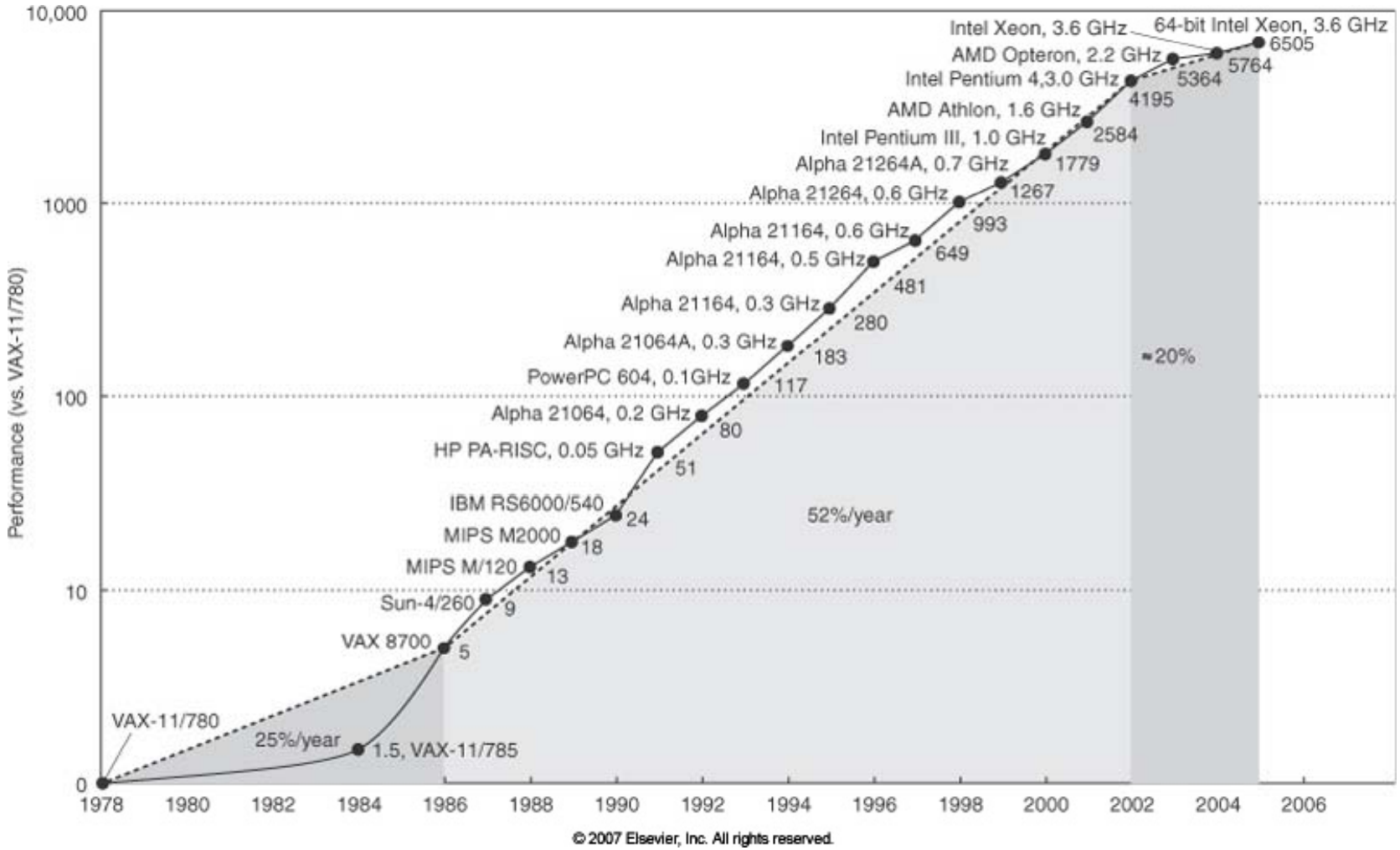
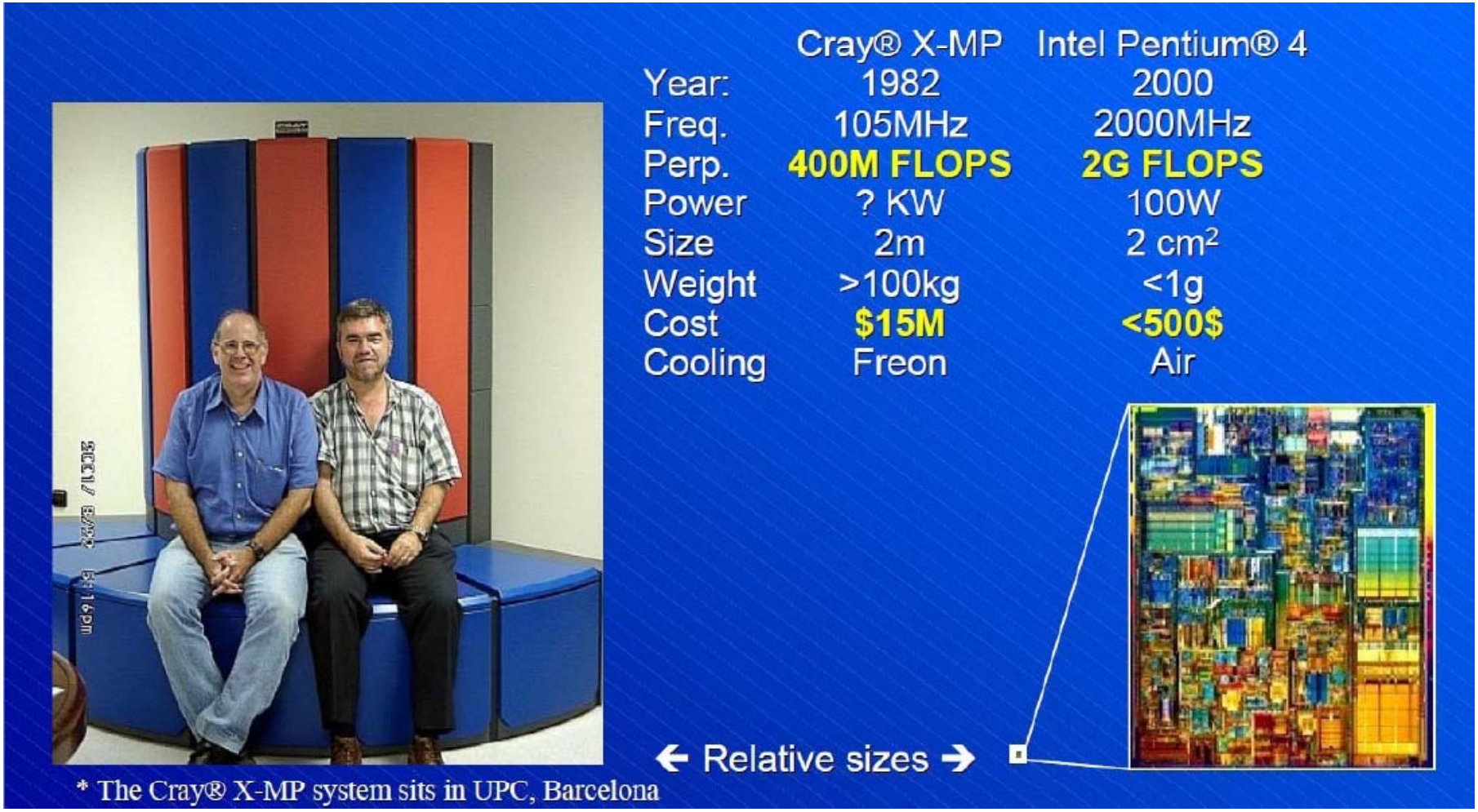
Processor Architecture

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Introduction (1)



Introduction (2)



Source: Intel

Trends (1)

- **Trends in computer architectures**
  - Pre-WWII: Mechanical calculating machines
  - WWII-50's: Technology improvement
    - Relays → vacuum tubes
    - High-level languages
  - 60's: Miniaturization/Packaging
    - Transistors
    - Integrated Circuits (ICs)
  - 70's: Semantic Gap
    - Complex instruction set
    - Large support in hardware
    - Microcoding

Trends (2)

- **Trends in computer architectures (cont'd)**
  - 80's: Keep it simple
    - RISC (Reduced Instruction Set Computer)
    - Shift complexity to software
  - 90's: What to do with all these transistors?
    - Large on-chip cache
    - Prefetching hardware
    - Speculative execution
    - Special-purpose instructions and hardware
    - Multiple processors on-a-chip
    - …

Instruction Set Architecture (1)

- **CISC (Complex Instruction Set Computer)**
  - Dominant style through mid-80's
  - Add instructions to perform "typical" programming tasks
  - Stack-oriented instruction set
    - Use stack to pass arguments, save program counter
    - Explicit push and pop instructions
  - Arithmetic instructions can access memory
    - `addl %eax, 12(%ebx, %ecx, 4)`
    - Requires memory read/write & complex address calculation
  - Condition codes
    - Set as side effect of arithmetic and logical instructions

Instruction Set Architecture (2)

- **RISC (Reduced Instruction Set Computer)**
  - Fewer, simpler instructions
    - Might take more to get given task done
    - Can be decoded easily
    - Can execute them with small and fast hardware
  - Register-oriented instruction set
    - Many more (typically 32+) registers
    - Use for arguments, return pointer, temporaries
  - Only load and store instructions can access memory
    - Single address mode: base register + displacement
  - No condition codes
    - Test instructions return 0/1 in register

MIPS Instruction Formats

```
        add     $3, $2, $1              ; R[3] = R[2] + R[1]
```

R-type

op	rs	rt	rd	shamt	funct

```
        addu    $3, $2, 1234            ; R[3] = R[2] + 1234
        lw      $3, $2, 12              ; R[3] = Mem[R[2] + 12]
        beqz    $3, dest                ; if (R[3] == 0)
                                                PC = PC + 4 + dest
```

I-type

op	rs	rt	16 bit address

```
        j       dest                    ; PC = dest
        jal     dest                    ; R[31] = PC + 4, PC = dest
```

J-type

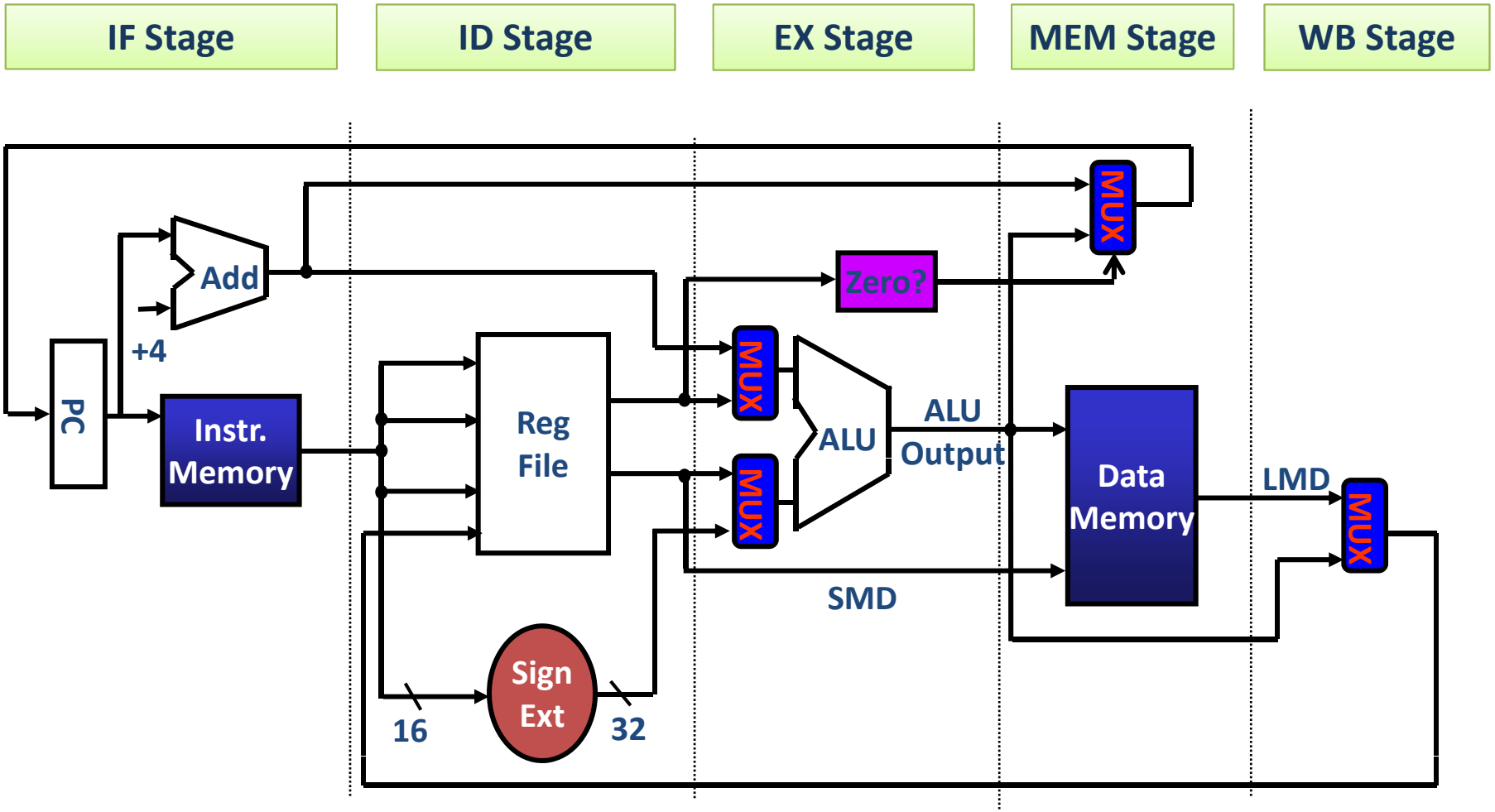
op	26 bit address

MIPS Registers

#	Name	Usage
0	$zero	The constant value 0
1	$at	Assembler temporary
2	$v0	Values for results and expression evaluation
3	$v1	
4	$a0	Arguments
5	$a1	
6	$a2	
7	$a3	
8	$t0	Temporaries (Caller-save registers)
9	$t1	
10	$t2	
11	$t3	
12	$t4	
13	$t5	
14	$t6	
15	$t7	

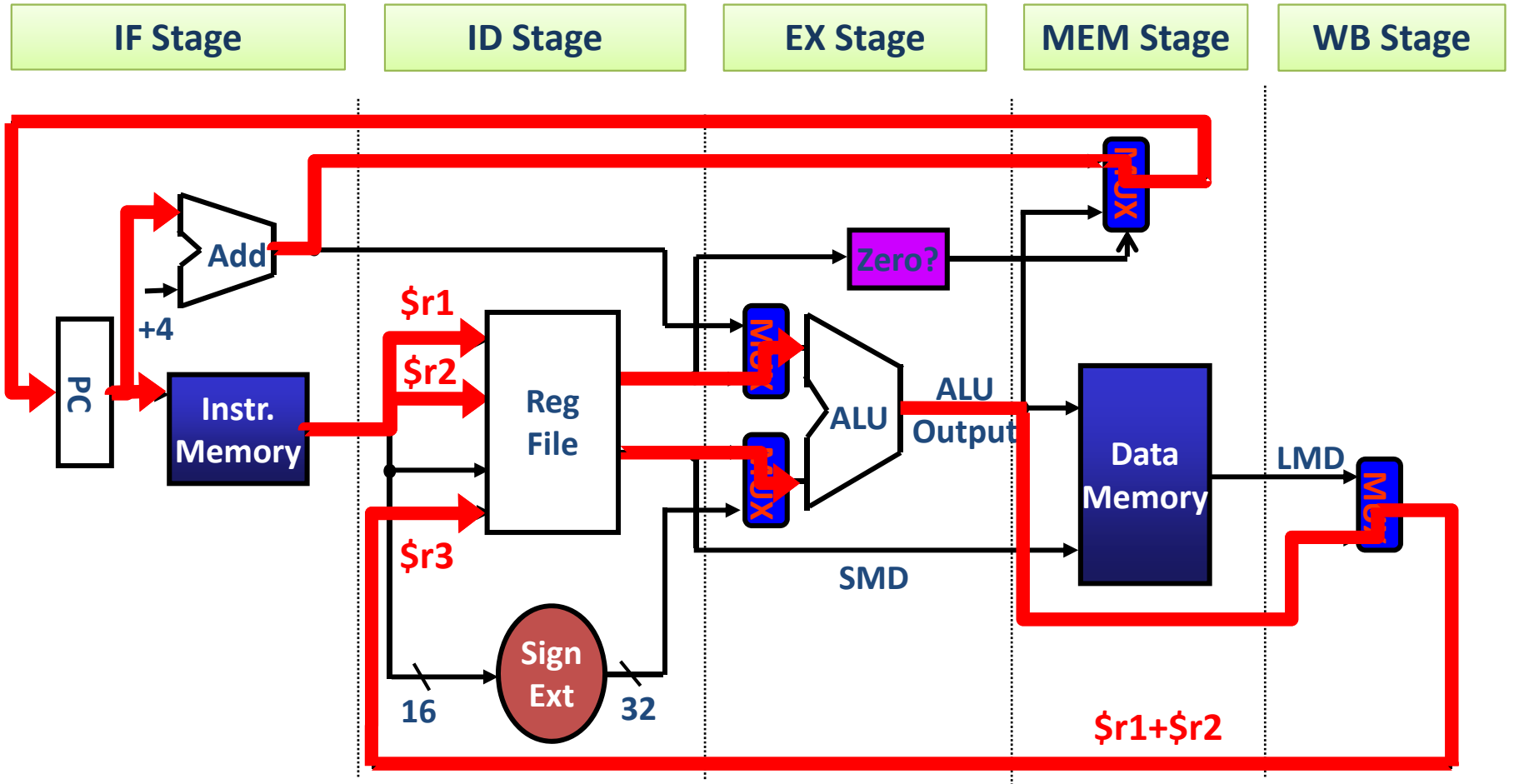
#	Name	Usage
16	$s0	Saved temporaries (Callee-save registers)
17	$s1	
18	$s2	
19	$s3	
20	$s4	
21	$s5	
22	$s6	
23	$s7	
24	$t8	More temporaries (Caller-save registers)
25	$t9	
26	$k0	Reserved for OS kernel
27	$k1	
28	$gp	Global pointer
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address

S-MIPS Datapath (5 stages)



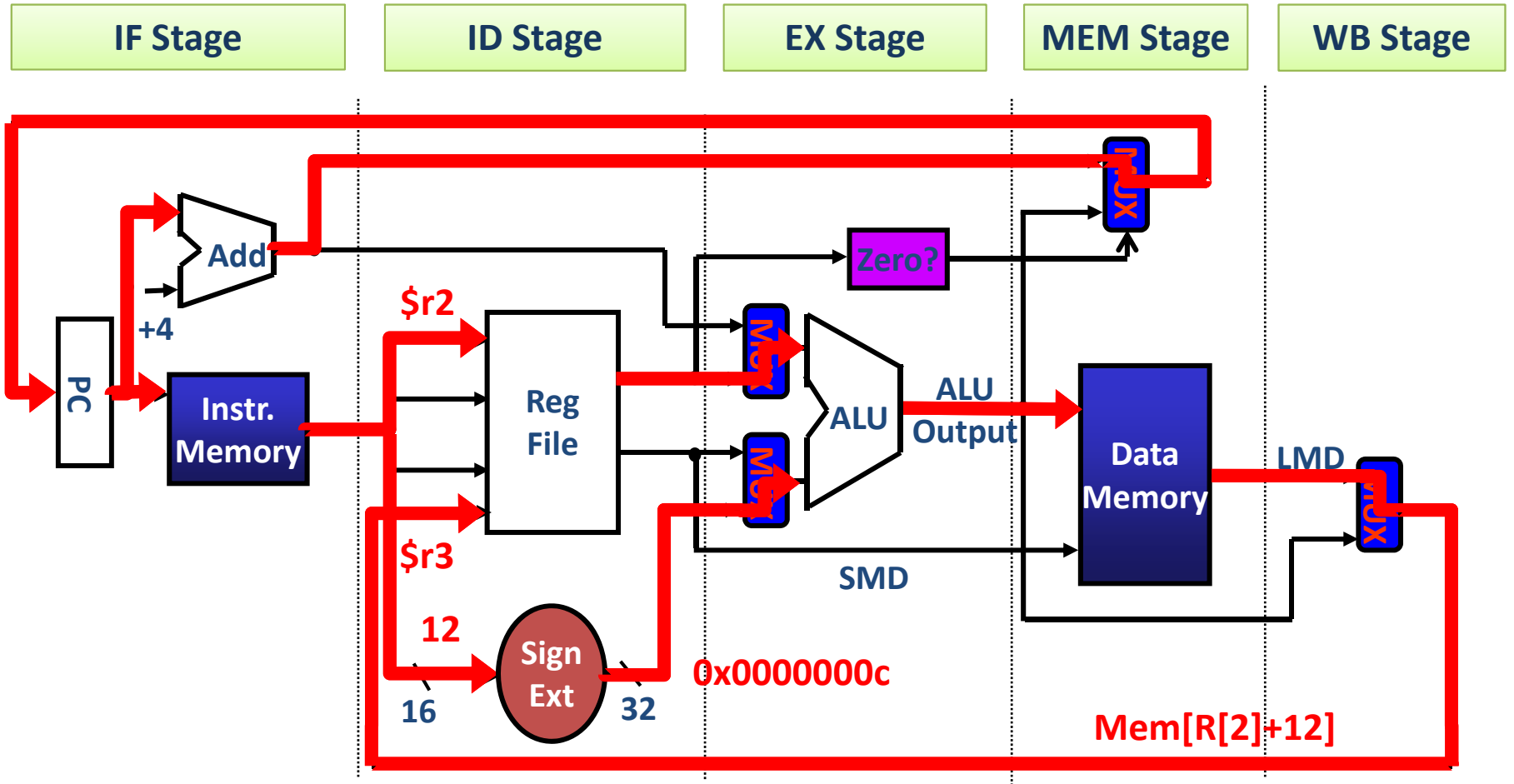
S-MIPS Datapath Example (1)

ALU instruction: add $r3, $r2, $r1 ; R[3]← R[2] + R[1]



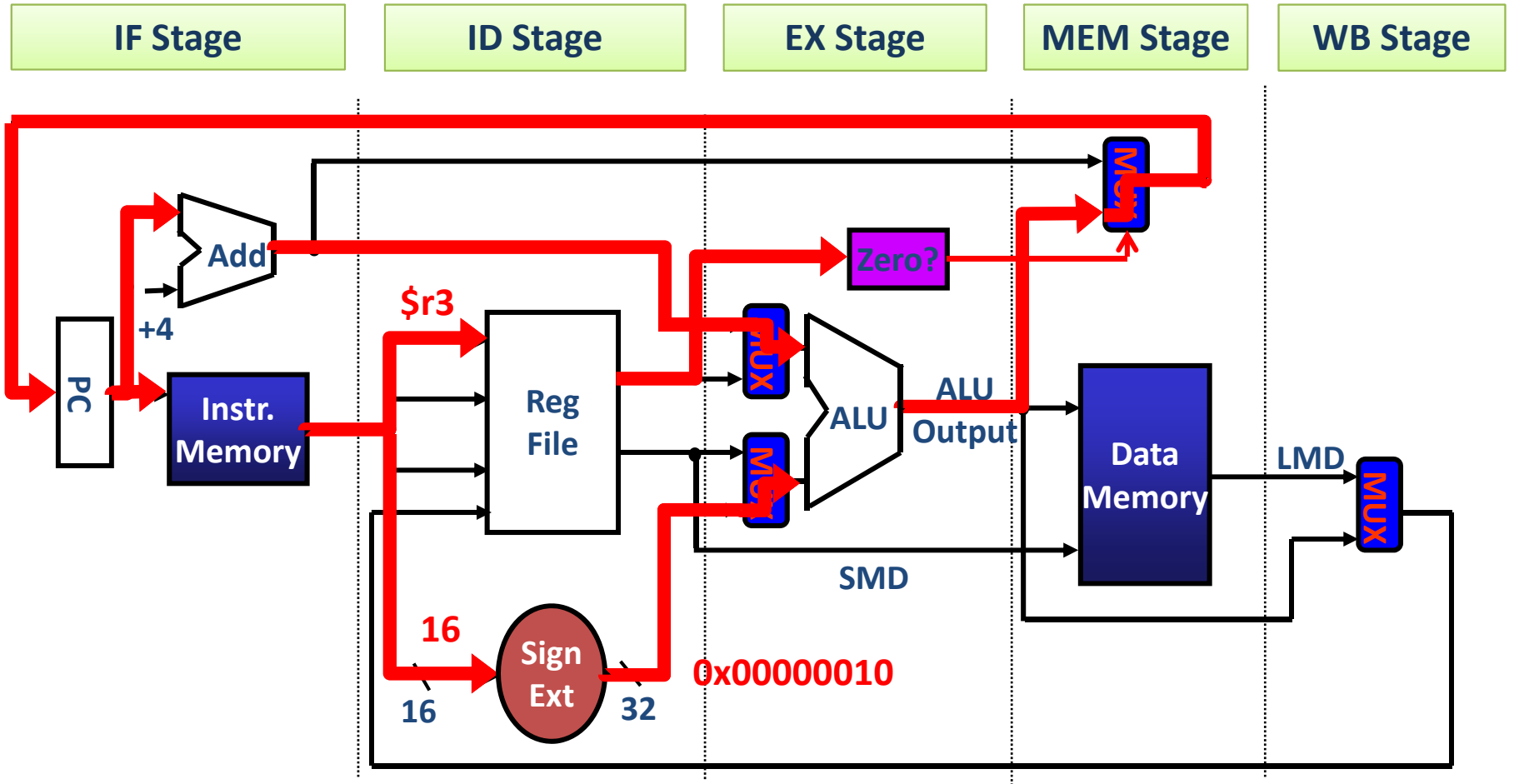
S-MIPS Datapath Example (2)

LOAD instruction: lw $r3, $r2, 12 ; R[3]← Mem[R[2] + 12]



S-MIPS Datapath Example (3)

Branch instruction: beqz $r3, 16 ; PC += (R[3]==0)? 4+16 : 4



Pipelining (1)

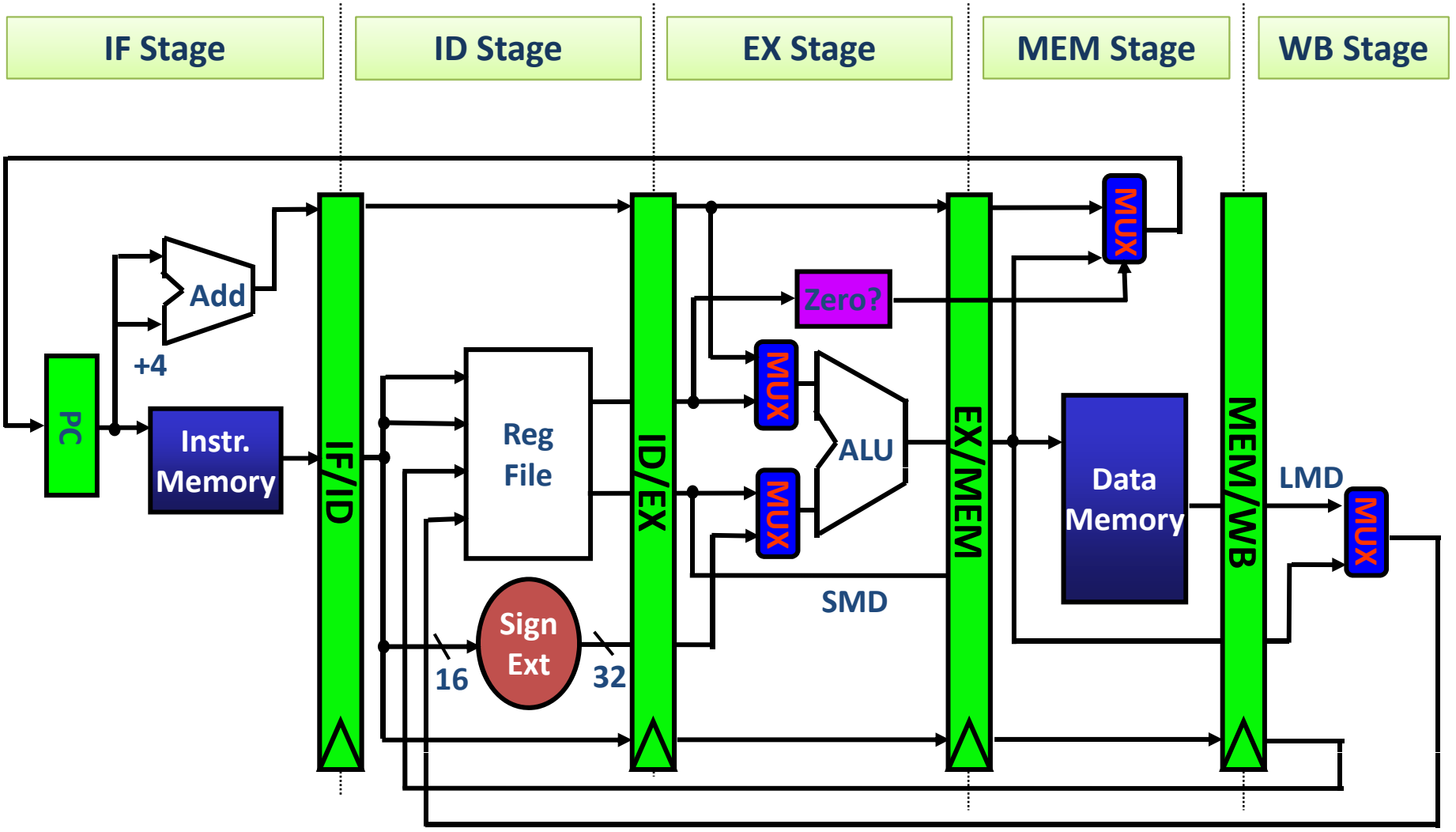
Pipelining (2)

- **Lessons**
  - Pipelining doesn't help latency of single task, it helps throughput of entire workload.
  - Pipeline rate limited by slowest pipeline stage.
  - Multiple tasks operating simultaneously.
  - Potential speedup = Number pipe stages
  - Unbalanced lengths of pipe stages reduces speedup.
  - Time to "fill" pipeline and time to "drain" it reduces speedup.

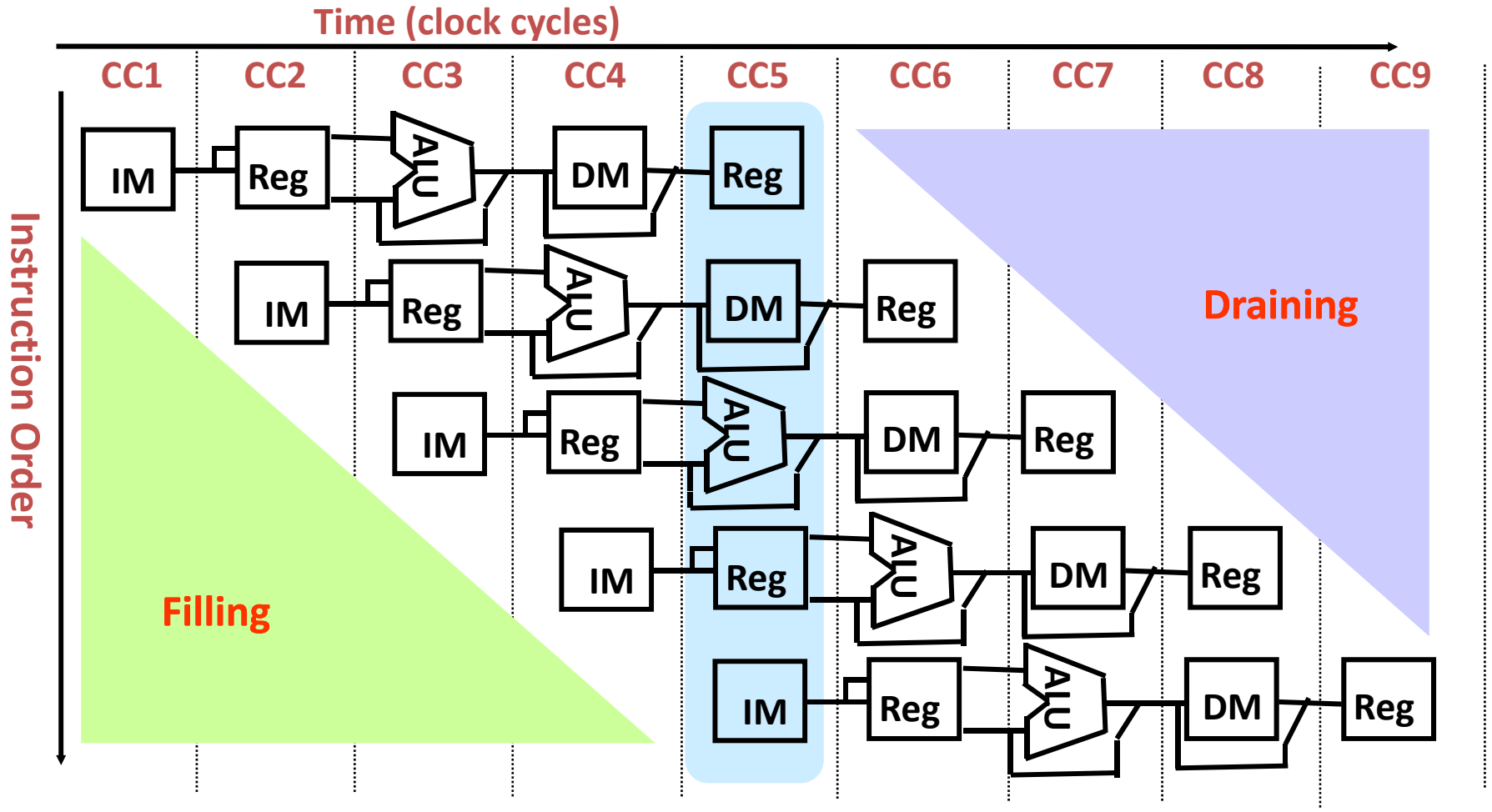
Instruction Pipelining

- **Why pipelining?**
  - Execute billions of instructions, so throughput is what matters.
- **What is desirable in instruction sets for pipelining?**
  - Variable length instructions vs. all instructions same length?
  - Memory operands part of any operation vs. memory operands only in loads or stores?
  - Register operand many places in instruction format vs. registers located in same place?

Pipelined S-MIPS Datapath



Visualizing Pipeline

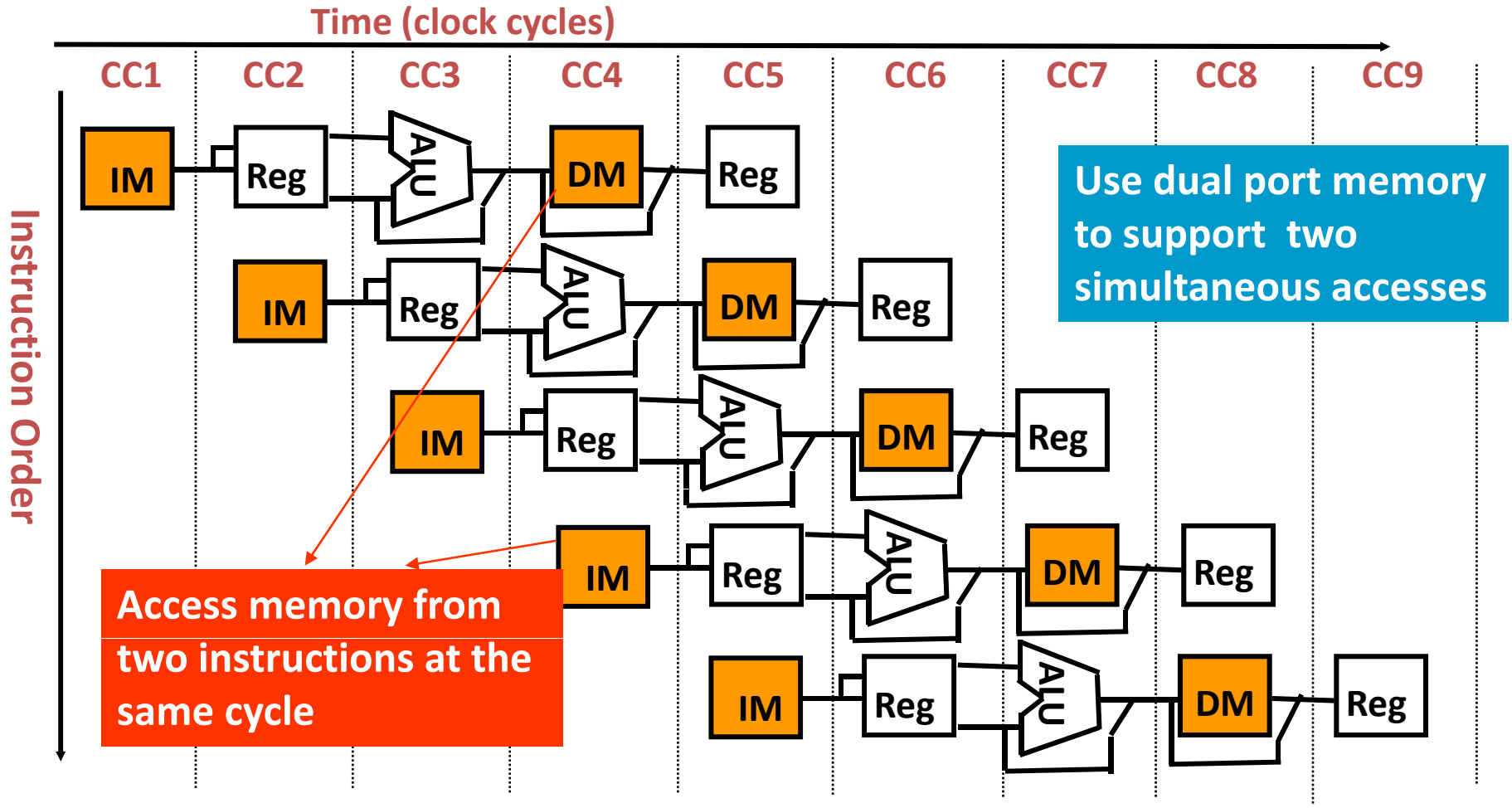


Pipeline Stall

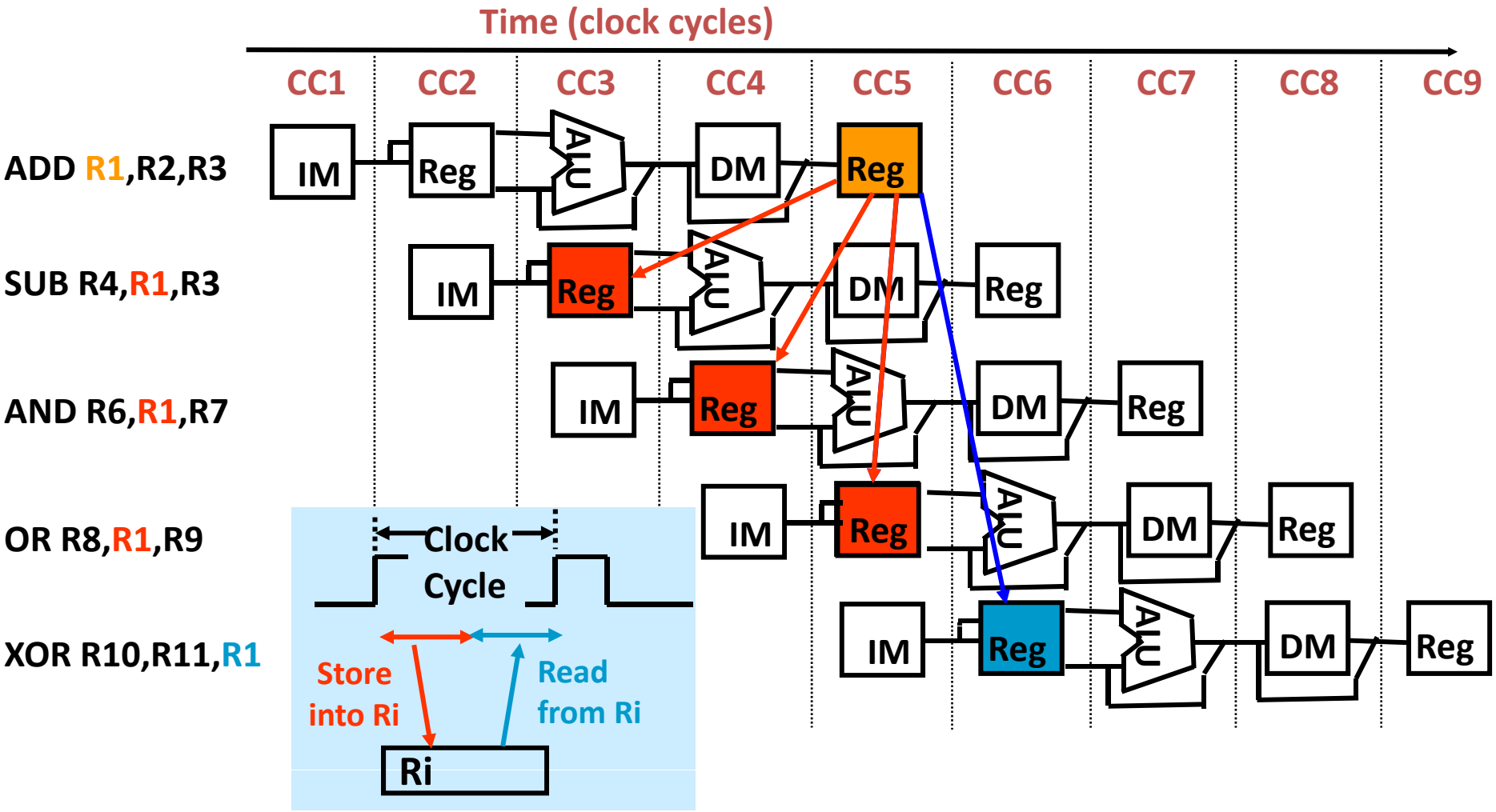
- **Limits to pipelining: hazards prevent the next instruction from executing during its designated clock cycle**
  - **Structural hazards**: HW cannot support the combination of instructions due to lack of HW capacity
  - **Data hazards**: Instruction depends on the result of prior instruction still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

- **Common solution is to stall the pipeline until the hazard is resolved, inserting one or more bubbles (idle clock cycles) in the pipeline.**

Structural Hazard

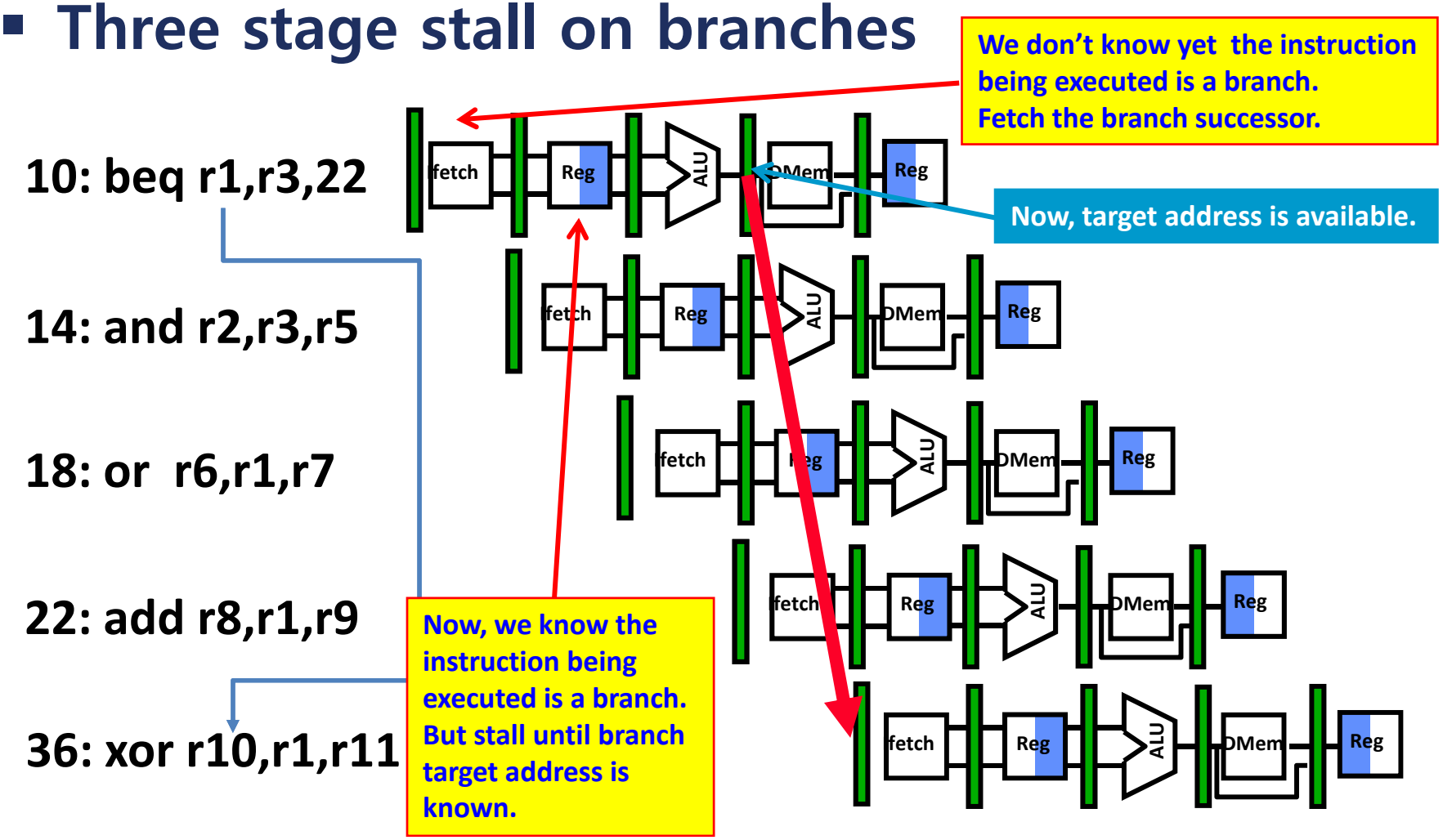


Data Hazard


Time (clock cycles)
CC1
CC2
CC3
CC4
CC5
CC6
CC7
CC8
CC9
ADD R1,R2,R3
SUB R4,R1,R3
AND R6,R1,R7
OR R8,R1,R9
XOR R10,R11,R1
IM
Reg
ALU
DM
Clock Cycle
Store into Ri
Read from Ri
Ri

Control Hazard

- Three stage stall on branches



Summary (1)

- **CISC vs. RISC**
  - CISC: easy for compiler, fewer code bytes
  - RISC: better for optimizing compilers, can make run fast with simple chip design
- **CISC vs. RISC: Current status**
  - For desktop processors, choice of ISA not a technical issue
    - With enough hardware, can make anything run fast
    - Code compatibility more important
  - For embedded processors, RISC makes sense
    - Smaller, cheaper, less power

Summary (2)

- **Pipelining**
  - Improved throughput
- **Problems in pipelining**
  - Structural hazards
  - Data hazards
  - Control hazards
- **Instruction set design affects complexity of pipeline implementation**