Outline

- Instruction set principles
  - What is an instruction set?
  - What is a good instruction set?
  - Instruction set aspects
  - RISC vs. CISC

  Instruction set examples: Appendices C, D, E, & F (online)
- Pipelining
  - Why pipelining?
  - Basic stages of a pipeline
  - Expected improvement
  - Complications?

[Hennessy/Patterson CA:AQA (3rd Edition): Chapter 2 & Appendix A]

Instruction Set Principles

[Hennessy/Patterson CA:AQA (3rd Edition): Chapter 2 & Appendix A]

Instruction Set Architecture (ISA)

"Instruction set architecture is the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine."

Source: IBM in 1964 when introducing the IBM 360 architecture

- An instruction set is a functional description of the processor
  - What operations can it do
  - What storage mechanisms does it support
- ISA defines the hardware/software interface

  A good interface:
  - Lasts through many implementations
  - Can be used in many different ways
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels

Instruction Set Design Issues

- What operations are supported?
  - add, sub, mul, move, compare . . .
- Where are operands stored?
  - Registers (how many of them are there), memory, stack, accumulator
- How many explicit operands are there?
  - 0, 1, 2, or 3
- How is the operand location specified?
  - register, immediate, indirect, . . .
- What type and size of operands are supported?
  - byte, int, float, double, string, vector, . . .

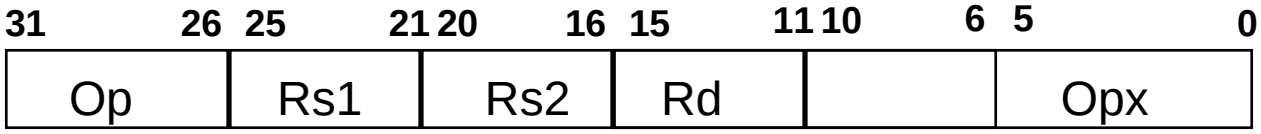
A "Typical" RISC

- 32-bit fixed format instruction (3 formats)
- Memory access only through load/store operations
- 32 32-bit general-purpose registers
  - R0 contains zero
  - Double precision operations take pair (floating point registers may be separate)
- 3-address (src1, src2, dst), register-register arithmetic instructions
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

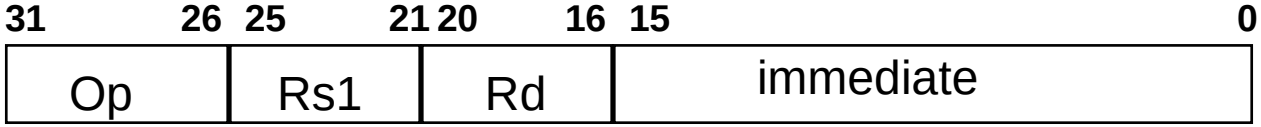
- Examples:
  SUN SPARC, **MIPS**, HP PA-RISC, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Example: MIPS

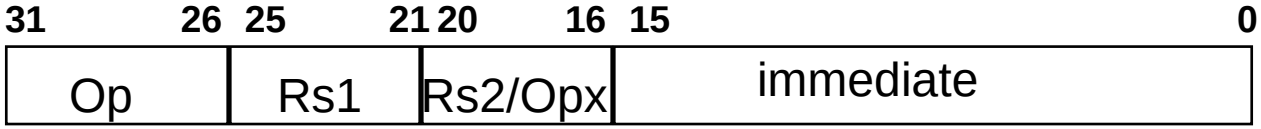
Register-Register



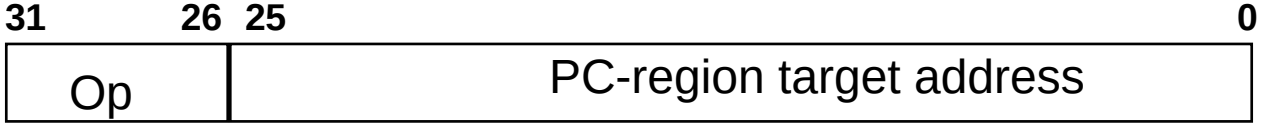
Register-Immediate (e.g., load/store)



Branch



Jump / Call



ISA Metrics

- Orthogonality
  - No special registers, few special cases, all operand modes available with any data type or instruction type
- Completeness
  - Support for a wide range of operations and target applications
- Regularity
  - No overloading for the meanings of instruction fields
- Streamlined
  - Resource needs easily determined
- **Ease of compilation (or assembly language programming)**
- **Ease of implementation**

Closer look at ISA Aspects

- Operand location
- Addressing modes
- Types of instructions

ISA Aspect (1): Operand Location

Accumulator (before 1960):

1 address	add A	acc ← acc + mem[A]

Stack (1960s to 1970s):

0 address	add	tos ← tos + next

Memory-Memory (1970s to 1980s):

2 address	add A, B	mem[A] ← mem[A] + mem[B]
3 address	add A, B, C	mem[A] ← mem[B] + mem[C]

Register-Memory (1970s to present):

2 address	add R1, A	R1 ← R1 + mem[A]
	load R1, A	R1 ← mem[A]

Register-Register, also called Load/Store (1960s to present):

3 address	add R1, R2, R3	R1 ← R2 + R3
	load R1, R2	R1 ← mem[R2]
	store R1, R2	mem[R1] ← R2

Choices for Operand Location

- Running example: C:= A + B
- Accumulator

```
load A      accum = M[A];
add B       accum += M[B];
store C     M[C] = accum;
```

  + Less hardware, code density
  – Memory bottleneck

- Stack

```
push A      S[++tos] = M[A];
push B      S[++tos] = M[B]
add         t1= S[tos--]; t2= S[tos--]; S[++tos]= t1 + t2;
pop C       M[C] = S[tos--];
```

  + Less hardware, code density
  – Memory, pipelining bottlenecks
  – x86 uses stack model for floating point computations

Choices for Operand Location (cont'd)

- Running example: C:= A + B
- Memory-Memory

```
add C, A, B          M[C] = M[A] + M[B];
```

  + Code density (most compact)
  – Memory bottleneck
  – No current machines support memory-memory (VAX did)

- Memory-Register

```
load R1, A           R1 = M[A];
add R1, B            R1 += M[B];
store C, R1          M[C] = R1;
```

  + Like several explicit (extended) accumulators
  + Code density, easy to decode
  – Asymmetric operands, different amount of work per instruction
  – Examples: IBM 360/370, x86, Motorola 68K

Choices for Operand Location (cont'd)

- Running example: C:= A + B
- Register-Register (Load-Store)

```
load R1, A          R1 = M[A];
load R2, B          R2 = M[B];
add R3, R1, R2      R3 = R1 + R2;
store C, R3         M[C] = R3;
```

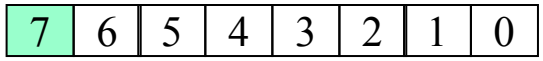
  + Easy decoding, operand symmetry
  + Deterministic cost for ALU operations (simple cost model)
  + Scheduling opportunities
  – Code density

Operand Location: Registers vs. Memory

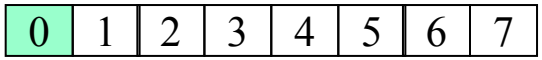
- Pros and cons of registers
  - + Faster, direct access
  - + Simple cost model (fixed latency, no misses)
  - + Short identifier
  - – Must save/restore on procedure calls, context switches
  - – Fixed size (larger-sized structures must live in memory)
- Pros and cons of more registers
  - + Possible to keep more operands for longer in faster memory
    - Shorter operand access time, lower memory traffic
  - – Longer specifiers
  - – Larger cost for saving CPU state
  - – Trend towards more registers
    - 8 (x86) -> 32 (MIPS/Alpha/PPC) -> 128 (IA-64)
    - Driven by increasing compiler involvement in scheduling

ISA Aspect (2): Addressing

- Endian-ness: Order of bytes in words
  - Big: byte at lowest address has the most significance (big end)
    - E.g., IBM, Sun SPARC



  - Little: bytes at lower address have lower significance (little end)
    - E.g., x86



  - Some processors allow mode to be selectable
    - E.g., PowerPC, MIPS (new implementations of the ISA)

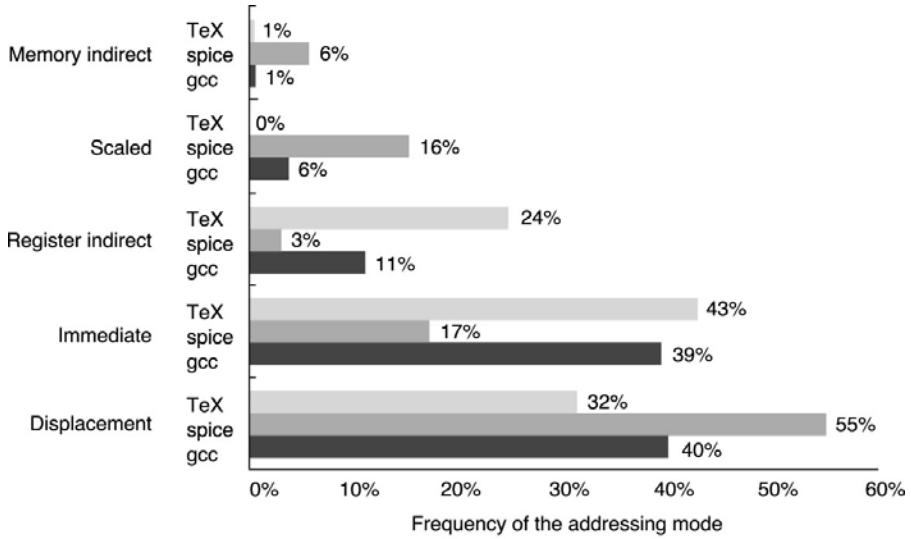
- Alignment: Natural boundaries defined by architecture
  - Aligned address: (address % size) equals 0
- Different ISAs support different restrictions on alignment
  - None (all alignments supported by hardware): expensive/exception handling
  - Restricted: misaligned access traps to software
  - Middle ground: misaligned data okay, but requires multiple instructions
    - E.g., MIPS: `lwl/lwr` (Load Word Left/Right: only modify part of register)

Types of Addressing Modes (VAX)

Mode	*Example*	*Action*
1. Register direct	`add R4, R3`	`R4 ← R4 + R3`
2. Immediate	`add R4, #3`	`R4 ← R4 + 3`
3. Displacement	`add R4, 100(R1)`	`R4 ← R4 + M[100+R1]`
4. Register indirect	`add R4, (R1)`	`R4 ← R4 + M[R1]`
5. Indexed	`add R4, (R1 + R2)`	`R4 ← R4 + M[R1 + R2]`
6. Direct	`add R4, (1000)`	`R4 ← R4 + M[1000]`
7. Memory Indirect	`add R4, @(R3)`	`R4 ← R4 + M[M[R3]]`
8. Autoincrement	`add R4, (R2)+`	`R4 ← R4 + M[R2]` `R2 ← R2 + d`
9. Autodecrement	`add R4, (R2)-`	`R4 ← R4 + M[R2]` `R2 ← R2 - d`
10. Scaled	`add R4, 100(R2)[R3]`	`R4 ← R4 +` `M[100 + R2 + R3*d]`

Which modes are actually used?

- Study by Clark and Emer
  - Modes 1–4 account for 93% of all VAX operands
- Register mode responsible for roughly half
- Memory modes



- An example of making the common case fast
  - RISC machines typically implement register, immediate, and displacement
  - Synthesize all other modes in software

Addressing Modes for Signal Processing

Two additional modes, motivated by DSP applications

- Modulo or circular addressing
  - DSPs deal with large (infinite, continuous) streams of data
  - Typically encoded as a circular buffer (mode allows auto-reset)
  - Keeps start and end registers with each address register
  - Allows autoincrement/autodecrement modes to reset the address
- Bit reverse addressing
  - Permit permutations on address (to support kernels such as FFT)
  - E.g., address/displ. (100) results in access to address/displ. (001)
- Used in assembly implementations of some libraries

- Renewed importance of autoincrement and autodecrement modes
  - Study on TI TMS320C54x DSP finds autoincrement mode use of ~18.8%

ISA Aspect (3): Types of Operations

• Arithmetic and Logic:	AND, ADD
• Data Transfer:	MOVE, LOAD, STORE
• Control	BRANCH, JUMP, CALL
• System	OS CALL, VM
• Floating Point	ADDF, MULF, DIVF
• Decimal	ADDD, CONVERT
• String	MOVE, COMPARE
• Graphics	(DE)COMPRESS

Relative Frequency of Instructions

- For the 80x86, averaged over five SPECint92 programs

Rank	*Instruction*	*Frequency*
1	load	22%
2	branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	register move	4%
9	call	1%
10	return	1%
Total		96%

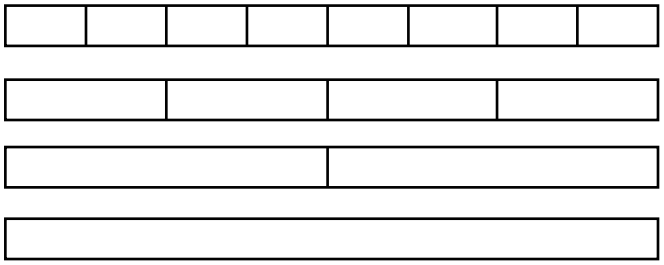
- Simple instructions dominate

Operations for Media and Signal Processing

- Results of media processing gauged in terms of human perception
  - Narrow data items (8-16 bits) as opposed to 32 or 64-bit words
  - Lower precision requirements
  - Real-time requirements → cannot cause overflow traps
- Several ISAs have been extended to support graphics and multimedia
  - Intel: MMX, Streaming SIMD Extensions (SSE, SSE2, and SSE3)
  - AMD: 3DNow!
  - Sun: Visual Instruction Set (VIS)
  - Motorola and IBM: AltiVec
- Common theme
  - Partitioned operations (SIMD), pack/unpack operations
  - Saturating arithmetic
  - Multiply-accumulate (MAC) instructions (dot products and vector multiplies)

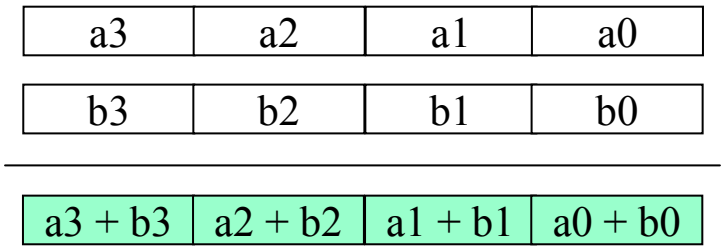
MMX

- Packed data types
  - Packed byte
  - Packed word
  - Double word
  - Quad word



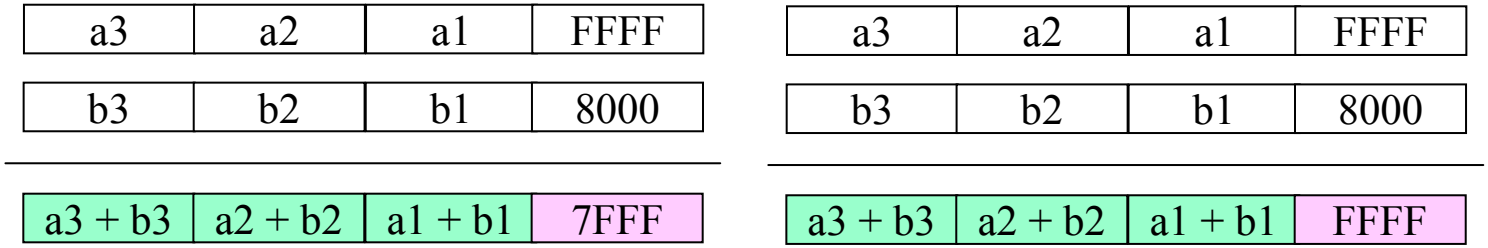
  - Stored in 64-bit FP registers
    - + MMX does not add any new state to the processor
    - – MMX and FP don't work well together

– SIMD instructions work on the packed types (single cycle)
  - Minimal hardware to isolate sub-operations (5% increase in chip area)
  - E.g., `PADD[W]`



MMX (cont'd)

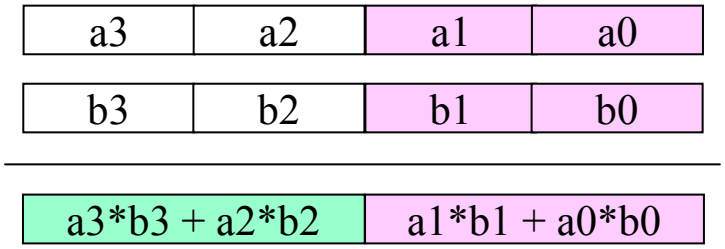
- Saturating arithmetic
  - Both wrap-around and saturating adds are supported
  - Saturating mode: results that overflow are set to the largest value



`PADD[W]`: Packed wrap-around add

`PADDUS[W]`: Packed saturating add

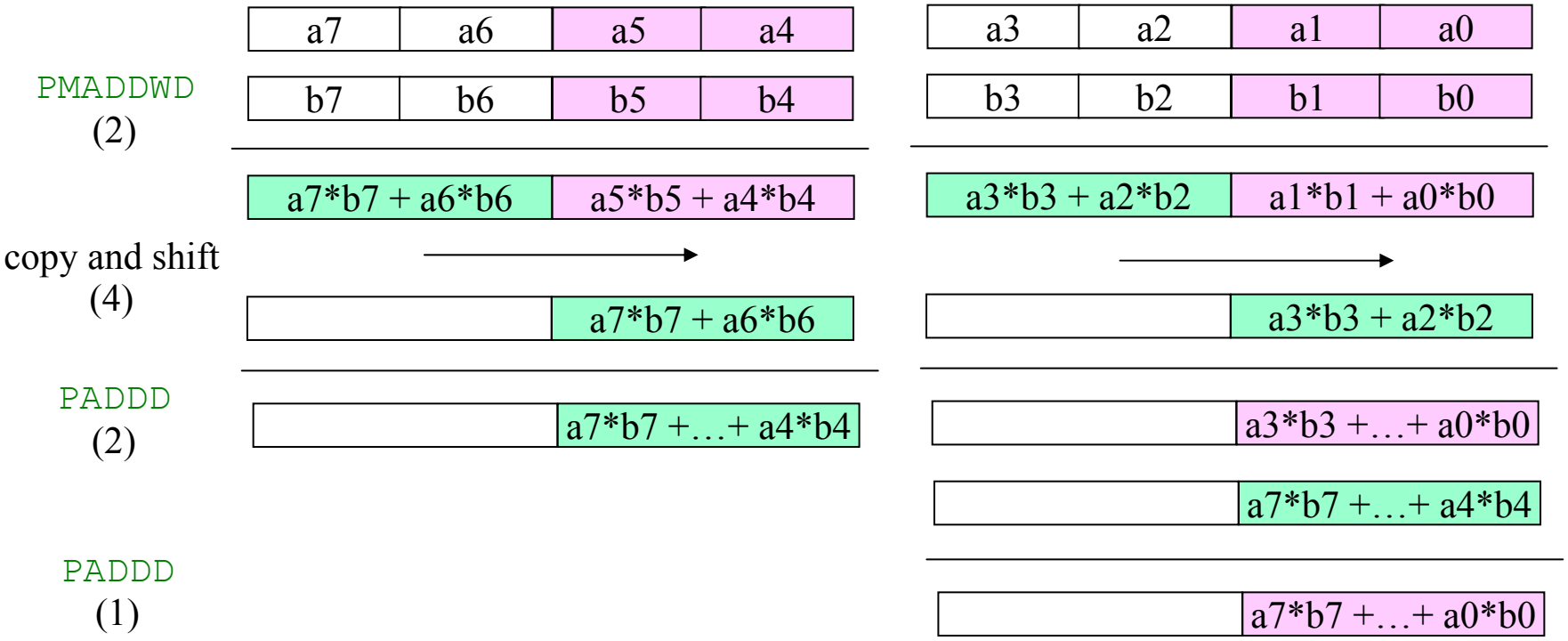
- Multiply-accumulate (MAC) operations
  - E.g., `PMADDWD`: Packed multiply-add word to double



MMX: Example

- Vector dot product on an 8-element vector
  - 9 MMX instructions (as compared to 40 without MMX)

$$x = \sum a(i) \times b(i)$$



Performance Impact of MMX

- Pentium processors (200 MHz, 512 KB L2) with and without MMX
  ("MMX Technology Architecture Overview" – Mittal, Peleg, Weiser
  Intel Technology Journal, 3Q 1997)

Application	*Without MMX*	*With MMX*	*Speedup*
Video	155.52	268.70	1.72
Image Processing	159.03	743.90	4.67
3D geometry	161.52	166.44	1.03
Audio	149.80	318.90	2.13
Overall	156.00	255.43	1.64

- More recent Intel processors (Pentium III and 4) incorporate additional instructions: Streaming SIMD Extensions (SSE, SSE2, and SSE3)
  - New 128-bit registers, data prefetching
  - SIMD floating point operations
  - Performance gain of 1.5 to 2.0 on video and 3D applications

Control Instructions

- Instructions that change the value of the PC
- Three kinds of instructions
  - (conditional) branches, (unconditional) jumps
  - Function calls, function returns
  - System calls, system returns
- Questions
  - What kinds of conditions are supported? How are the conditions set?
  - How is the target address specified?
  - For function and system calls, how is the return address specified?
    - Most recent processors use an implicit register
      - + Simple scheme
      - – Software needs to save/restore register contents
  - Which instructions save/restore CPU state?

Control Instructions (1): Conditions

- Compare and branch
  - \+ Single instruction branches
  - – Needs ALU stage for branch instructions as well; May be too much work for pipelined execution
- Condition codes (e.g., zero, negative, overflow)

  Condition code set by ALU operations
  - \+ Sometimes set for free
  - – Part of the CPU state, needs to be saved and restored
  - – Constrains the ordering of instructions

  "compare" instruction can be used

- Recent processors (e.g., Alpha, IA-64) offer predicated instructions
  - – combine comparison, branch, and ALU operations
  - – more about these later in the course

Control Instructions (2): Target Address

Four choices

- PC-relative with immediate
  - \+ Position independent, all info present for computing target
  - \+ Short immediate sufficient (compact instructions)
  - – Target must be known statically

  Uses: branches/jumps within function
- Arbitrary immediate

  Uses: function calls
- Register
  - \+ Short specifier, target can be dynamic
  - – Extra instruction to load register

  Uses: indirect calls (e.g., DLLs, virtual functions), returns, switches
- Vectored traps
  - \+ Protection (hence, heavyweight)

  Uses: system calls

Control Instructions (3): Save and Restore State

- Only call (function and system) instructions need to save state
  - Function calls: save registers
  - System calls: save registers, flags, PC, PSW, …

Two choices

- Software saving/restoring
  - Calling convention distinguishes between caller- and callee-save registers
- Hardware saving/restoring
  - Explicit instructions: VAX
  - Implicit: SPARC register windows
    - 32 registers: 8 input, 8 output, 8 local, 8 global
    - On call: 8 output of caller become 8 input of callee
      - local/output registers are new set (no need to save/restore)
    + No save/restore on shallow call graphs
    - Makes advanced architectural techniques (e.g., register renaming) hard

The RISC vs. CISC Debate

- Early 80s: Several projects challenged how processors were being built
  - Berkeley RISC-I (Patterson), Stanford MIPS (Hennessy), IBM 801
  - Contrasted their design, RISC (Reduced Instruction Set Computer) with what began to be called CISC (Complex Instruction Set Computer)
- RISC argument
  - CISC is too complex to ever be implemented well
    - too many addressing modes, variable format instructions, many multi-cycle operations, microcoding, hand-assembled programs
  - RISC characterized by
    - Single-cycle operation
    - Hardwired control
    - Load/store organization
    - Fixed instruction format
    - Few modes
    - Reliance on compiler optimization

Motivated by quantitative studies of program behavior

Focus on optimizing the common case

The Reality of RISC vs. CISC

- RISC does help compiler optimizations
  - Load/store architecture supports register allocation
    - Explicit choices of what values reside where in the memory hierarchy
  - Simple instructions make instruction selection, optimization easier

- However, CISC does not have any fundamental implementation flaws
  - Fixable with more transistors
    - Good CISC pipeline can be constructed with modest increase in transistors
    - Not an issue given Moore's law

- Most commercially successful ISA is x86 (CISC)
  - Current-day Pentium processors translate CISC instructions into sequences of RISC micro-ops
    - Internal microarchitecture is actually RISC
  - Better substrate for implementing advanced architectural techniques

Announcements

- Assignment 0 out today; suggested due date: By next class (Feb. 1st)

Pipelining

[Hennessy/Patterson CA:AQA (3rd Edition): Chapter 2, Appendix A]

Computer Pipelines

- Computers execute billions of instructions, so instruction throughput is what matters

Main idea behind pipelining

- Divide instruction execution across several stages
  - each stage accesses only a subset of the CPU's resources
- Example: Classic 5-stage RISC pipeline

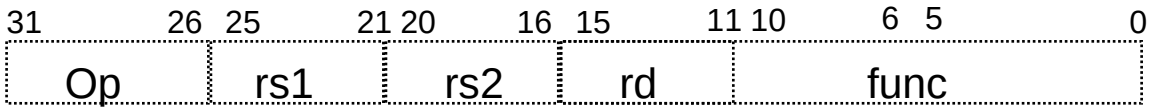
  IF ⟶ ID ⟶ EX ⟶ MEM ⟶ WB
- Simultaneously have different instructions in different stages
  - Ideally, can issue a new instruction every cycle
  - Cycle time determined by longest stage

- Pipelining only improves throughput, not latency
  - Each instruction still needs to go through each stage

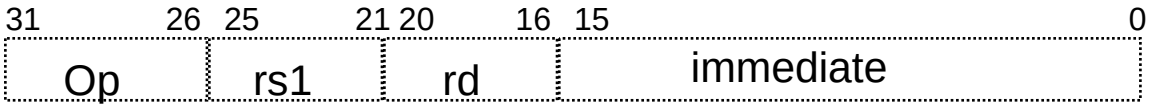
Classic 5-Stage RISC Pipeline

- Some key properties of RISC architectures simplify implementation
  - all instructions same length
  - registers located in same place in instruction format
  - memory operands only in loads or stores
- E.g., MIPS

Register-Register (R-type) ADD R1, R2, R3



Register-Immediate (I-type) SUB R1, R2, #3

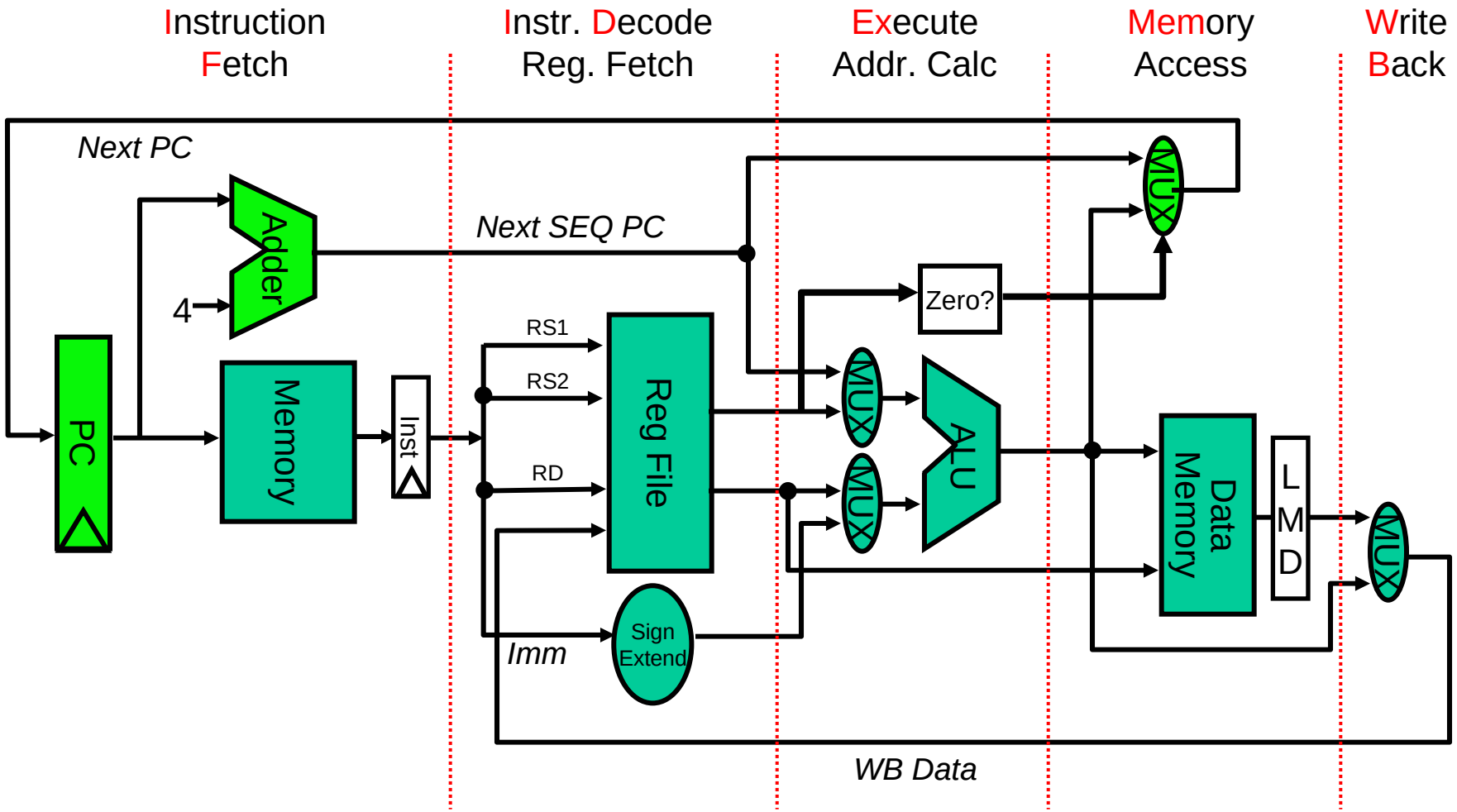


Jump / Call (J-type) JUMP end

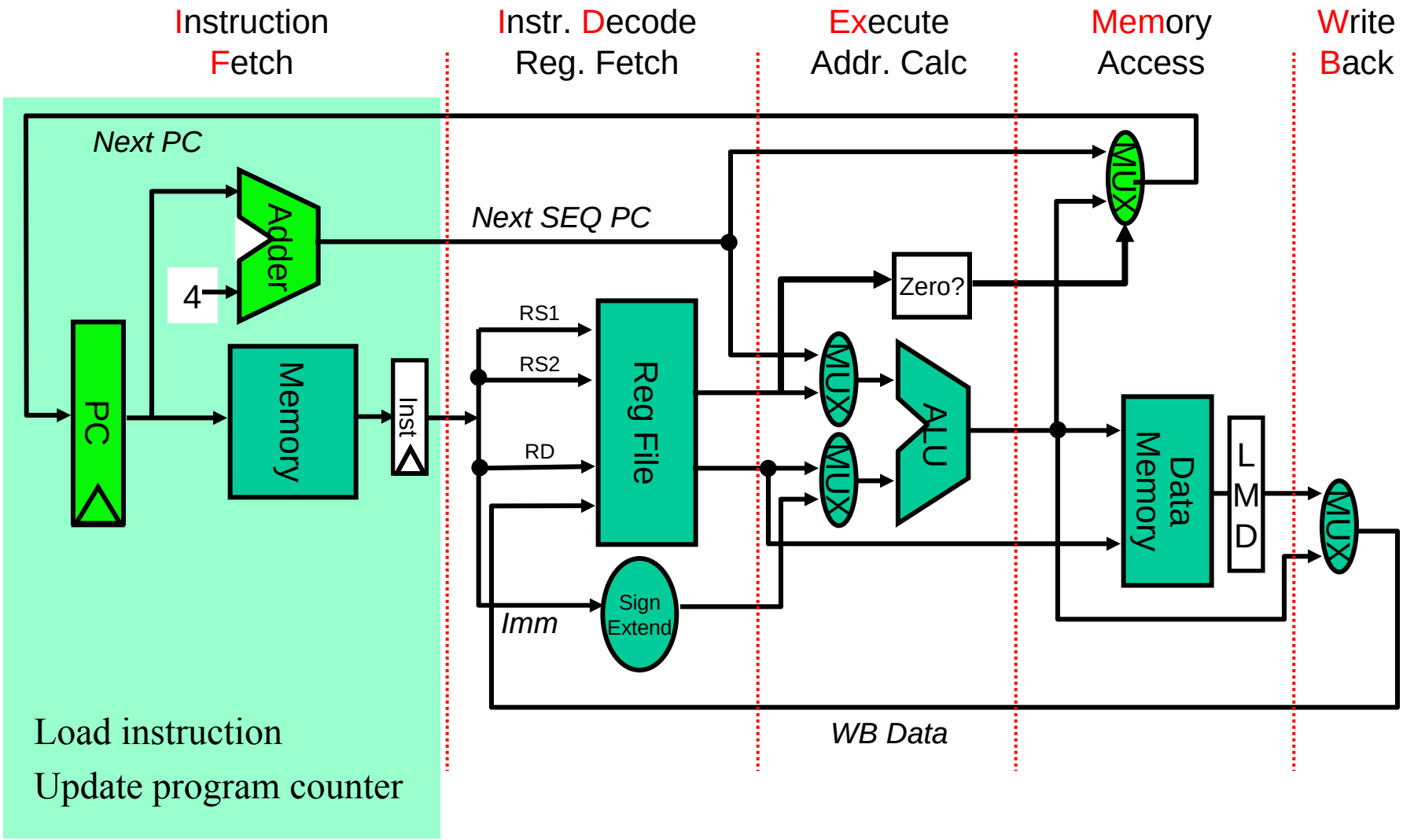
(jump, jump and link, trap and return from exception)

Unpipelined Implementation of a RISC ISA

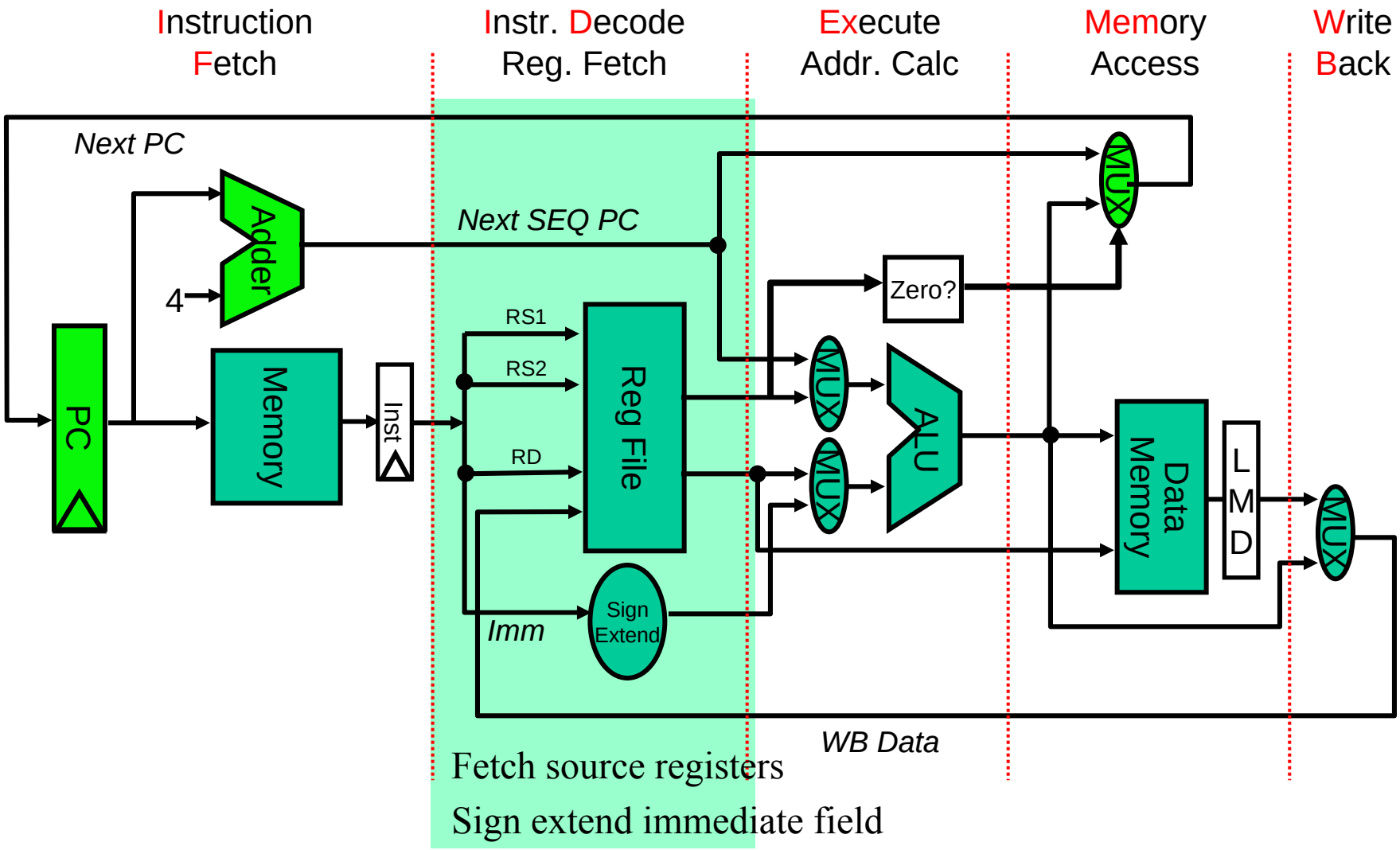
- Most MIPS instructions can be implemented in 5 cycles



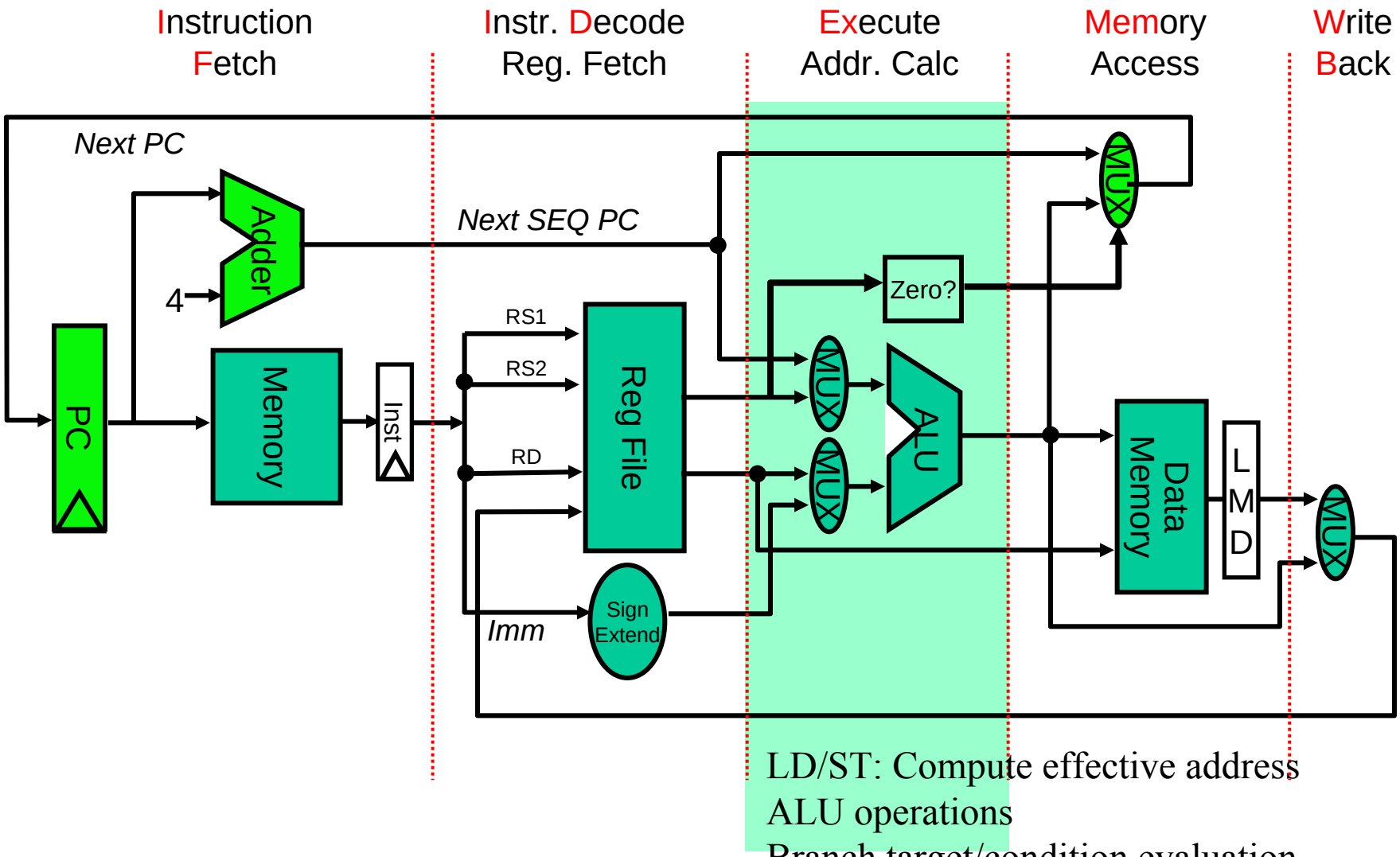
Unpipelined Implementation (1) – Instruction Fetch



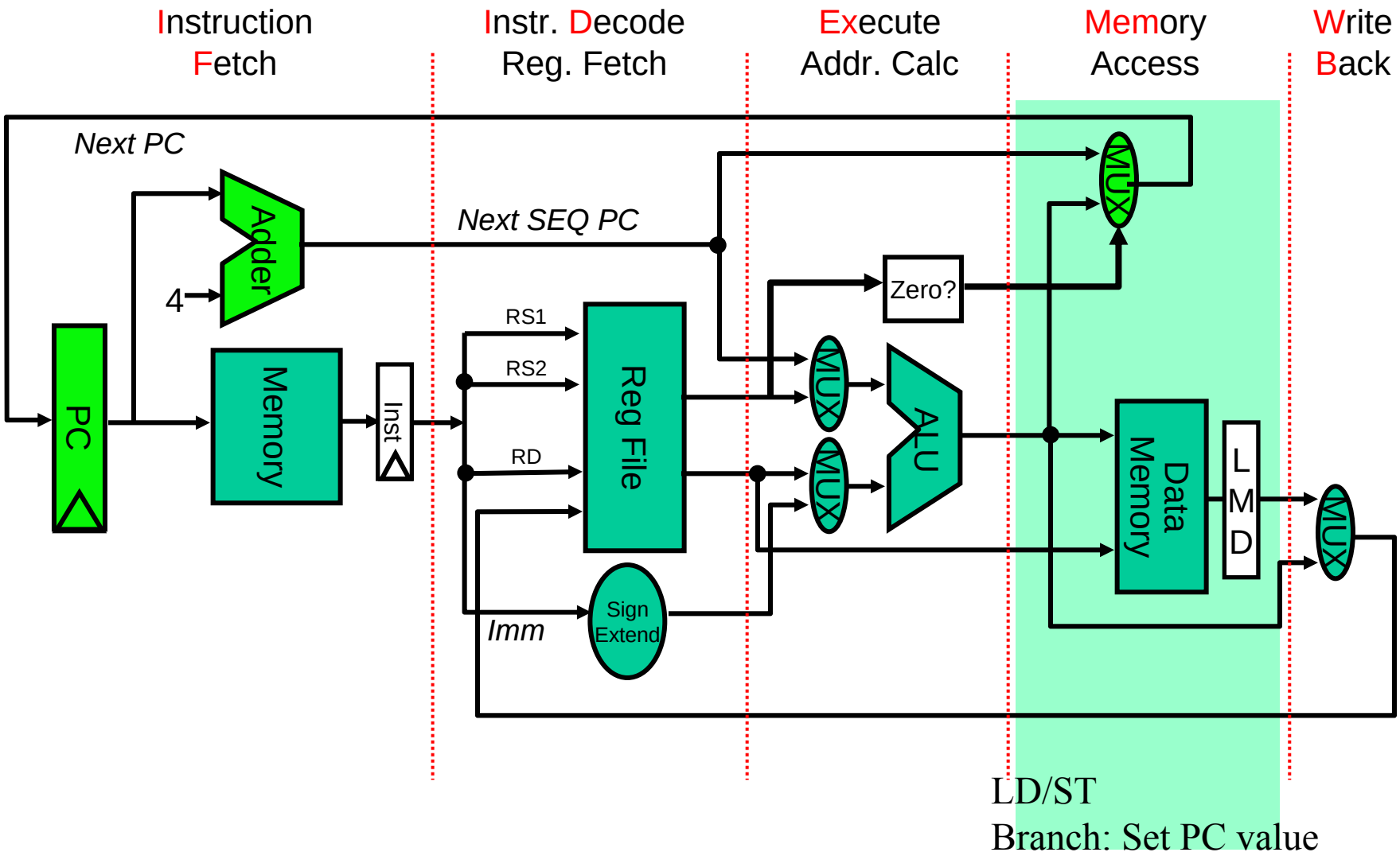
Unpipelined Implementation (2) – Instruction Decode



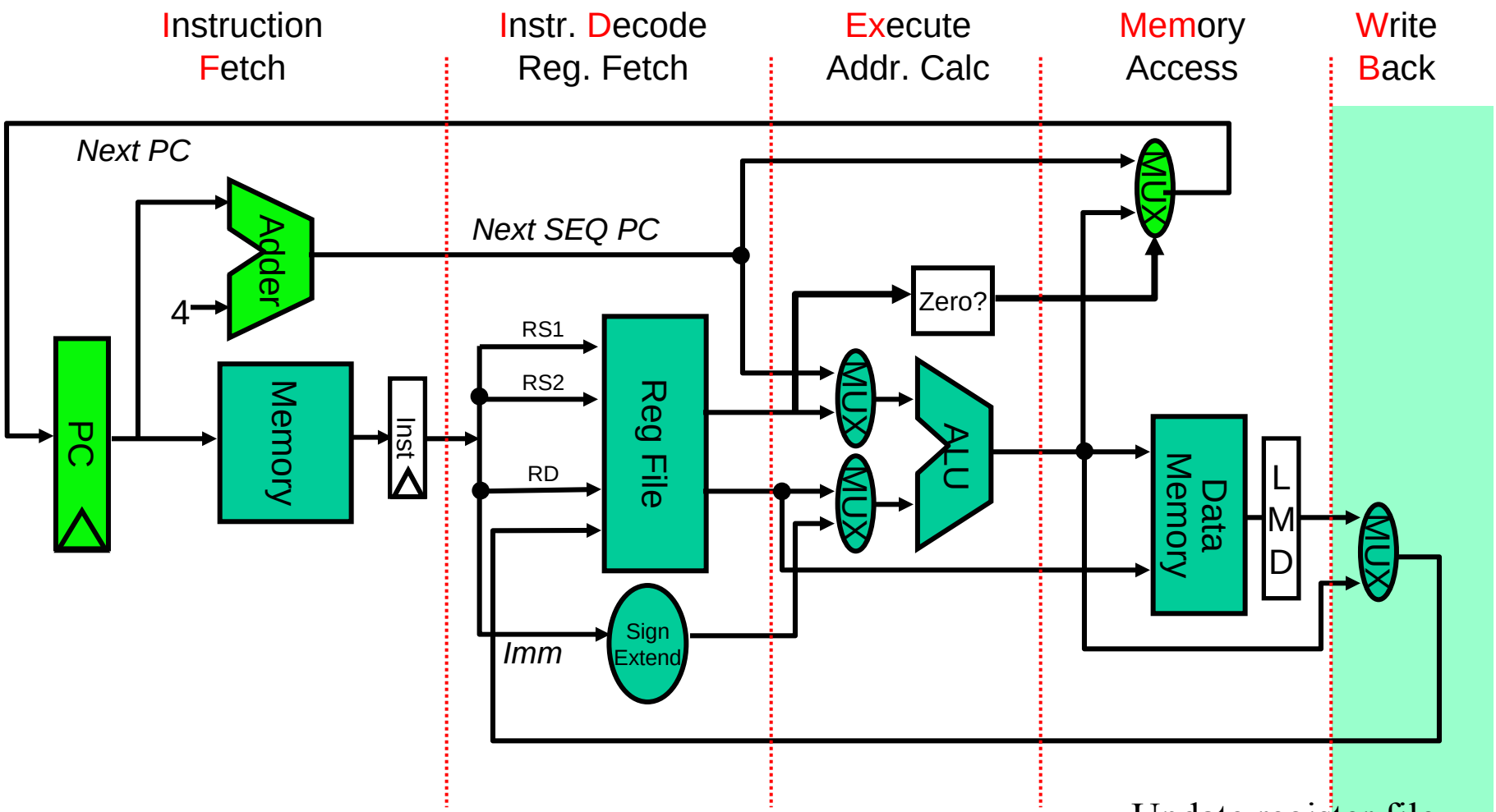
Unpipelined Implementation (3) – Execute



Unpipelined Implementation (4) – Memory Access



Unpipelined Implementation (4) – Write Back



CPI for the Multiple-Cycle RISC Implementation

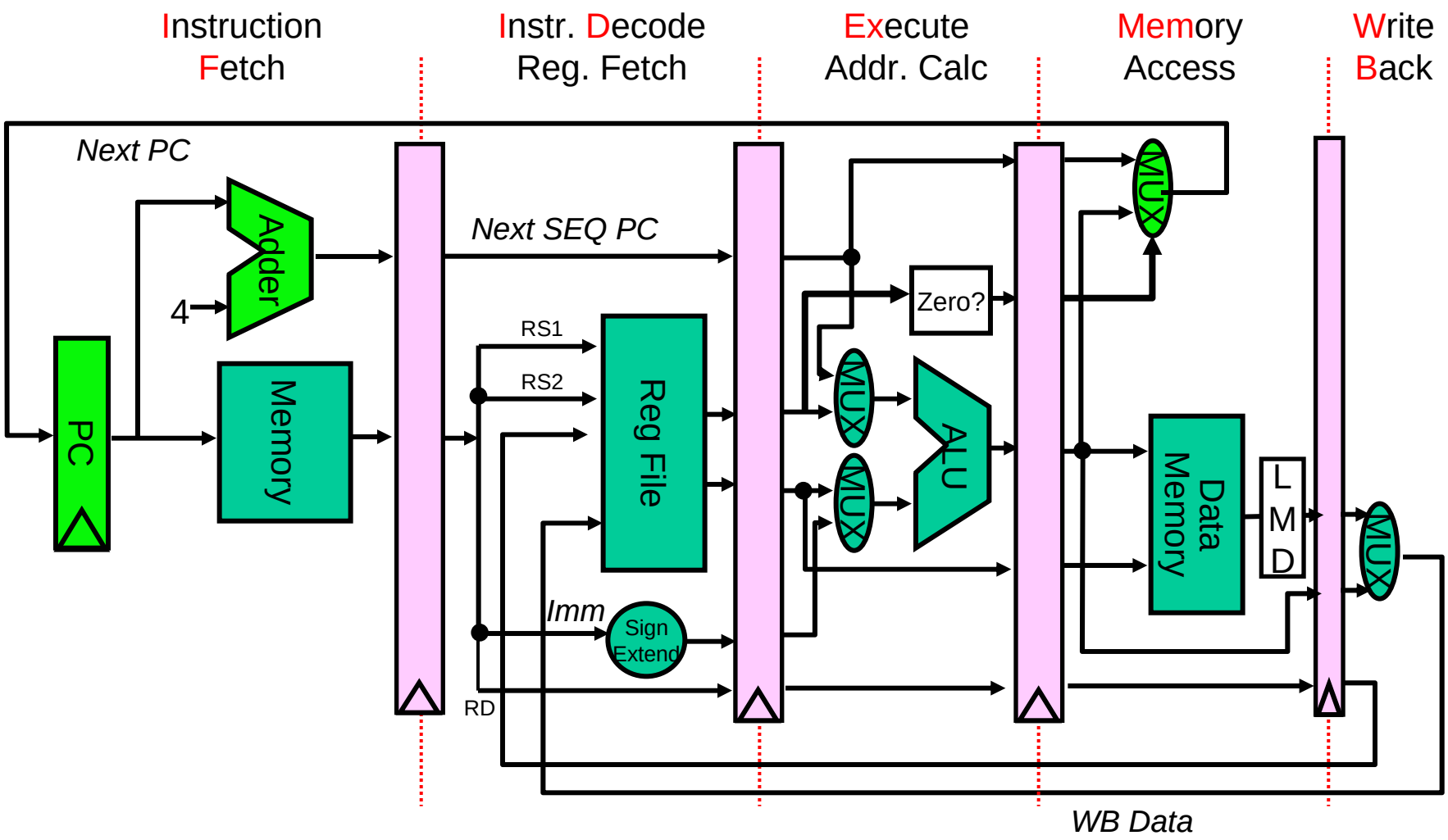
- Branches and stores: 4 cycles (no WB), all other instructions: 5 cycles
  - If 20% of the instructions are branches or loads

    CPI = 0.8*5 + 0.2*4 = 4.80
- ALU operations can be allowed to complete in 4 cycles (no MEM)
  - If 40% of the instructions are ALU operations

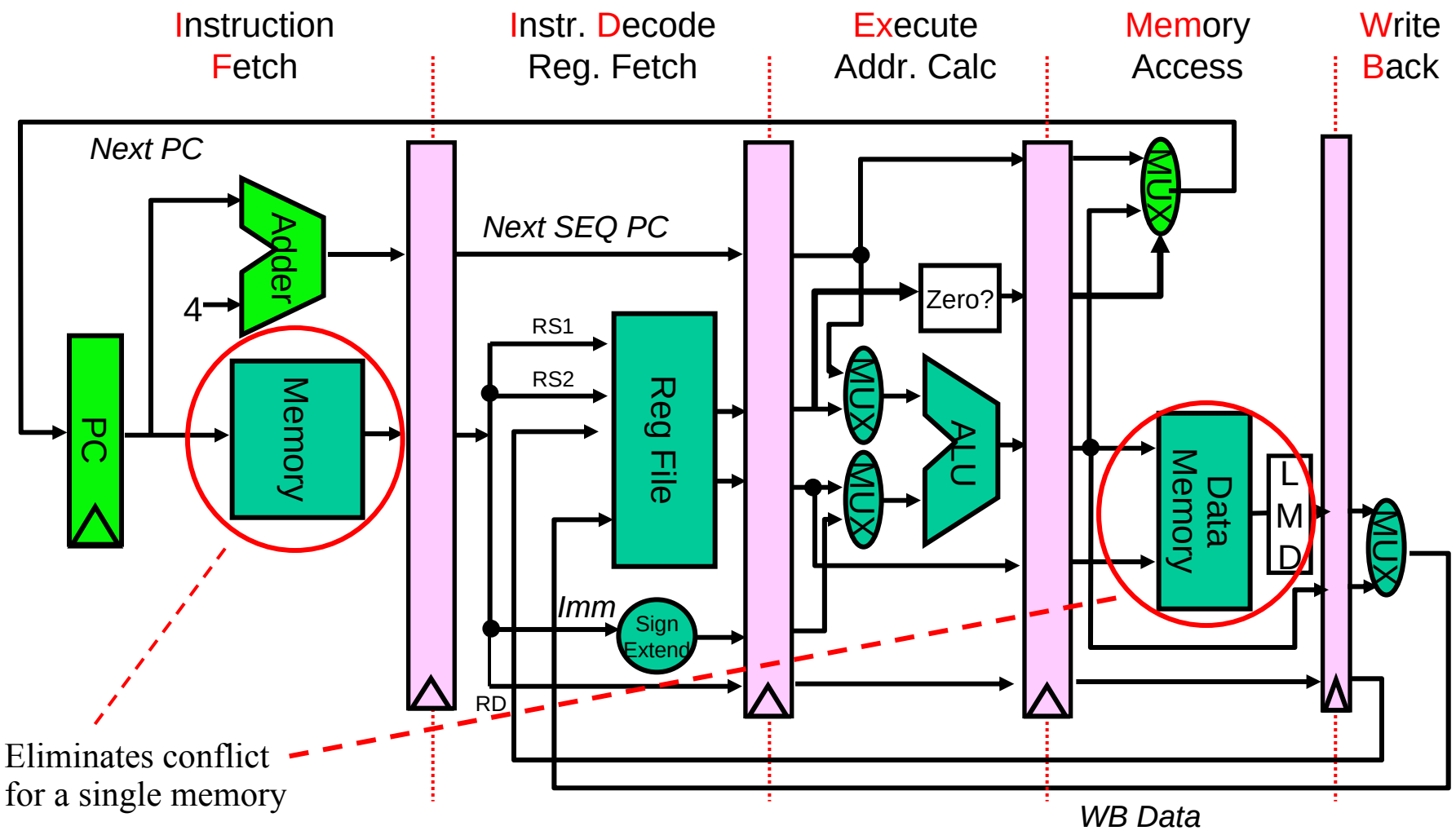
    CPI = 0.4*5 + 0.6*4 = 4.40
- Pipelining the implementation can help reduce the CPI

Instr.	*Clock Number*								
	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
i+1		IF	ID	EX	MEM	WB			
i+2			IF	ID	EX	MEM	WB		
i+3				IF	ID	EX	MEM	WB	
i+4					IF	ID	EX	MEM	WB

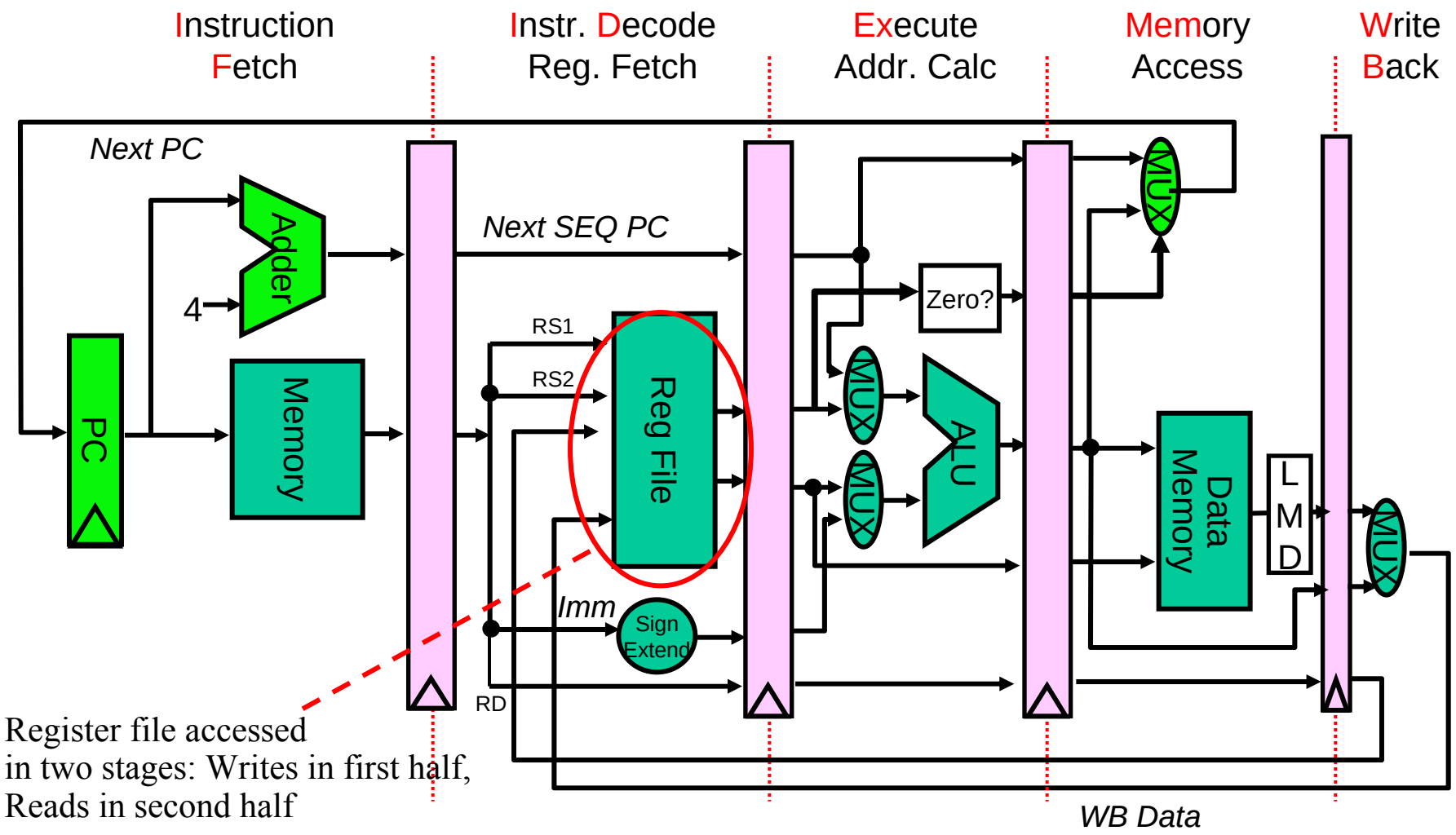
Pipelined Implementation of a RISC ISA



Pipelined Implementation of a RISC ISA (1): Separation of Instruction and Data Memories

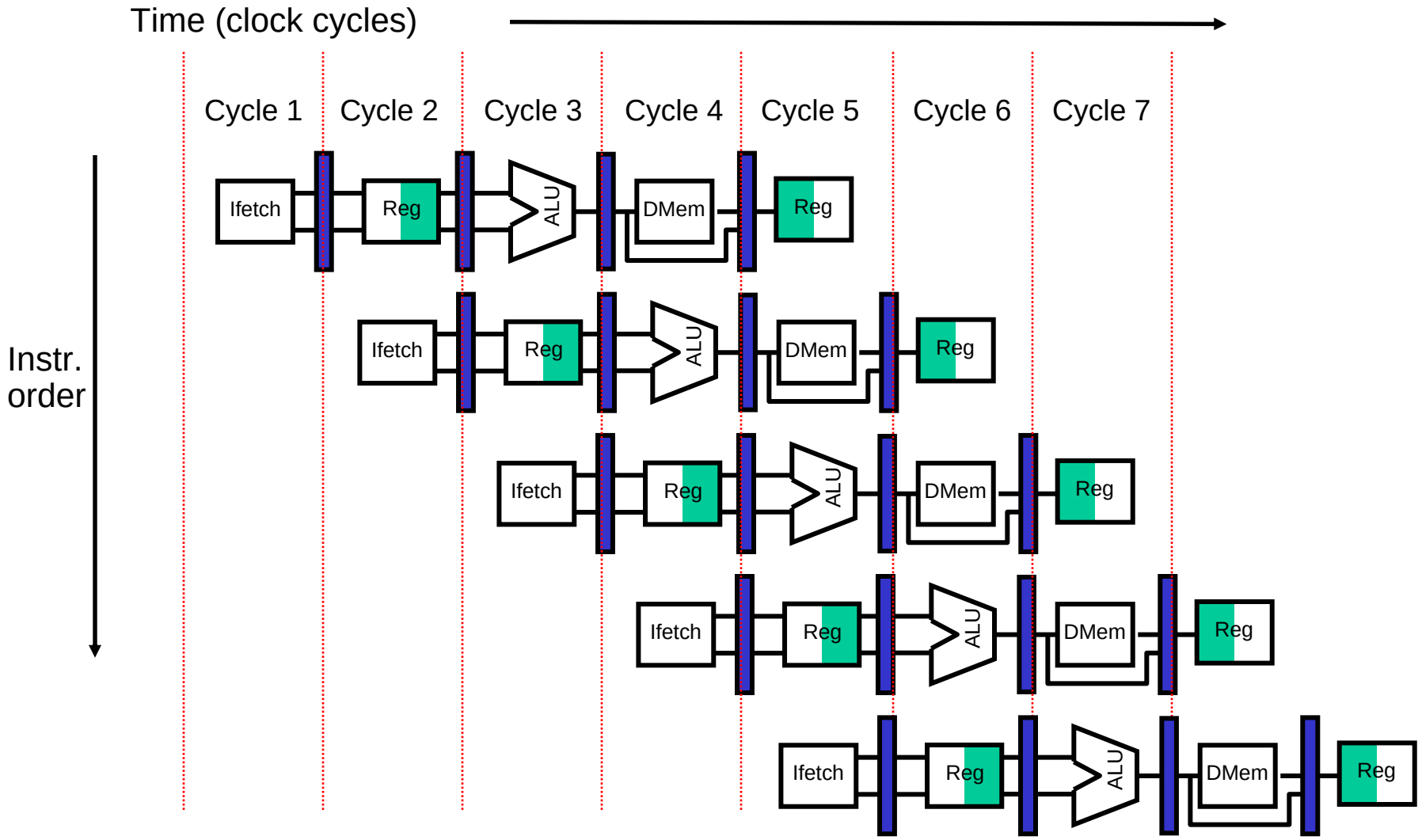


Pipelined Implementation of a RISC ISA (2): Split-phase Register Access



Register file accessed in two stages: Writes in first half, Reads in second half

Visualizing Pipelining



Speedup from Pipelining

- Assume that a multiple cycle RISC implementation has a 10 ns clock cycle, loads take 5 clock cycles and account for 40% of the instructions, and all other instructions take 4 clock cycles.
- If pipelining the machine adds 1 ns to the clock cycle, how much speedup in instruction execution rate do we get from pipelining?

MC Ave. Instr. Time = Clock cycle x Average CPI

= 10 ns x (0.6 x 4 + 0.4 x 5)

= 44 ns

PL Ave. Instr. Time = 10 + 1 = 11 ns

Speedup = 44 / 11 = 4

- The above expression assumes a pipelining CPI of 1
  Should we expect this in practice? Any complications?