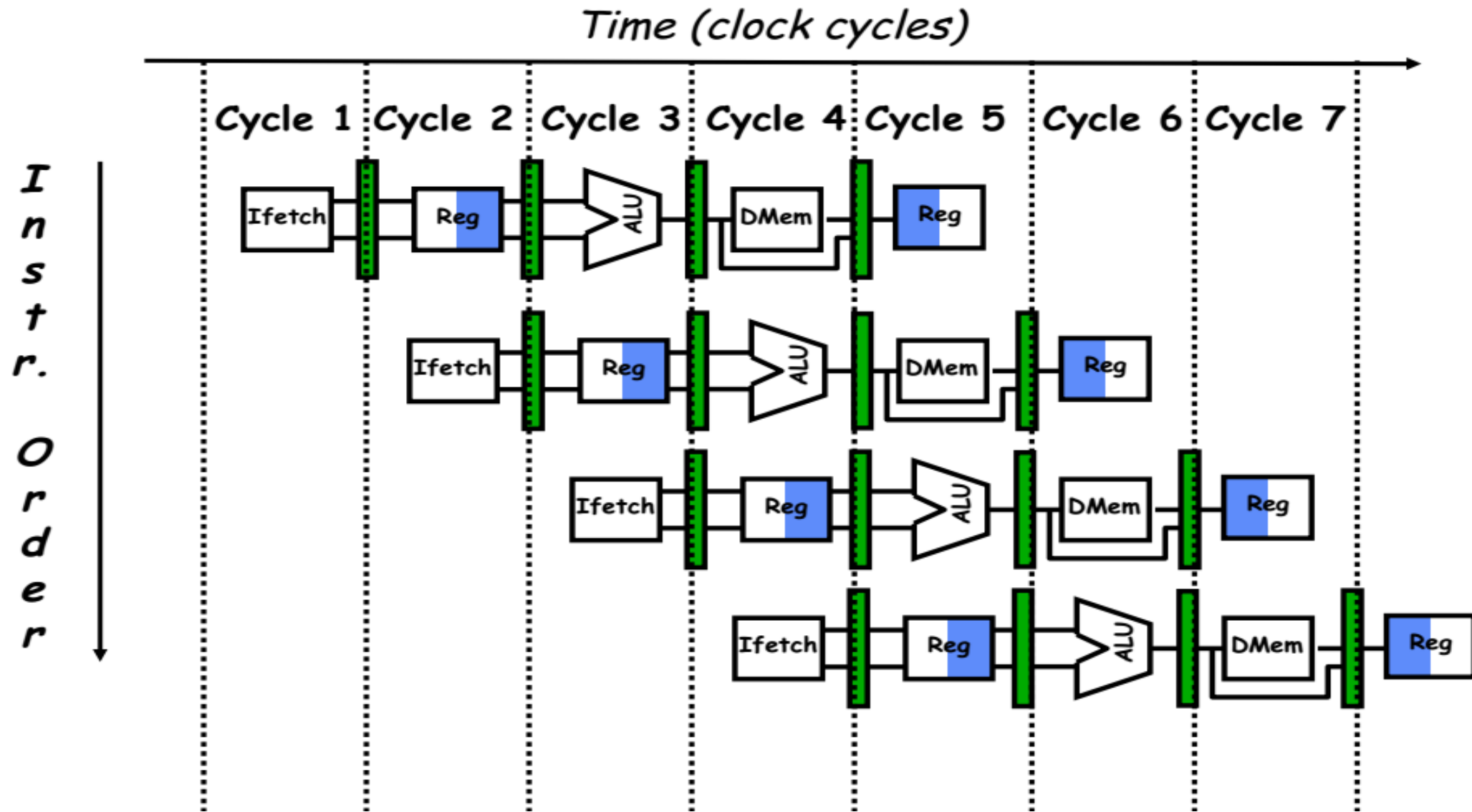
Figure A.3, Page A-9



- Data stationary control
 - local decode for each instruction phase / pipeline stage

Visualizing Pipelining

Figure A.2, Page A-8

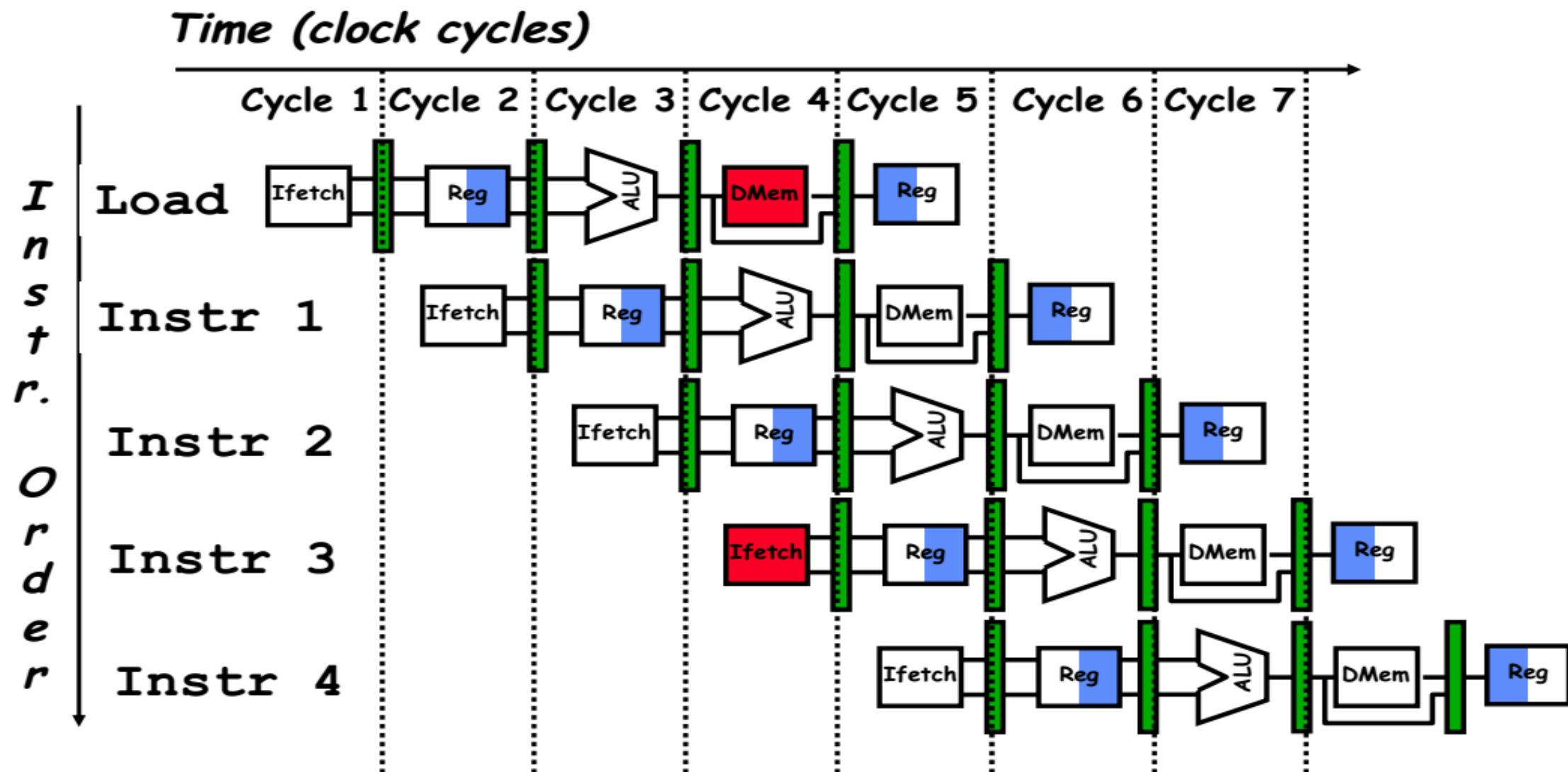


Pipelining is not quite that easy!

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

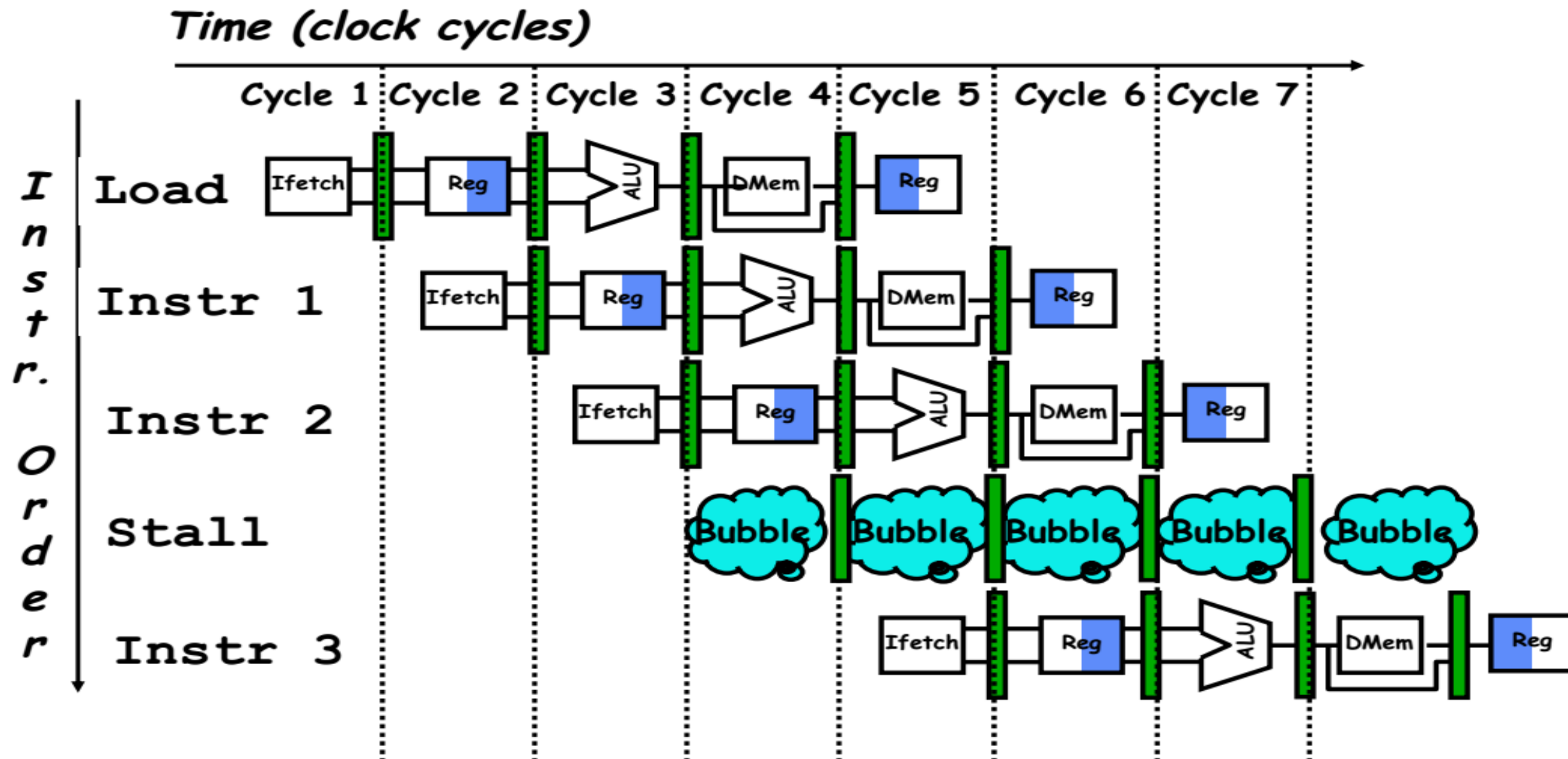
One Memory Port/Structural Hazards

Figure A.4, Page A-14



One Memory Port/Structural Hazards

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



How do you "bubble" the pipe?

Speed Up Equation for Pipelining

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

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

For simple RISC pipeline, CPI = 1:

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

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

$$\begin{aligned}\text{SpeedUp}_A &= \text{Pipeline Depth} / (1 + 0) \times (\text{clock}_{\text{unpipe}} / \text{clock}_{\text{pipe}}) \\ &= \text{Pipeline Depth}\end{aligned}$$

$$\begin{aligned}\text{SpeedUp}_B &= \text{Pipeline Depth} / (1 + 0.4 \times 1) \times (\text{clock}_{\text{unpipe}} / (\text{clock}_{\text{unpipe}} / 1.05)) \\ &= (\text{Pipeline Depth} / 1.4) \times 1.05 \\ &= 0.75 \times \text{Pipeline Depth}\end{aligned}$$

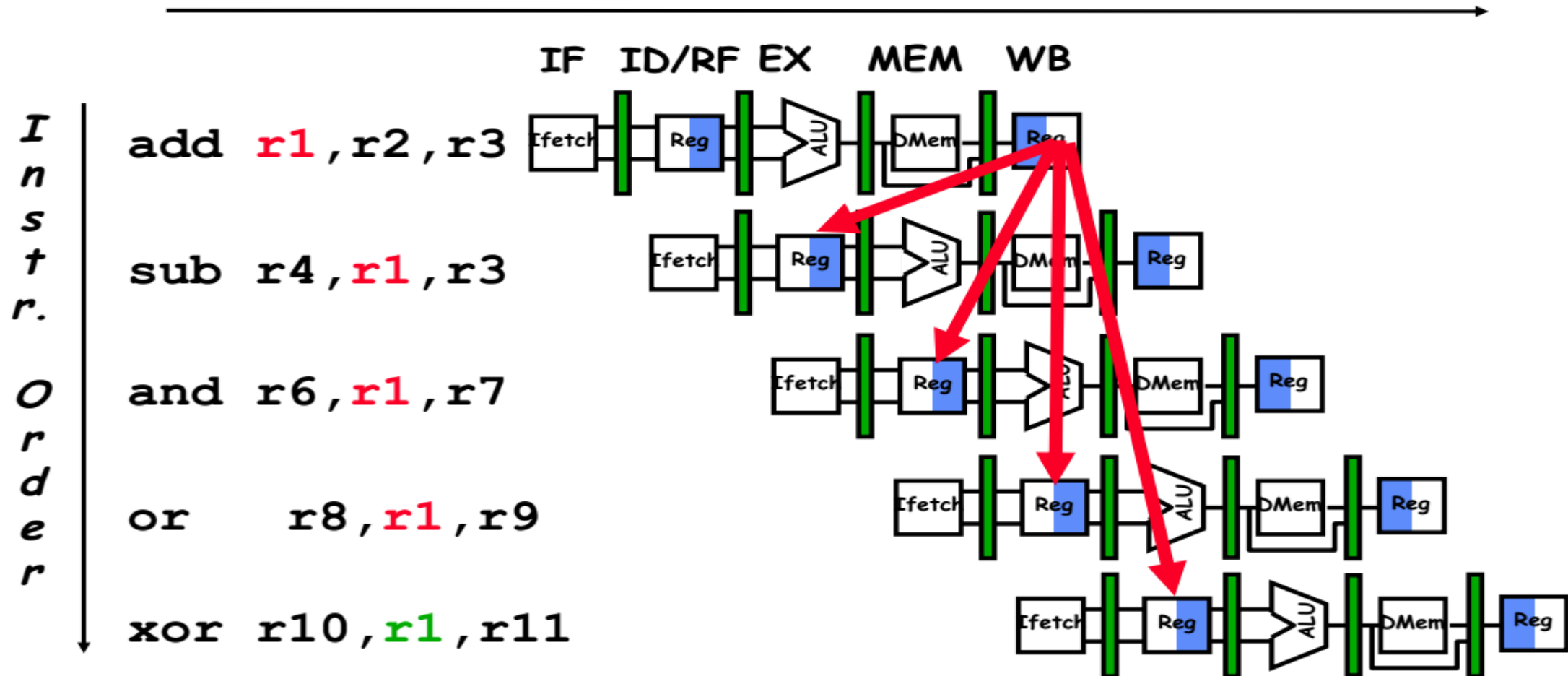
$$\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth} / (0.75 \times \text{Pipeline Depth}) = 1.33$$

- Machine A is 1.33 times faster

Data Hazard on R1

Figure A.6, Page A-17


Time (clock cycles)



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.