

18-447

Computer Architecture

Lecture 4: ISA Tradeoffs (Continued)

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2014, 1/22/2014

X86: Small Semantic Gap: String Operations

REP MOVS (DEST SRC)

```
IF AddressSize = 16
  THEN
    Use CX for CountReg;
  ELSE IF AddressSize = 64 and REX.W used
    THEN Use RCX for CountReg; FI;
  ELSE
    Use ECX for CountReg;
  FI;
WHILE CountReg ≠ 0
  DO
    Service pending interrupts (if any);
    Execute associated string instruction;
    CountReg ← (CountReg - 1);
    IF CountReg = 0
      THEN exit WHILE loop; FI;
    IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
      or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
      THEN exit WHILE loop; FI;
  OD;
```

```
DEST ← SRC;
IF (Byte move)
  THEN IF DF = 0
    THEN
      (R)E SI ← (R)E SI + 1;
      (R)E DI ← (R)E DI + 1;
    ELSE
      (R)E SI ← (R)E SI - 1;
      (R)E DI ← (R)E DI - 1;
    FI;
  ELSE IF (Word move)
    THEN IF DF = 0
      (R)E SI ← (R)E SI + 2;
      (R)E DI ← (R)E DI + 2;
      FI;
    ELSE
      (R)E SI ← (R)E SI - 2;
      (R)E DI ← (R)E DI - 2;
    FI;
  ELSE IF (Doubleword move)
    THEN IF DF = 0
      (R)E SI ← (R)E SI + 4;
      (R)E DI ← (R)E DI + 4;
      FI;
    ELSE
      (R)E SI ← (R)E SI - 4;
      (R)E DI ← (R)E DI - 4;
    FI;
  ELSE IF (Quadword move)
    THEN IF DF = 0
      (R)E SI ← (R)E SI + 8;
      (R)E DI ← (R)E DI + 8;
      FI;
    ELSE
      (R)E SI ← (R)E SI - 8;
      (R)E DI ← (R)E DI - 8;
    FI;
  FI;
```

FI;

How many instructions does this take in ARM and MIPS?

Small Semantic Gap Examples in VAX

- FIND FIRST
 - Find the first set bit in a bit field
 - Helps OS resource allocation operations
- SAVE CONTEXT, LOAD CONTEXT
 - Special context switching instructions
- INSQUEUE, REMQUEUE
 - Operations on doubly linked list
- INDEX
 - Array access with bounds checking
- STRING Operations
 - Compare strings, find substrings, ...
- Cyclic Redundancy Check Instruction
- EDITPC
 - Implements editing functions to display fixed format output
- Digital Equipment Corp., “[VAX11 780 Architecture Handbook](#),” 1977-78.

Small versus Large Semantic Gap

- CISC vs. RISC
 - Complex instruction set computer → complex instructions
 - Initially motivated by “not good enough” code generation
 - Reduced instruction set computer → simple instructions
 - John Cocke, mid 1970s, IBM 801
 - Goal: enable better compiler control and optimization

- RISC motivated by
 - Memory stalls (no work done in a complex instruction when there is a memory *stall*?)
 - When is this correct?
 - Simplifying the hardware → lower cost, higher frequency
 - Enabling the compiler to optimize the code better
 - Find fine-grained parallelism to reduce *stalls*

How High or Low Can You Go?

- **Very large semantic gap**

- Each instruction specifies the complete set of control signals in the machine
- Compiler generates control signals
- Open microcode (John Cocke, circa 1970s)
 - Gave way to optimizing compilers

- **Very small semantic gap**

- ISA is (almost) the same as high-level language
- Java machines, LISP machines, object-oriented machines, capability-based machines

A Note on ISA Evolution

- ISAs have evolved to reflect/satisfy the concerns of the day
- Examples:
 - Limited on-chip and off-chip memory size
 - Limited compiler optimization technology
 - Limited memory bandwidth
 - Need for specialization in important applications (e.g., MMX)
- Use of translation (in HW and SW) enabled underlying implementations to be similar, regardless of the ISA
 - Concept of dynamic/static interface
 - Contrast it with hardware/software interface

Effect of Translation

- One can translate from one ISA to another *ISA* to change the semantic gap tradeoffs
- Examples
 - Intel's and AMD's x86 implementations translate x86 instructions into programmer-invisible microoperations (simple instructions) in hardware
 - Transmeta's x86 implementations translated x86 instructions into "secret" VLIW instructions in software (code morphing software)
- Think about the tradeoffs

ISA-level Tradeoffs: Instruction Length

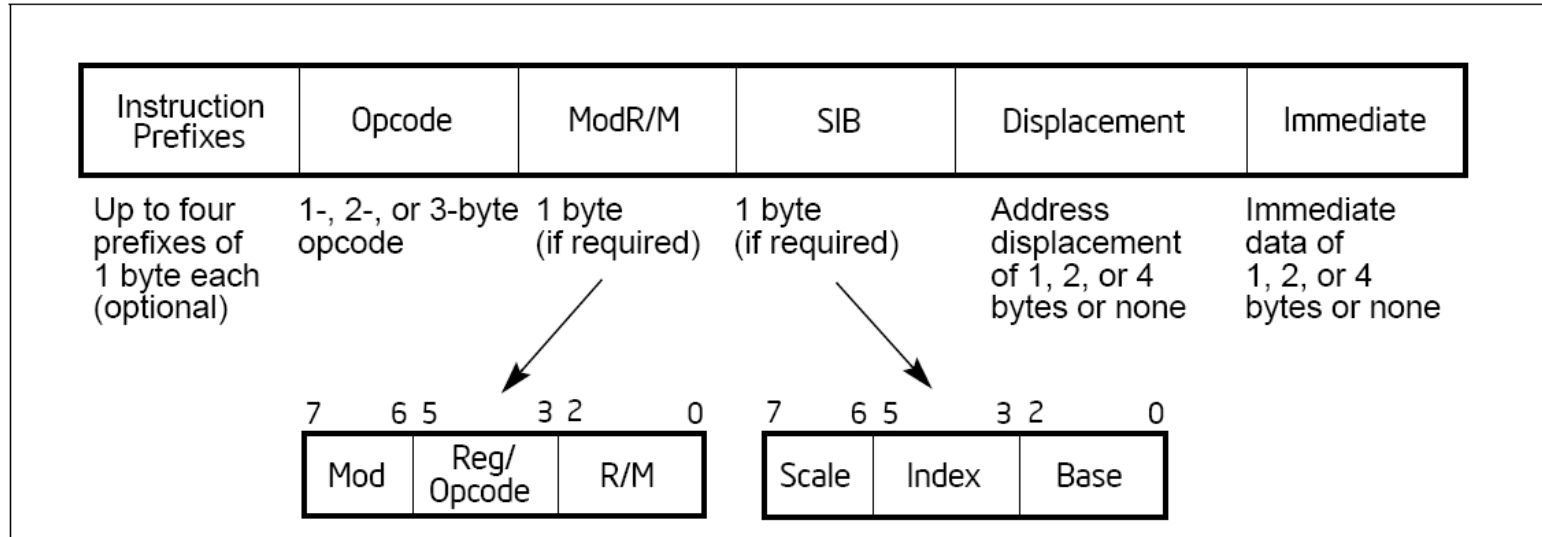
- **Fixed length:** Length of all instructions the same
 - + Easier to decode single instruction in hardware
 - + Easier to decode multiple instructions concurrently
 - Wasted bits in instructions (**Why is this bad?**)
 - Harder-to-extend ISA (how to add new instructions?)
- **Variable length:** Length of instructions different (determined by opcode and sub-opcode)
 - + Compact encoding (**Why is this good?**)
 - Intel 432: Huffman encoding (sort of). 6 to 321 bit instructions. **How?**
 - More logic to decode a single instruction
 - Harder to decode multiple instructions concurrently
- **Tradeoffs**
 - Code size (memory space, bandwidth, latency) vs. hardware complexity
 - ISA extensibility and expressiveness
 - Performance? Smaller code vs. difficult decode

ISA-level Tradeoffs: Uniform Decode

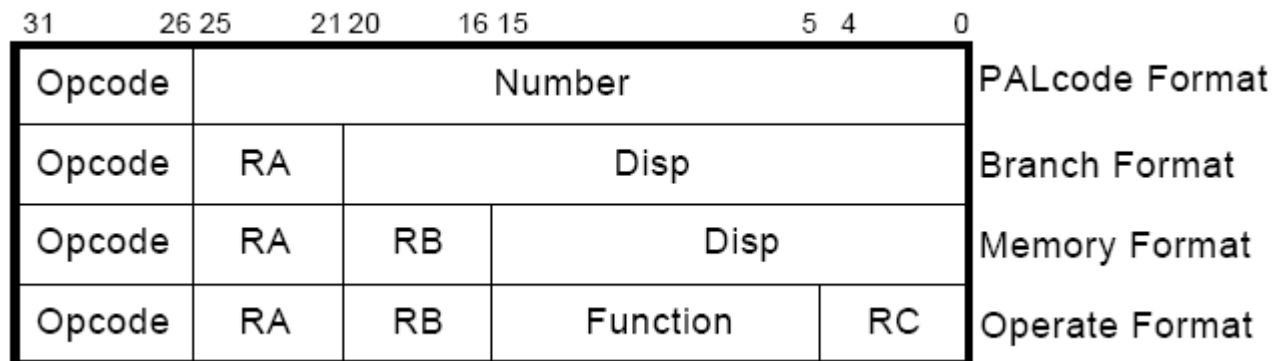
- **Uniform decode:** Same bits in each instruction correspond to the same meaning
 - Opcode is always in the same location
 - Ditto operand specifiers, immediate values, ...
 - Many “RISC” ISAs: Alpha, MIPS, SPARC
 - + Easier decode, simpler hardware
 - + Enables parallelism: generate target address before knowing the instruction is a branch
 - Restricts instruction format (fewer instructions?) or wastes space
- **Non-uniform decode**
 - E.g., opcode can be the 1st-7th byte in x86
 - + More compact and powerful instruction format
 - More complex decode logic

x86 vs. Alpha Instruction Formats

■ x86:

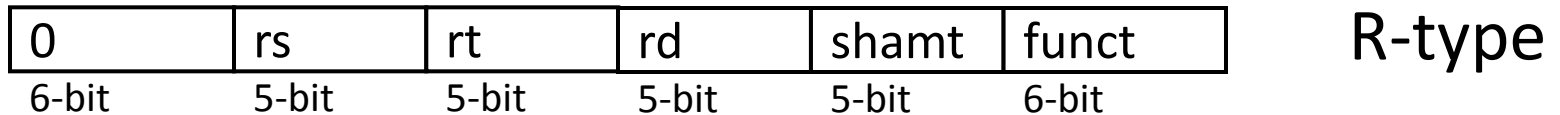


■ Alpha:



MIPS Instruction Format

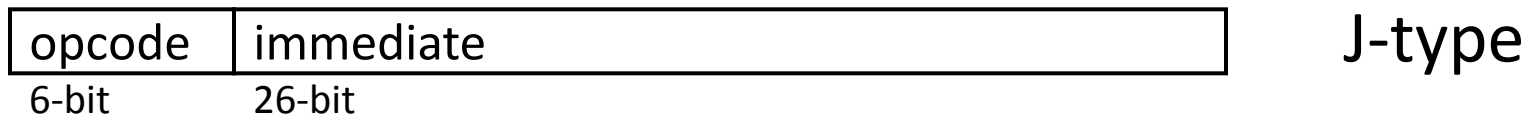
- R-type, 3 register operands



- I-type, 2 register operands and 16-bit immediate operand



- J-type, 26-bit immediate operand



- Simple Decoding

- 4 bytes per instruction, regardless of format
- must be 4-byte aligned (2 lsb of PC must be 2b'00)
- format and fields easy to extract in hardware

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

Cond	0	0	1	Opcode				S	Rn	Rd	Operand 2						<i>Data Processing / PSR Transfer</i>						
Cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm	<i>Multiply</i>						
Cond	0	0	0	0	1	U	A	S	RdHi	RdLo	Rn		1	0	0	1	Rm	<i>Multiply Long</i>					
Cond	0	0	0	1	0	B	0	0	Rn	Rd	0	0	0	0	1	0	0	1	Rm	<i>Single Data Swap</i>			
Cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn	<i>Branch and Exchange</i>
Cond	0	0	0	P	U	0	W	L	Rn	Rd	0	0	0	0	1	S	H	1	Rm	<i>Halfword Data Transfer: register offset</i>			
Cond	0	0	0	P	U	1	W	L	Rn	Rd	Offset		1	S	H	1	Offset	<i>Halfword Data Transfer: immediate offset</i>					
Cond	0	1	1	P	U	B	W	L	Rn	Rd	Offset						<i>Single Data Transfer</i>						
Cond	0	1	1													1		<i>Undefined</i>					
Cond	1	0	0	P	U	S	W	L	Rn	Register List							<i>Block Data Transfer</i>						
Cond	1	0	1	L	Offset											<i>Branch</i>							
Cond	1	1	0	P	U	N	W	L	Rn	CRd	CP#	Offset				<i>Coprocessor Data Transfer</i>							
Cond	1	1	1	0	CP Opc				CRn	CRd	CP#	CP	0	CRm	<i>Coprocessor Data Operation</i>								
Cond	1	1	1	0	CP Opc			L	CRn	Rd	CP#	CP	1	CRm	<i>Coprocessor Register Transfer</i>								
Cond	1	1	1	1	Ignored by processor											<i>Software Interrupt</i>							

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

Figure 4-1: ARM instruction set formats

A Note on Length and Uniformity

- Uniform decode usually goes with fixed length
- In a variable length ISA, uniform decode can be a property of instructions of the same length
 - It is hard to think of it as a property of instructions of different lengths

A Note on RISC vs. CISC

- Usually, ...
- RISC
 - Simple instructions
 - Fixed length
 - Uniform decode
 - Few addressing modes
- CISC
 - Complex instructions
 - Variable length
 - Non-uniform decode
 - Many addressing modes

ISA-level Tradeoffs: Number of Registers

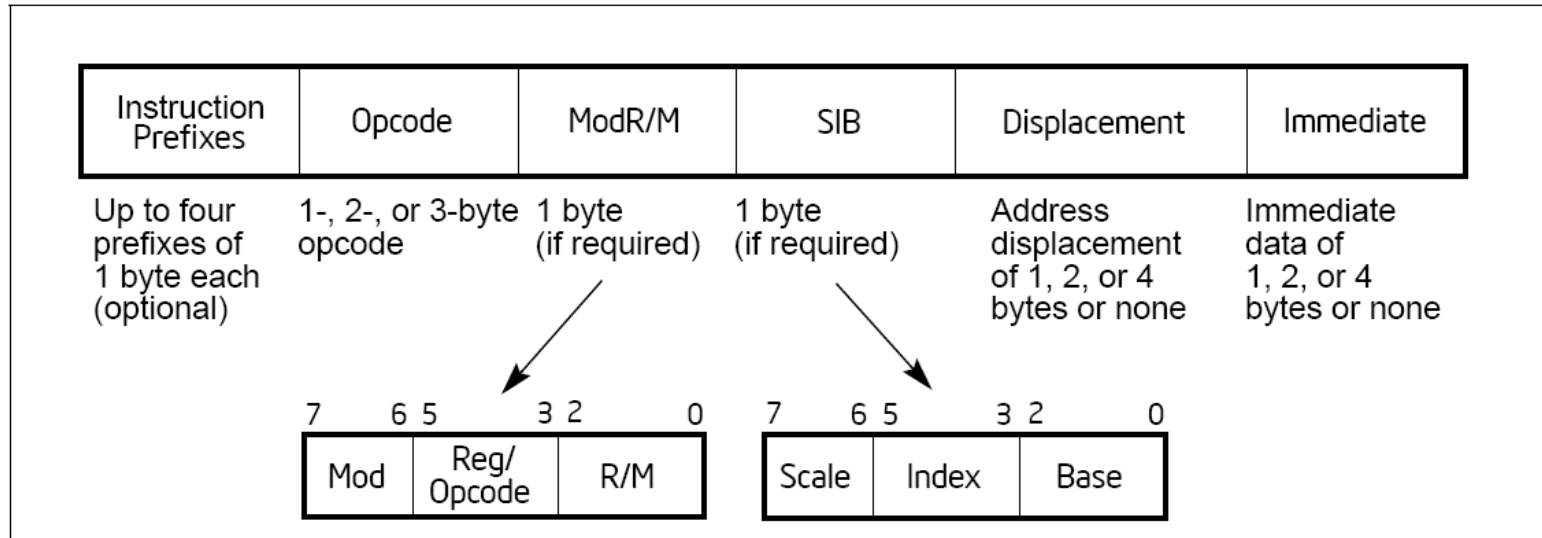
- Affects:
 - Number of bits used for encoding register address
 - Number of values kept in fast storage (register file)
 - (uarch) Size, access time, power consumption of register file
- Large number of registers:
 - + Enables better register allocation (and optimizations) by compiler → fewer saves/restores
 - Larger instruction size
 - Larger register file size

ISA-level Tradeoffs: Addressing Modes

- Addressing mode specifies how to obtain an operand of an instruction
 - Register
 - Immediate
 - Memory (displacement, register indirect, indexed, absolute, memory indirect, autoincrement, autodecrement, ...)
- More modes:
 - + help better support programming constructs (arrays, pointer-based accesses)
 - make it harder for the architect to design
 - too many choices for the compiler?
 - Many ways to do the same thing complicates compiler design
 - *Wulf, “Compilers and Computer Architecture,” IEEE Computer 1981*

x86 vs. Alpha Instruction Formats

■ x86:



■ Alpha:

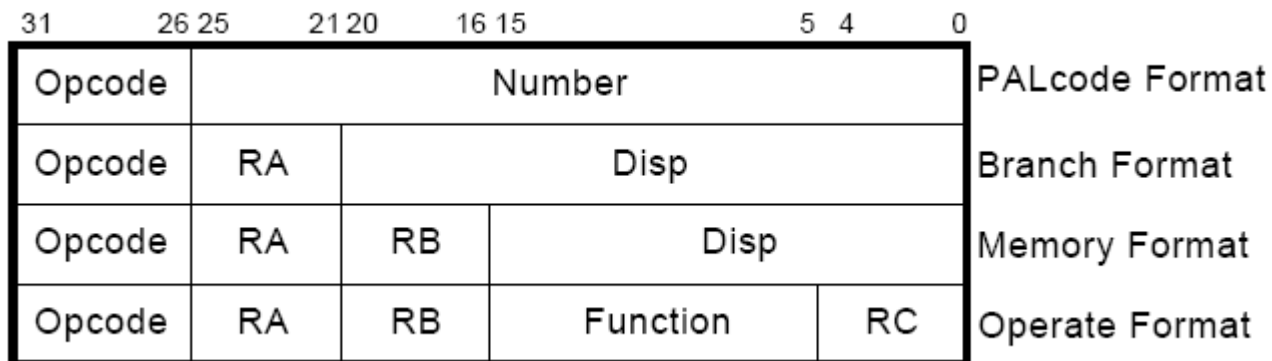


Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

x86

			AL AX	CL CX	DL DX	BL BX	AH SP	CH BP	DH SI	BH DI
			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
			0	1	2	3	4	5	6	7
			000	001	010	011	100	101	110	111
			Value of ModR/M Byte (in Hexadecimal)							
Effective Address	Mod	R/M								
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--] ¹		100	04	0C	14	1C	24	2C	34	3C
disp32 ²		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8 ³	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

register
indirect

absolute

register +
displacement

register

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table.

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base – (In binary) Base –			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX] [ECX] [EDX] [EBX] none [EBP] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 08 10 18 20 28 30 38	01 09 11 19 21 29 31 39	02 0A 12 1A 22 2A 32 3A	03 0B 13 1B 23 2B 33 3B	04 0C 14 1C 24 2C 34 3C	05 0D 15 1D 25 2D 35 3D	06 0E 16 1E 26 2E 36 3E	07 0F 17 1F 27 2F 37 3F
[EAX+2] [ECX+2] [EDX+2] [EBX+2] none [EBP+2] [ESI+2] [EDI+2]	01	000 001 010 011 100 101 110 111	40 48 50 58 60 68 70 78	41 49 51 59 61 69 71 79	42 4A 52 5A 62 6A 72 7A	43 4B 53 5B 63 6B 73 7B	44 4C 54 5C 64 6C 74 7C	45 4D 55 5D 65 6D 75 7D	46 4E 56 5E 66 6E 76 7E	47 4F 57 5F 67 6F 77 7F
[EAX+4] [ECX+4] [EDX+4] [EBX+4] none [EBP+4] [ESI+4] [EDI+4]	10	000 001 010 011 100 101 110 111	80 88 90 98 A0 A8 B0 B8	81 89 91 99 A1 A9 B1 B9	82 8A 92 9A A2 AA B2 BA	83 8B 93 9B A3 AB B3 BB	84 8C 94 9C A4 AC B4 BC	85 8D 95 9D A5 AD B5 BD	86 8E 96 9E A6 AE B6 BE	87 8F 97 9F A7 AF B7 BF
[EAX+8] [ECX+8] [EDX+8] [EBX+8] none [EBP+8] [ESI+8] [EDI+8]	11	000 001 010 011 100 101 110 111	C0 C8 D0 D8 E0 E8 F0 F8	C1 C9 D1 D9 E1 E9 F1 F9	C2 CA D2 DA E2 EA F2 FA	C3 CB D3 DB E3 EB F3 FB	C4 CC D4 DC E4 EC F4 FC	C5 CD D5 DD E5 ED F5 FD	C6 CE D6 DE E6 EE F6 FE	C7 CF D7 DF E7 EF F7 FF

indexed
(base +
index)

scaled
(base +
index*4)

NOTES:

- The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

MOD bits	Effective Address
00	[scaled index] + disp32
01	[scaled index] + disp8 + [EBP]
10	[scaled index] + disp32 + [EBP]

X86 SIB-D Addressing Mode

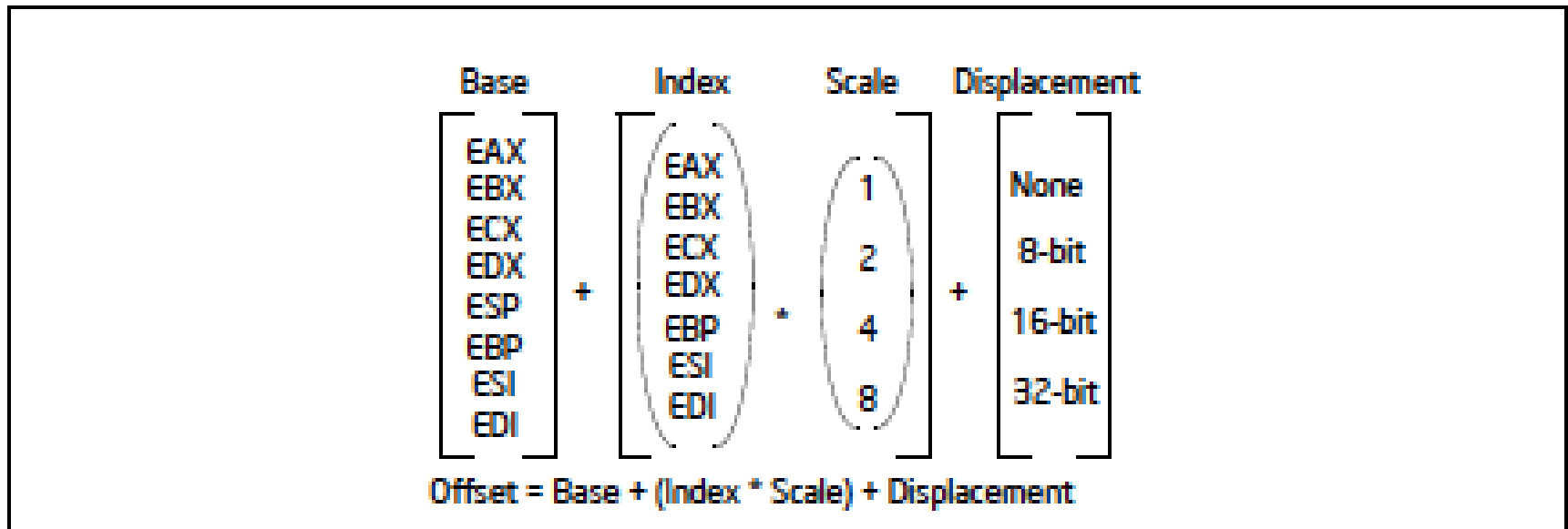


Figure 3-11. Offset (or Effective Address) Computation

x86 Manual Vol. 1, page 3-22 -- see course resources on website
Also, see Section 3.7.3 and 3.7.5

X86 Manual: Suggested Uses of Addressing Modes

The following addressing modes suggest uses for common combinations of address components.

- **Displacement** — A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.
- **Base** — A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.
- **Base + Displacement** — A base register and a displacement can be used together for two distinct purposes:
 - As an index into an array when the element size is not 2, 4, or 8 bytes—The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.
 - To access a field of a record: the base register holds the address of the beginning of the record, while the displacement is a static offset to the field.

An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.

x86 Manual Vol. 1, page 3-22 -- see course resources on website
Also, see Section 3.7.3 and 3.7.5

X86 Manual: Suggested Uses of Addressing Modes

- **(Index * Scale) + Displacement** — This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.
- **Base + Index + Displacement** — Using two registers together supports either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).
- **Base + (Index * Scale) + Displacement** — Using all the addressing components together allows efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.

x86 Manual Vol. 1, page 3-22 -- see course resources on website
Also, see Section 3.7.3 and 3.7.5

Other Example ISA-level Tradeoffs

- Condition codes vs. not
- VLIW vs. single instruction
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Unaligned access vs. not
- Hardware interlocks vs. software-guaranteed interlocking
- Software vs. hardware managed page fault handling
- Cache coherence (hardware vs. software)
- ...

Back to Programmer vs. (Micro)architect

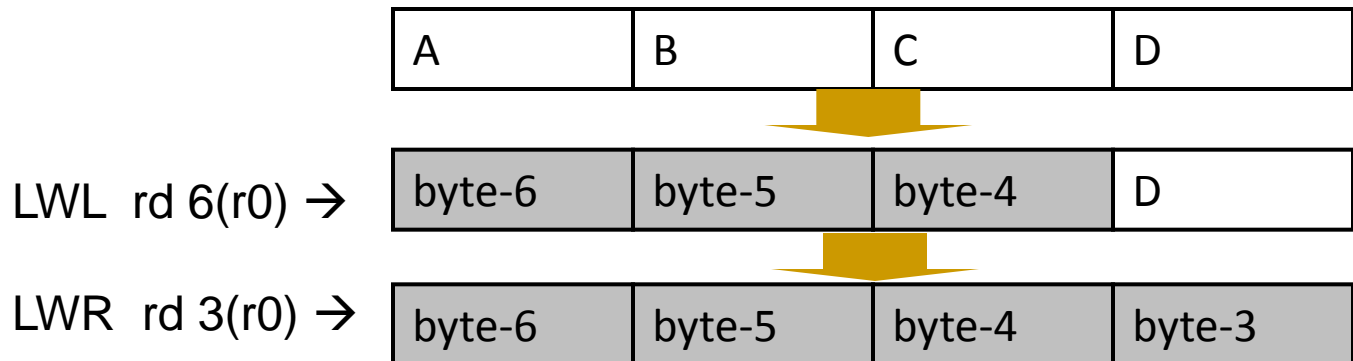
- Many ISA features designed to aid programmers
- But, complicate the hardware designer's job

- Virtual memory
 - vs. overlay programming
 - Should the programmer be concerned about the size of code blocks fitting physical memory?
- Addressing modes
- Unaligned memory access
 - Compile/programmer needs to align data

MIPS: Aligned Access

MSB	byte-3	byte-2	byte-1	byte-0	LSB
	byte-7	byte-6	byte-5	byte-4	

- LW/SW alignment restriction: 4-byte word-alignment
 - not designed to fetch memory bytes not within a word boundary
 - not designed to rotate unaligned bytes into registers
- Provide separate opcodes for the “infrequent” case



- LWL/LWR is slower
- Note LWL and LWR still fetch within word boundary

X86: Unaligned Access

- LD/ST instructions automatically align data that spans a “word” boundary
- Programmer/compiler does not need to worry about where data is stored (whether or not in a word-aligned location)

4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.

X86: Unaligned Access

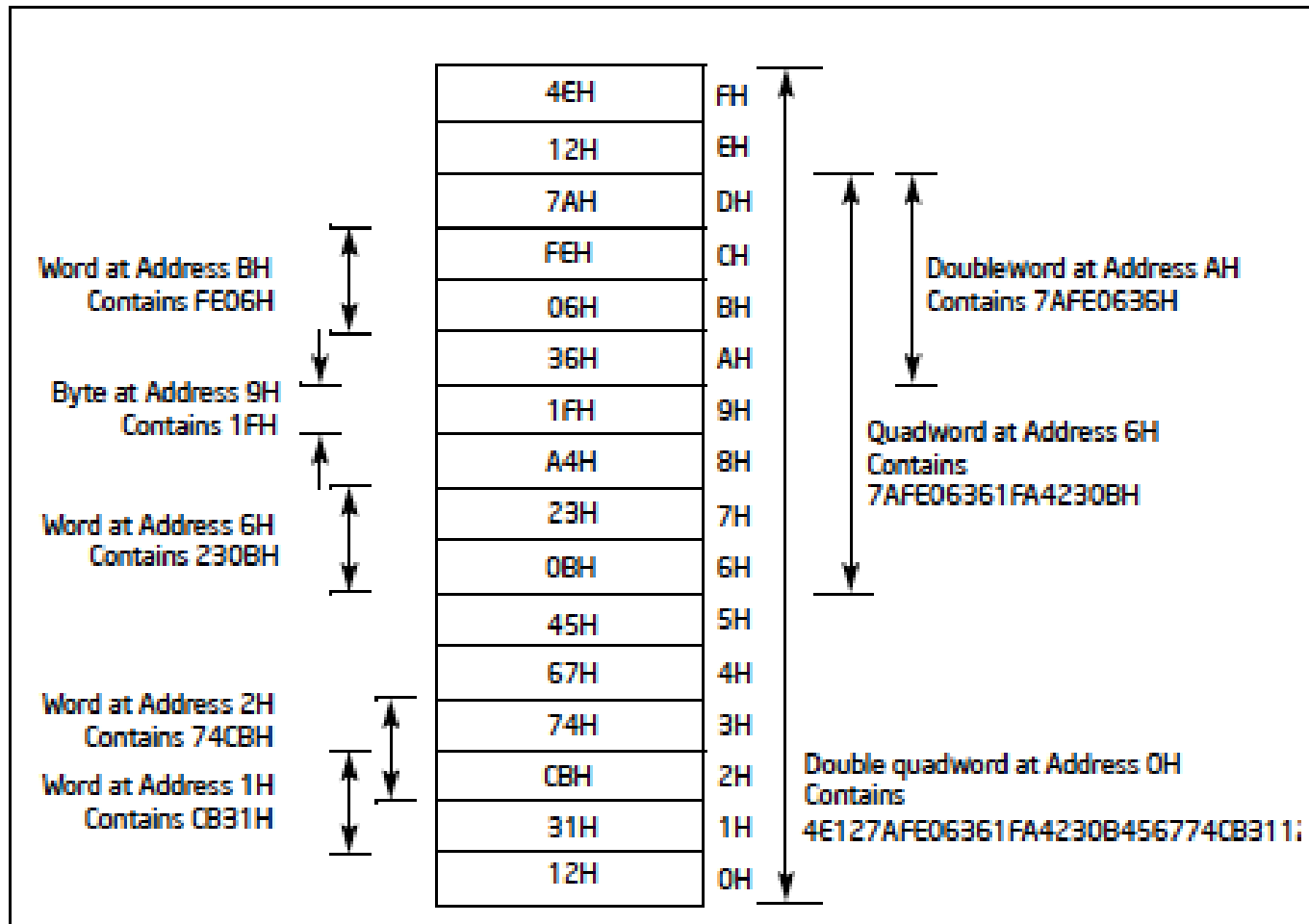


Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory

Aligned vs. Unaligned Access

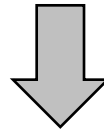
- Pros of having no restrictions on alignment
- Cons of having no restrictions on alignment
- Filling in the above: an exercise for you...

Implementing the ISA: Microarchitecture Basics

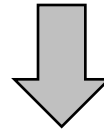
How Does a Machine Process Instructions?

- What does processing an instruction mean?
- Remember the von Neumann model

A = Architectural (programmer visible) state before an instruction is processed



Process instruction



A' = Architectural (programmer visible) state after an instruction is processed

- Processing an instruction: Transforming A to A' according to the ISA specification of the instruction

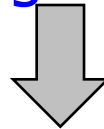
The “Process instruction” Step

- ISA specifies abstractly what A' should be, given an instruction and A
 - It defines an abstract finite state machine where
 - State = programmer-visible state
 - Next-state logic = instruction execution specification
 - From ISA point of view, there are no “intermediate states” between A and A' during instruction execution
 - One state transition per instruction
- Microarchitecture implements how A is transformed to A'
 - There are many choices in implementation
 - We can have programmer-invisible state to optimize the speed of instruction execution: multiple state transitions per instruction
 - Choice 1: $A \rightarrow A'$ (transform A to A' in a single clock cycle)
 - Choice 2: $A \rightarrow A+MS1 \rightarrow A+MS2 \rightarrow A+MS3 \rightarrow A'$ (take multiple clock cycles to transform A to A')

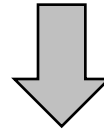
A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
 - *No intermediate, programmer-invisible state updates*

A = Architectural (programmer visible) state
at the beginning of a clock cycle



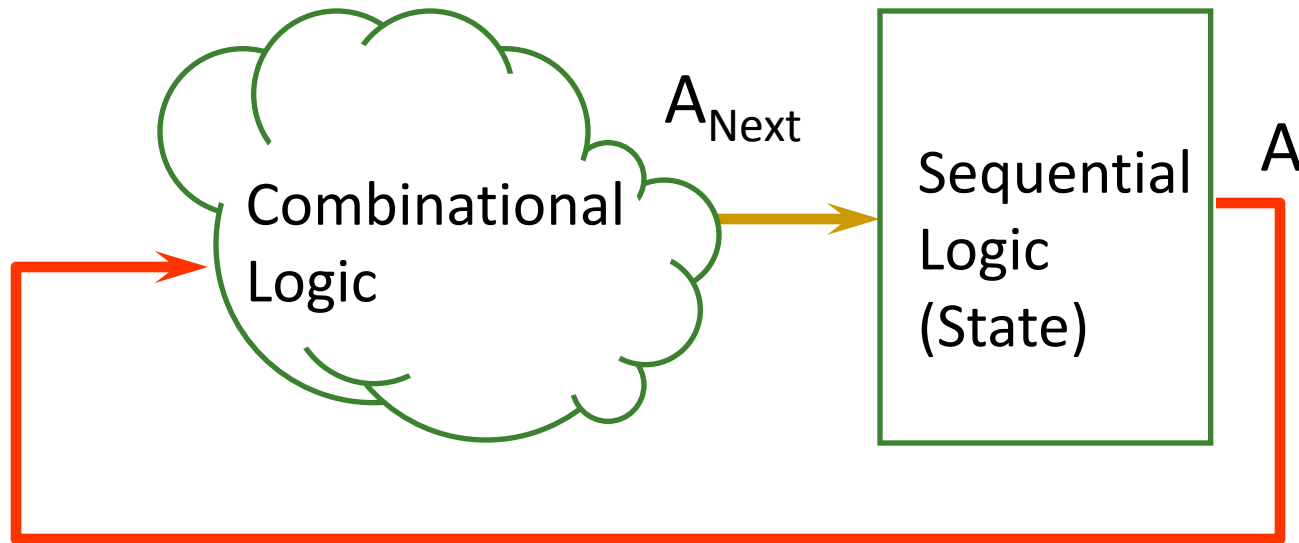
Process instruction in one clock cycle



A' = Architectural (programmer visible) state
at the end of a clock cycle

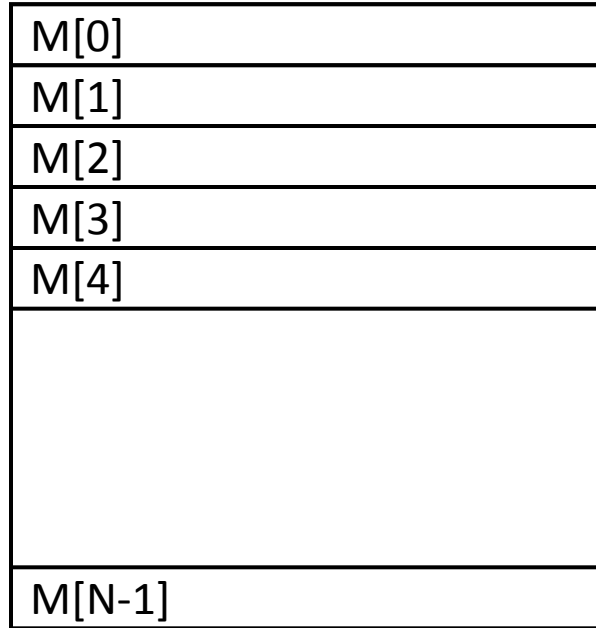
A Very Basic Instruction Processing Engine

- Single-cycle machine



- What is the *clock cycle time* determined by?
- What is the *critical path* of the combinational logic determined by?

Remember: Programmer Visible (Architectural) State

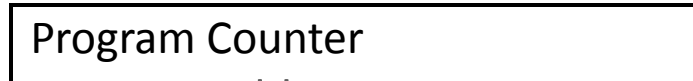


Memory
array of storage locations
indexed by an address



Registers

- given special names in the ISA (as opposed to addresses)
- general vs. special purpose



memory address
of the current instruction

Instructions (and programs) specify how to transform
the values of programmer visible state

Single-cycle vs. Multi-cycle Machines

■ Single-cycle machines

- ❑ Each instruction takes a single clock cycle
- ❑ All state updates made at the end of an instruction's execution
- ❑ Big disadvantage: The slowest instruction determines cycle time → long clock cycle time

■ Multi-cycle machines

- ❑ Instruction processing broken into multiple cycles/stages
- ❑ State updates can be made during an instruction's execution
- ❑ Architectural state updates made only at the end of an instruction's execution
- ❑ Advantage over single-cycle: The slowest "stage" determines cycle time

- Both single-cycle and multi-cycle machines literally follow the von Neumann model at the microarchitecture level

Instruction Processing “Cycle”

- Instructions are processed under the direction of a “control unit” step by step.
- Instruction cycle: Sequence of steps to process an instruction
- Fundamentally, there are six phases:
 - Fetch
 - Decode
 - Evaluate Address
 - Fetch Operands
 - Execute
 - Store Result
- Not all instructions require all six stages (see P&P Ch. 4)

Instruction Processing “Cycle” vs. Machine Clock Cycle

- Single-cycle machine:
 - All six phases of the instruction processing cycle take a *single machine clock cycle* to complete
- Multi-cycle machine:
 - All six phases of the instruction processing cycle can take *multiple machine clock cycles* to complete
 - In fact, **each phase can take multiple clock cycles to complete**

Instruction Processing Viewed Another Way

- Instructions transform Data (AS) to Data' (AS')
- This transformation is done by functional units
 - Units that “operate” on data
- These units need to be told what to do to the data
- An instruction processing engine consists of two components
 - **Datapath**: Consists of **hardware elements that deal with and transform data signals**
 - functional units that operate on data
 - hardware structures (e.g. wires and muxes) that enable the flow of data into the functional units and registers
 - storage units that store data (e.g., registers)
 - **Control logic**: Consists of **hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data**

Single-cycle vs. Multi-cycle: Control & Data

- Single-cycle machine:
 - Control signals are generated in the same clock cycle as data signals are operated on
 - Everything related to an instruction happens in one clock cycle
- Multi-cycle machine:
 - Control signals needed in the next cycle can be generated in the previous cycle
 - Latency of control processing can be overlapped with latency of datapath operation
- We will see the difference clearly in *microprogrammed multi-cycle microarchitecture*

Many Ways of Datapath and Control Design

- There are many ways of designing the data path and control logic
- Single-cycle, multi-cycle, pipelined datapath and control
- Single-bus vs. multi-bus datapaths
 - See your homework 2 question
- Hardwired/combinational vs. microcoded/microprogrammed control
 - Control signals generated by combinational logic versus
 - Control signals stored in a memory structure
- Control signals and structure depend on the datapath design

Flash-Forward: Performance Analysis

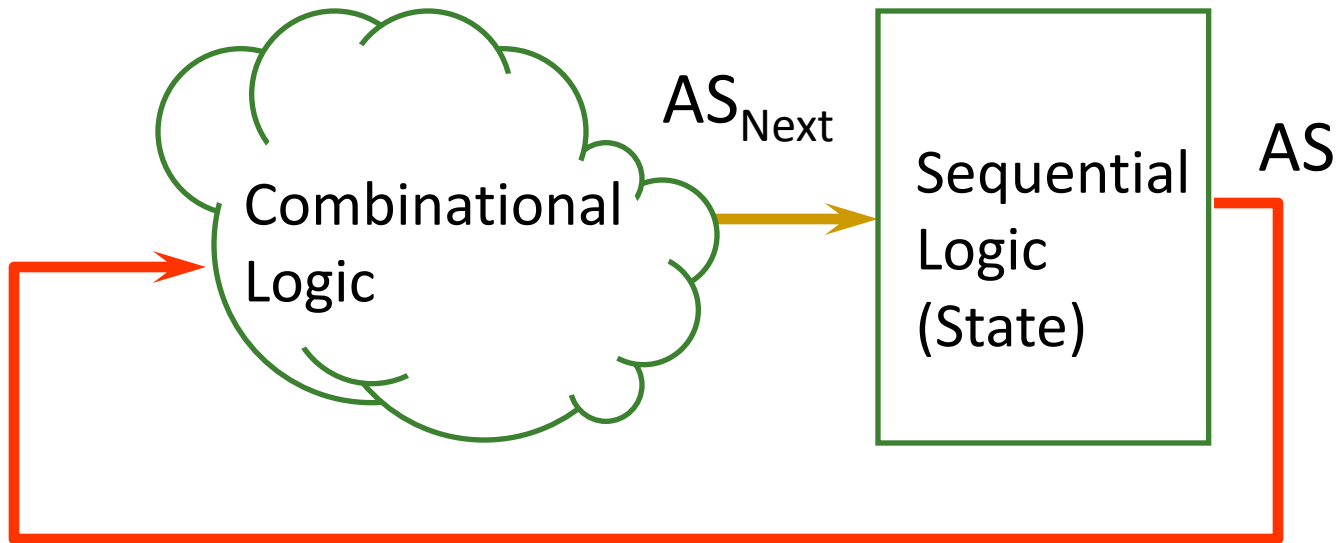
- Execution time of an instruction
 - $\{\text{CPI}\} \times \{\text{clock cycle time}\}$
 - Execution time of a program
 - Sum over all instructions [$\{\text{CPI}\} \times \{\text{clock cycle time}\}$]
 - $\{\# \text{ of instructions}\} \times \{\text{Average CPI}\} \times \{\text{clock cycle time}\}$
 - Single cycle microarchitecture performance
 - $\text{CPI} = 1$
 - Clock cycle time = long
 - Multi-cycle microarchitecture performance
 - $\text{CPI} = \text{different for each instruction}$
 - Average CPI \rightarrow hopefully small
 - Clock cycle time = short
- Now, we have two degrees of freedom to optimize independently**

A Single-Cycle Microarchitecture

A Closer Look

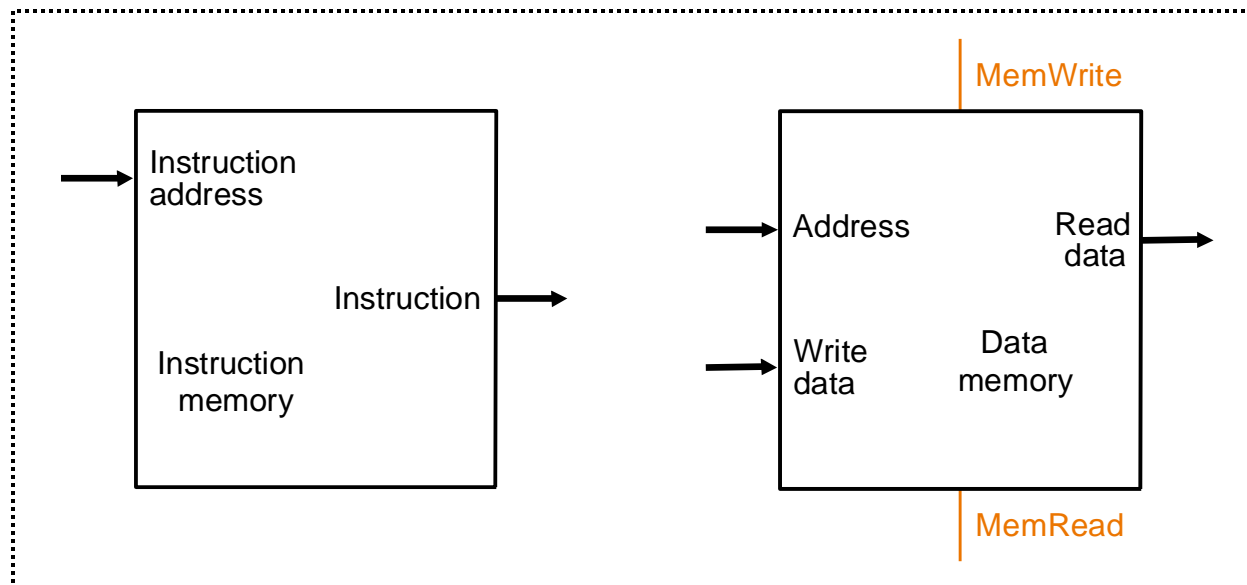
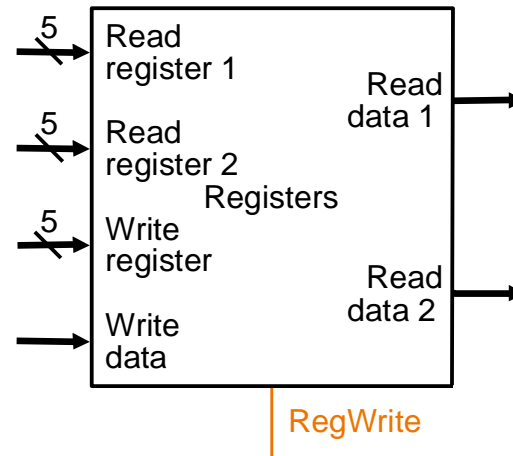
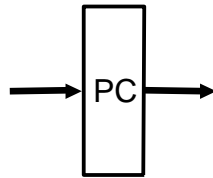
Remember...

- Single-cycle machine



Let's Start with the State Elements

- Data and control inputs

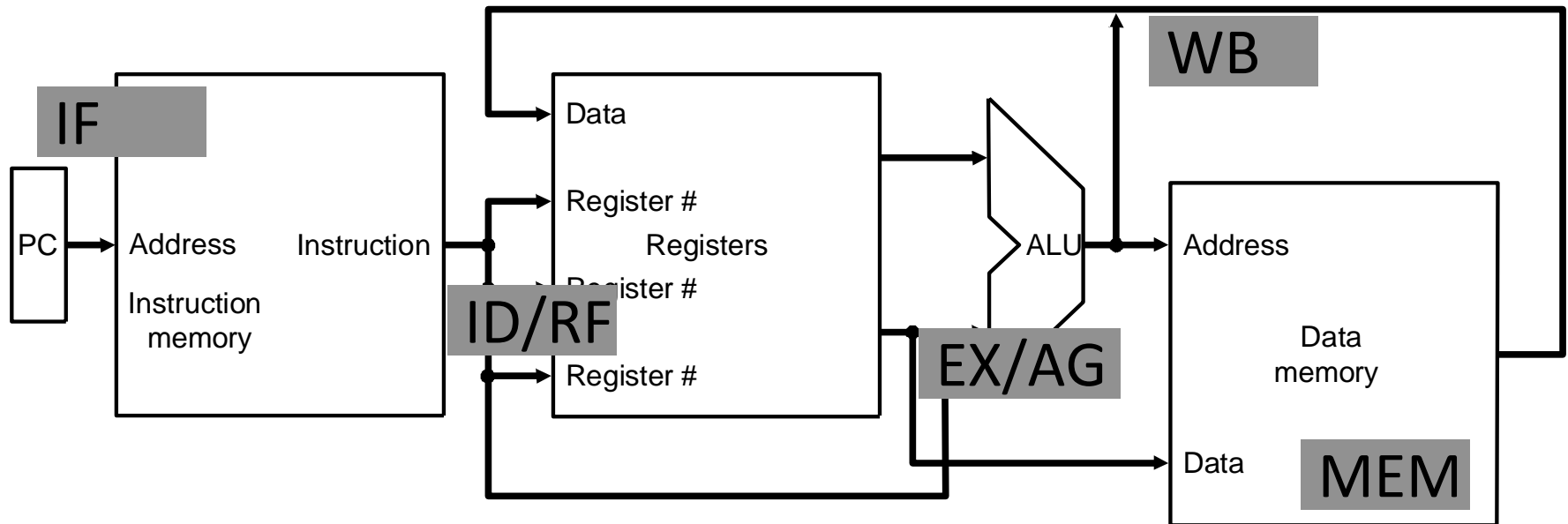


For Now, We Will Assume

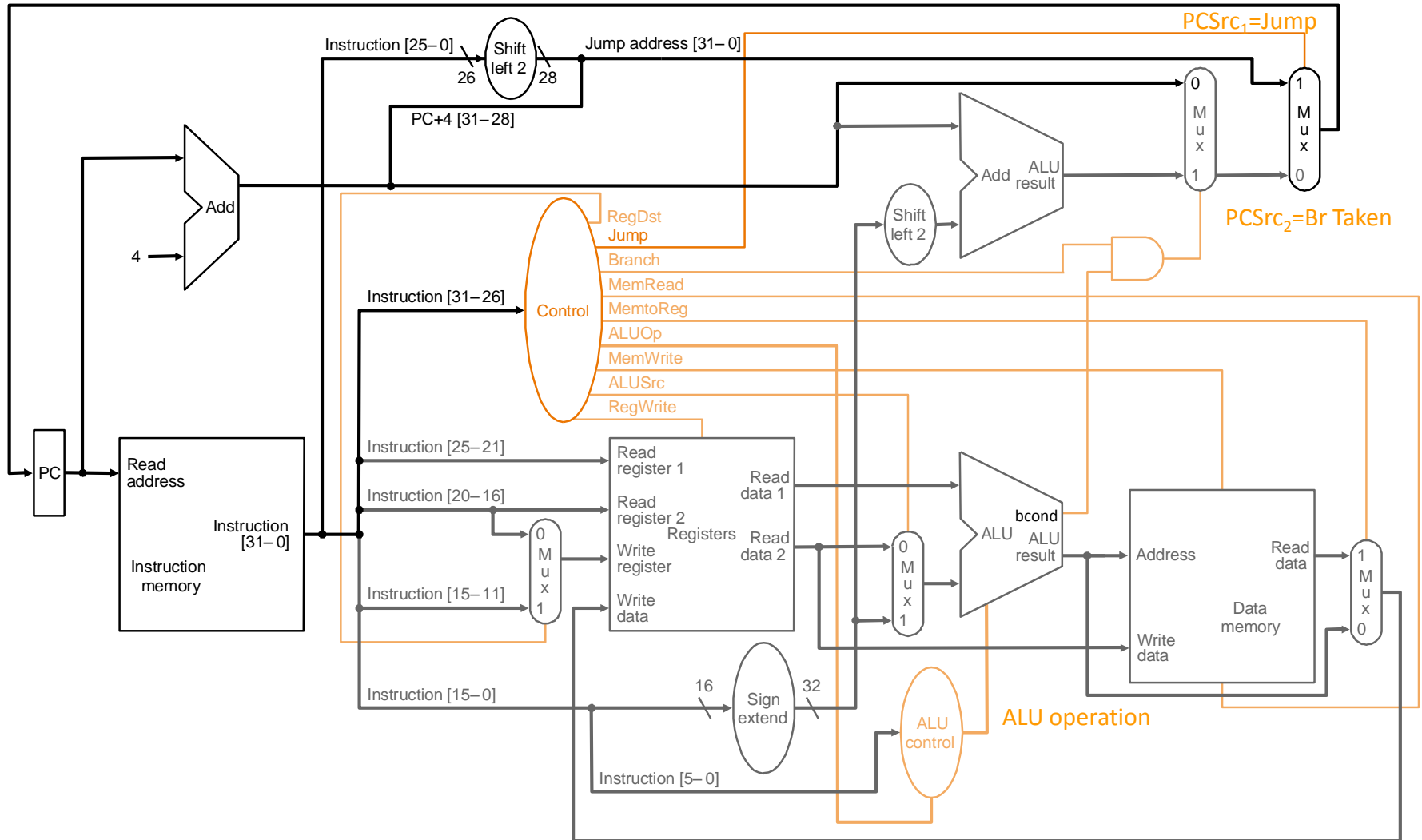
- “Magic” memory and register file
- Combinational read
 - output of the read data port is a combinational function of the register file contents and the corresponding read select port
- Synchronous write
 - the selected register is updated on the positive edge clock transition when write enable is asserted
 - Cannot affect read output in between clock edges
 - Can affect read output at clock edges (but who cares?)
- Single-cycle, synchronous memory
 - Contrast this with memory that tells when the data is ready
 - i.e., Ready bit: indicating the read or write is done

Instruction Processing

- 5 generic steps (P&H)
 - ❑ Instruction fetch (IF)
 - ❑ Instruction decode and register operand fetch (ID/RF)
 - ❑ Execute/Evaluate memory address (EX/AG)
 - ❑ Memory operand fetch (MEM)
 - ❑ Store/writeback result (WB)



What Is To Come: The Full Datapath



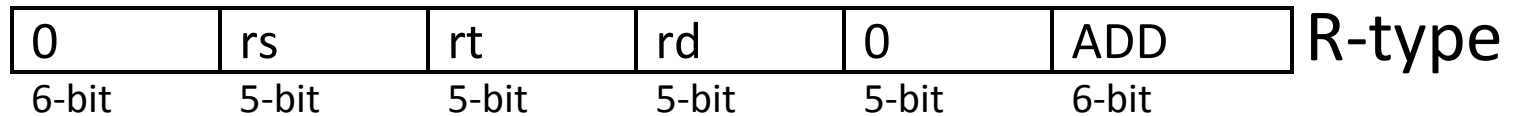
Single-Cycle Datapath for *Arithmetic and Logical Instructions*

R-Type ALU Instructions

- Assembly (e.g., register-register signed addition)

ADD rd_{reg} rs_{reg} rt_{reg}

- Machine encoding



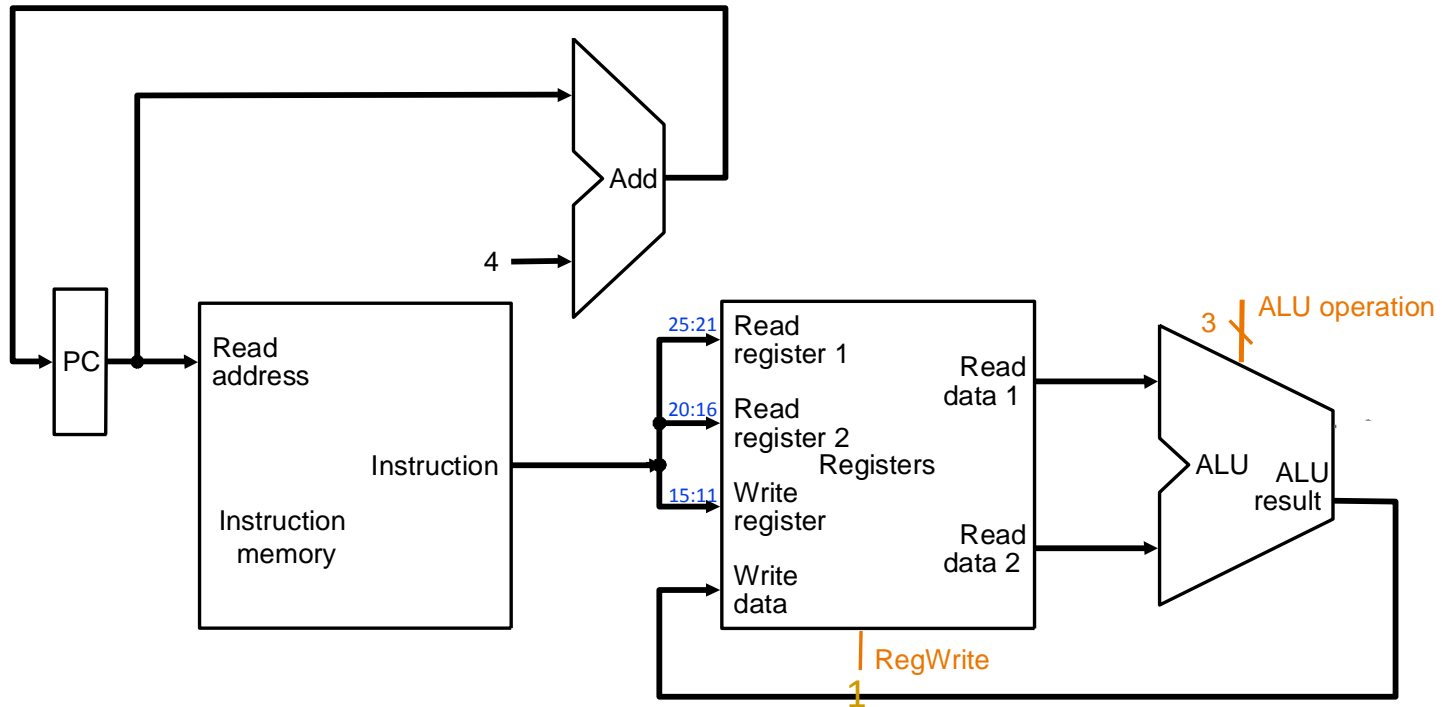
- Semantics

if MEM[PC] == ADD rd rs rt

GPR[rd] ← GPR[rs] + GPR[rt]

PC ← PC + 4

ALU Datapath



Combinational
state update logic

```

if MEM[PC] == ADD rd rs rt
    GPR[rd] ← GPR[rs] + GPR[rt]
    PC ← PC + 4
  
```

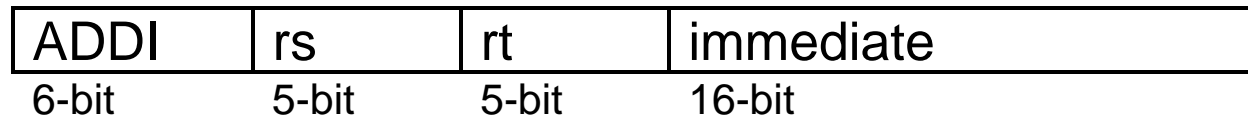
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

I-Type ALU Instructions

- Assembly (e.g., register-immediate signed additions)

ADDI rt_{reg} rs_{reg} $immediate_{16}$

- Machine encoding



I-type

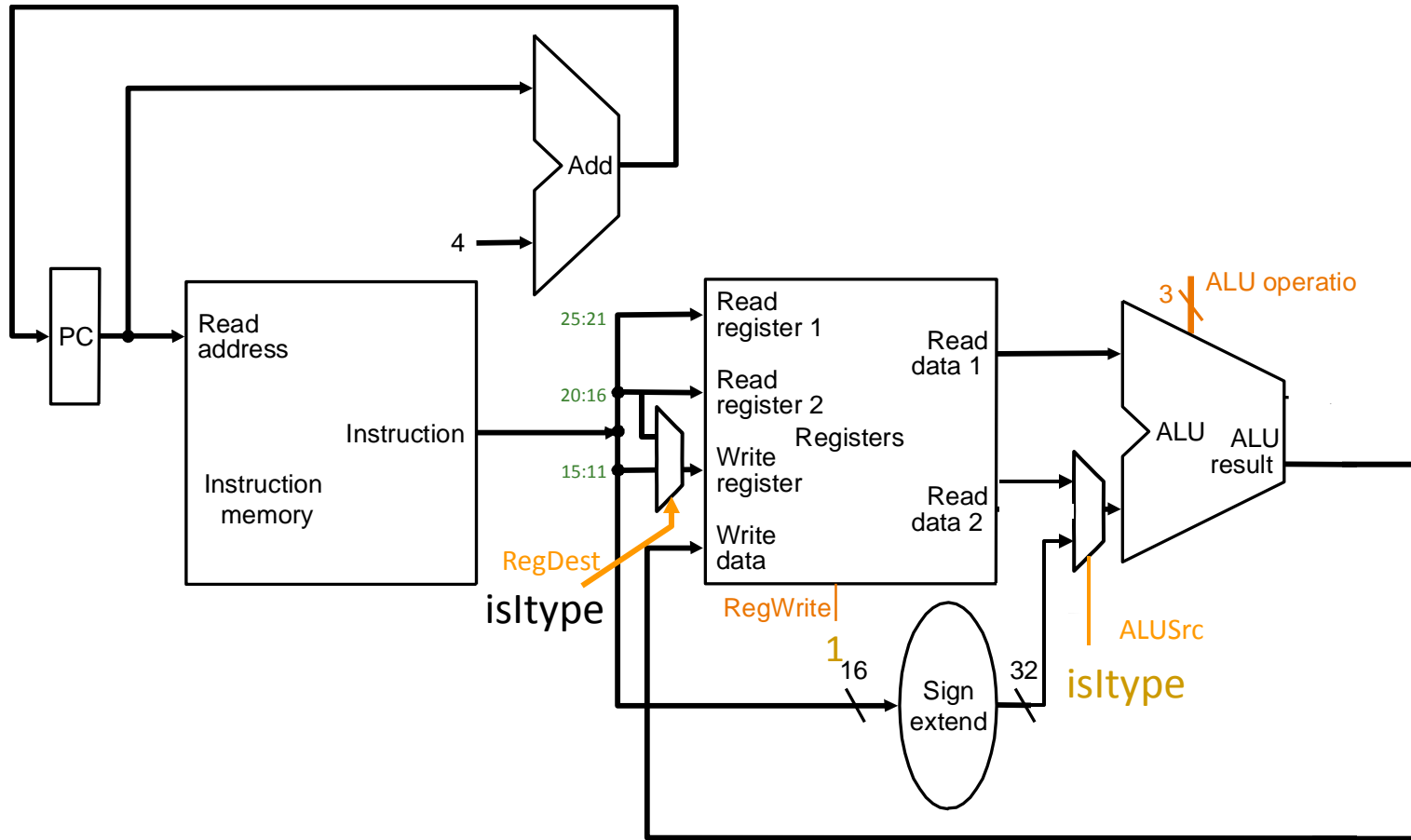
- Semantics

if MEM[PC] == ADDI rt rs $immediate$

$GPR[rt] \leftarrow GPR[rs] + \text{sign-extend}(immediate)$

$PC \leftarrow PC + 4$

Datapath for R and I-Type ALU Insts.



if MEM[PC] == ADDI rt rs immediate
 $GPR[rt] \leftarrow GPR[rs] + \text{sign-extend}(\text{immediate})$
 $PC \leftarrow PC + 4$



Combinational
state update logic

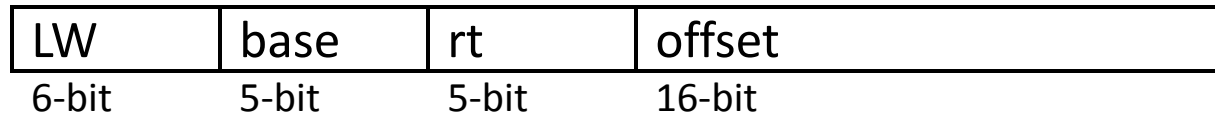
Single-Cycle Datapath for *Data Movement Instructions*

Load Instructions

- Assembly (e.g., load 4-byte word)

LW rt_{reg} $offset_{16}$ ($base_{reg}$)

- Machine encoding



I-type

- Semantics

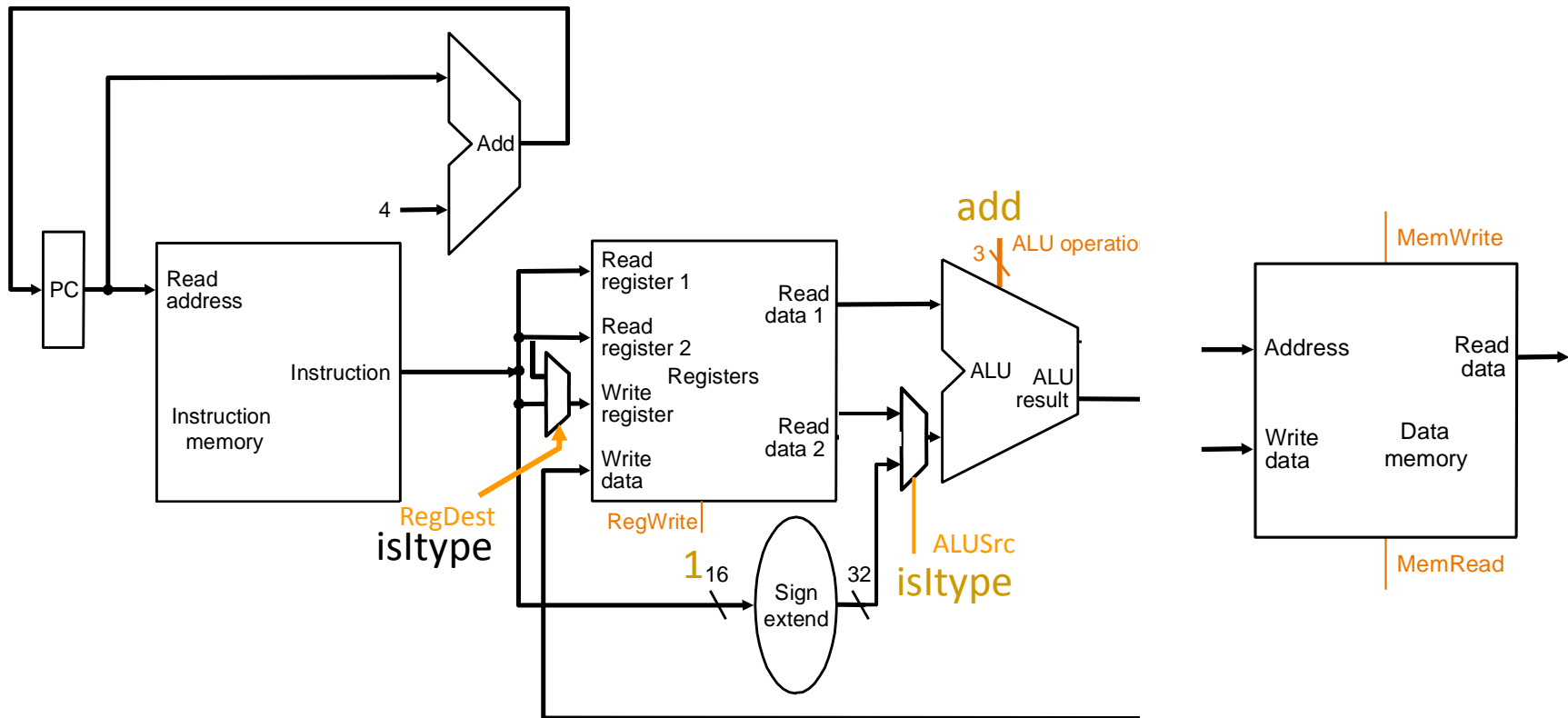
if $MEM[PC] == LW\ rt\ offset_{16}\ (base)$

EA = sign-extend(offset) + GPR[base]

GPR[rt] ← MEM[translate(EA)]

PC ← PC + 4

LW Datapath



if $MEM[PC] == LW \text{ rt offset}_{16} \text{ (base)}$
 $EA = \text{sign-extend}(\text{offset}) + GPR[\text{base}]$
 $GPR[\text{rt}] \leftarrow MEM[\text{translate}(EA)]$
 $PC \leftarrow PC + 4$



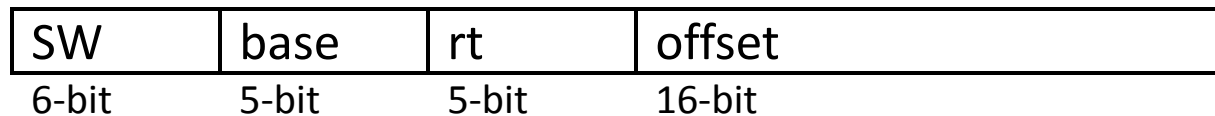
Combinational
state update logic 55

Store Instructions

- Assembly (e.g., store 4-byte word)

$SW\ rt_{reg}\ offset_{16}\ (base_{reg})$

- Machine encoding



I-type

- Semantics

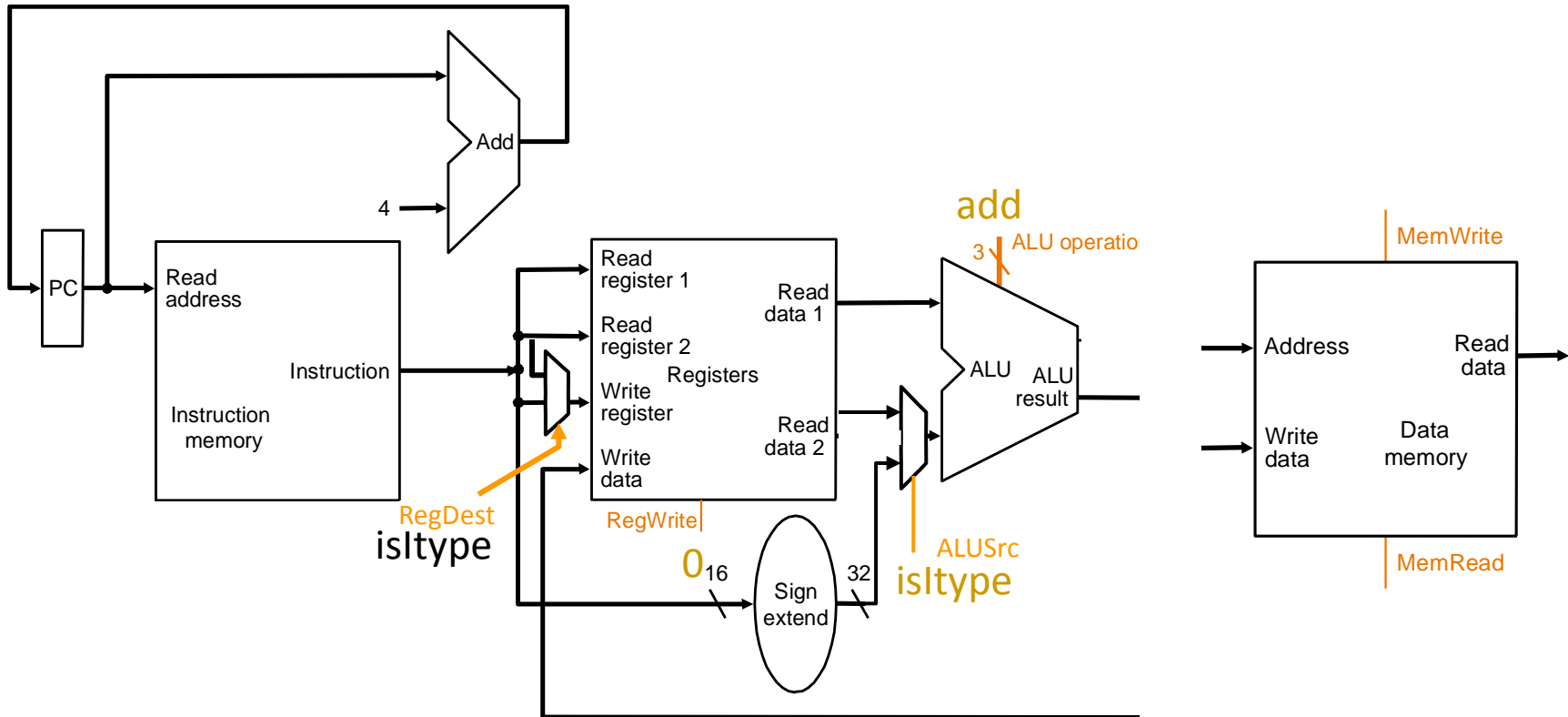
if $MEM[PC] == SW\ rt\ offset_{16}\ (base)$

$EA = \text{sign-extend}(\text{offset}) + GPR[\text{base}]$

$MEM[\text{translate}(EA)] \leftarrow GPR[rt]$

$PC \leftarrow PC + 4$

SW Datapath

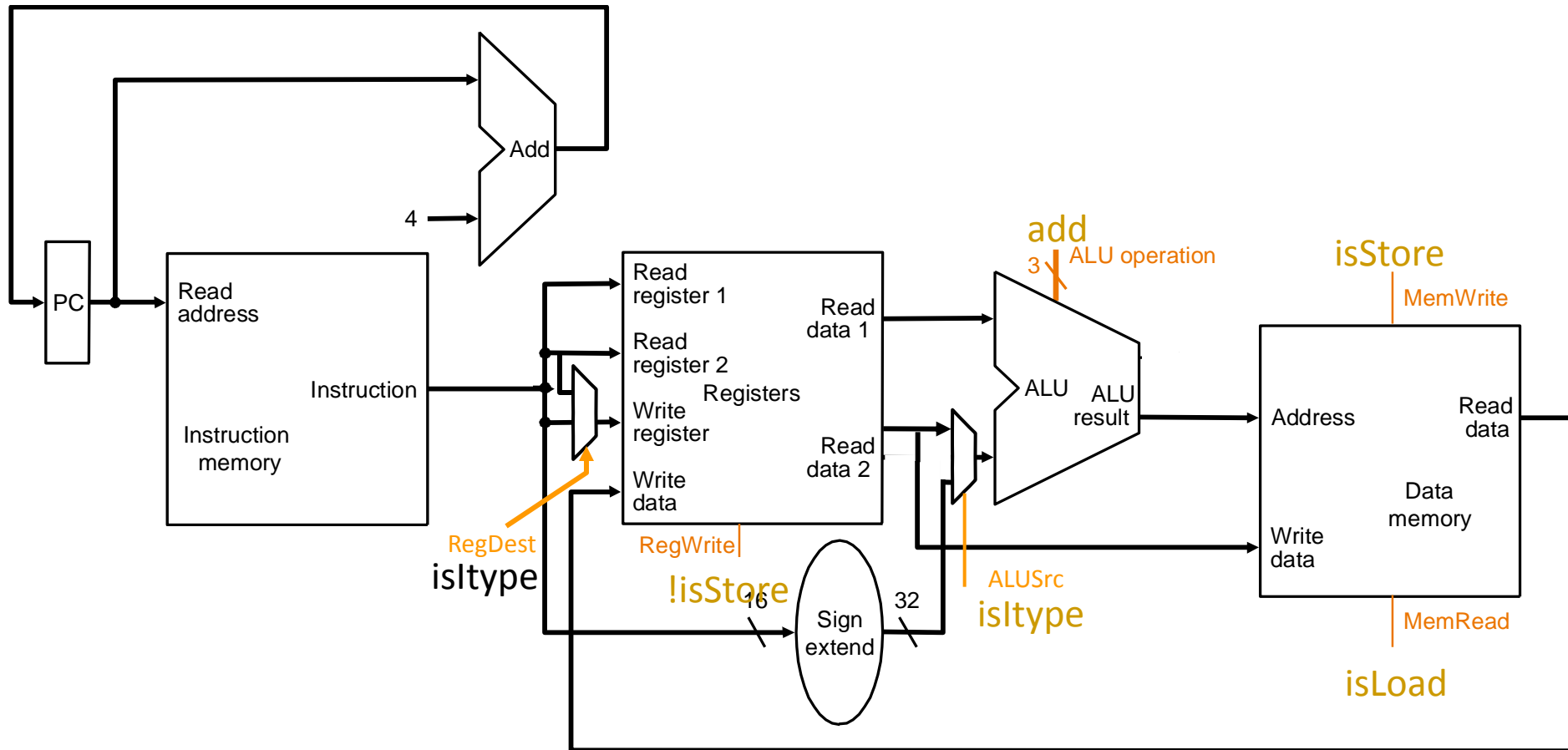


if $MEM[PC] == SW \text{ rt offset}_{16}(\text{base})$
 $EA = \text{sign-extend}(\text{offset}) + GPR[\text{base}]$
 $MEM[\text{translate}(EA)] \leftarrow GPR[\text{rt}]$
 $PC \leftarrow PC + 4$

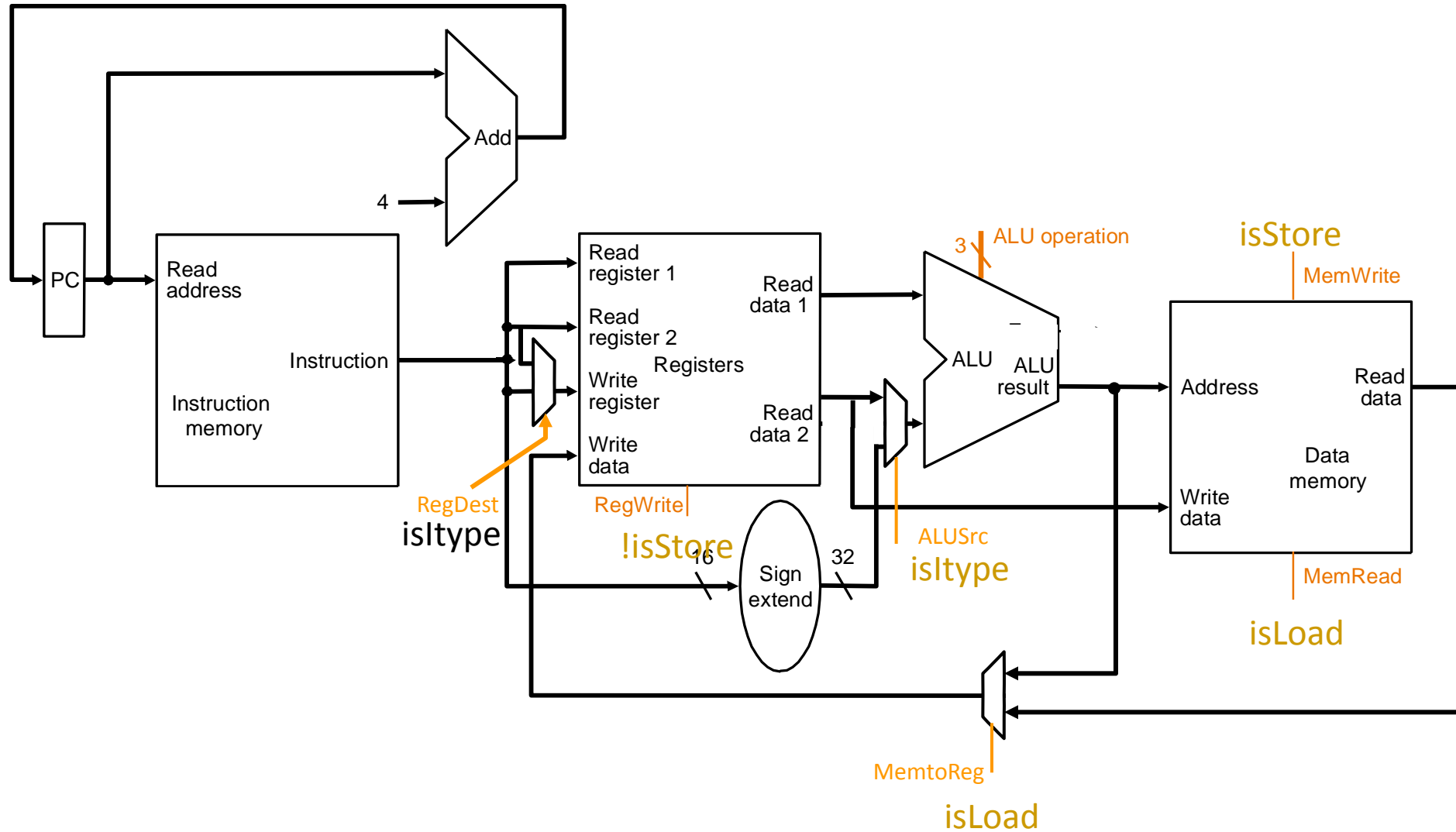


Combinational
state update logic 57

Load-Store Datapath



Datapath for Non-Control-Flow Insts.



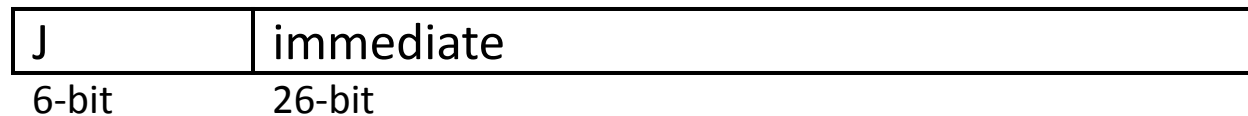
Single-Cycle Datapath for *Control Flow Instructions*

Unconditional Jump Instructions

- Assembly

J immediate₂₆

- Machine encoding



J-type

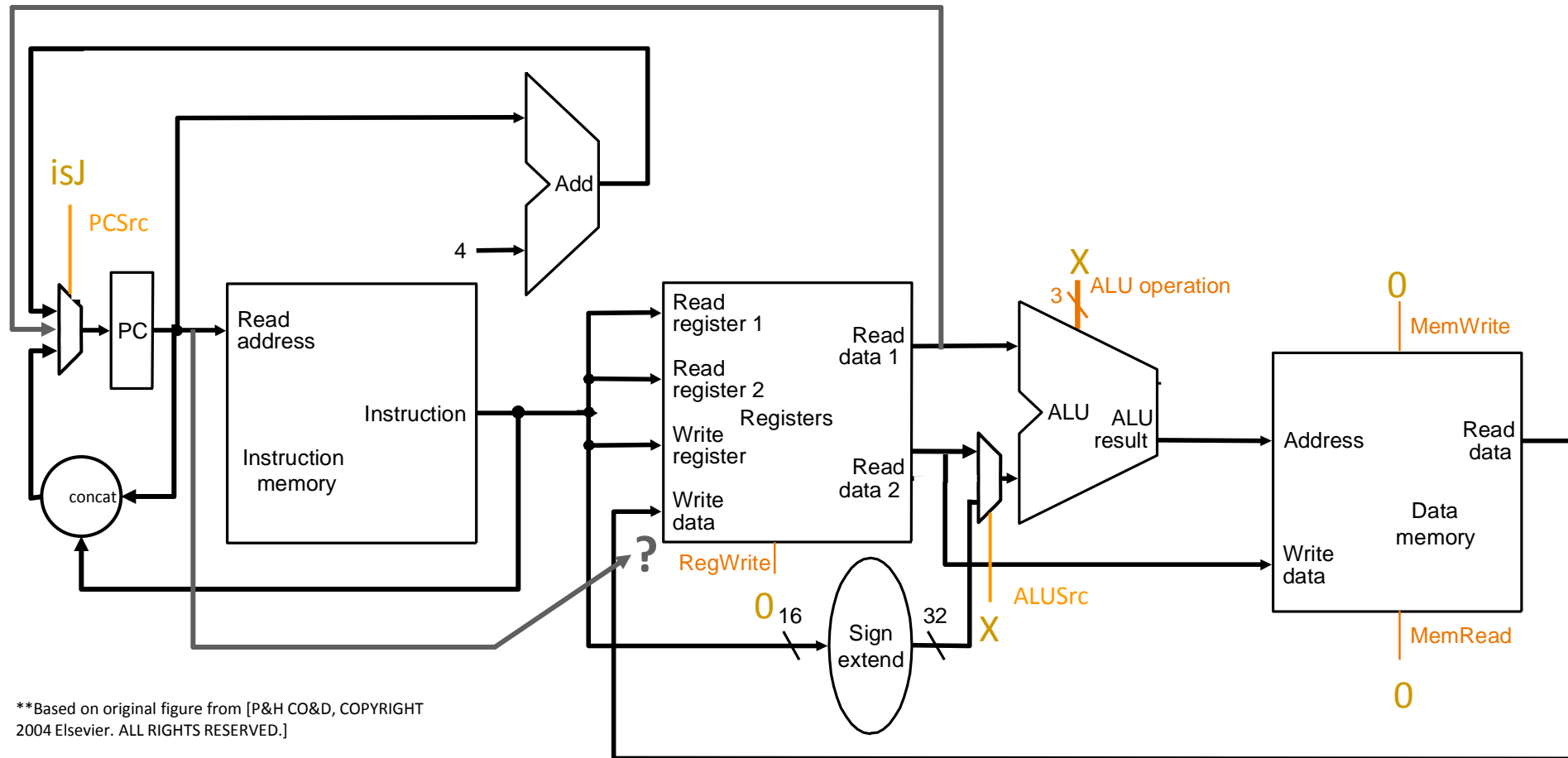
- Semantics

if MEM[PC] == J immediate₂₆

target = { PC[31:28], immediate₂₆, 2' b00 }

PC ← target

Unconditional Jump Datapath



**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

if MEM[PC]==J immediate26
 PC = { PC[31:28], immediate26, 2' b00 }

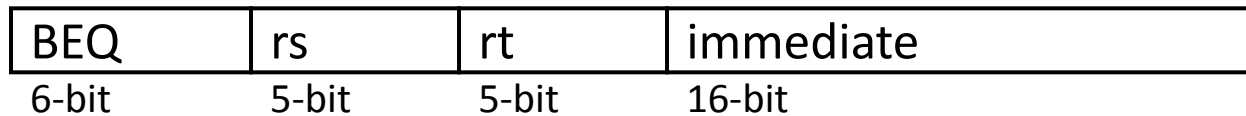
What about JR, JAL, JALR?

Conditional Branch Instructions

- Assembly (e.g., branch if equal)

BEQ rs_{reg} rt_{reg} immediate₁₆

- Machine encoding



I-type

- Semantics (assuming no branch delay slot)

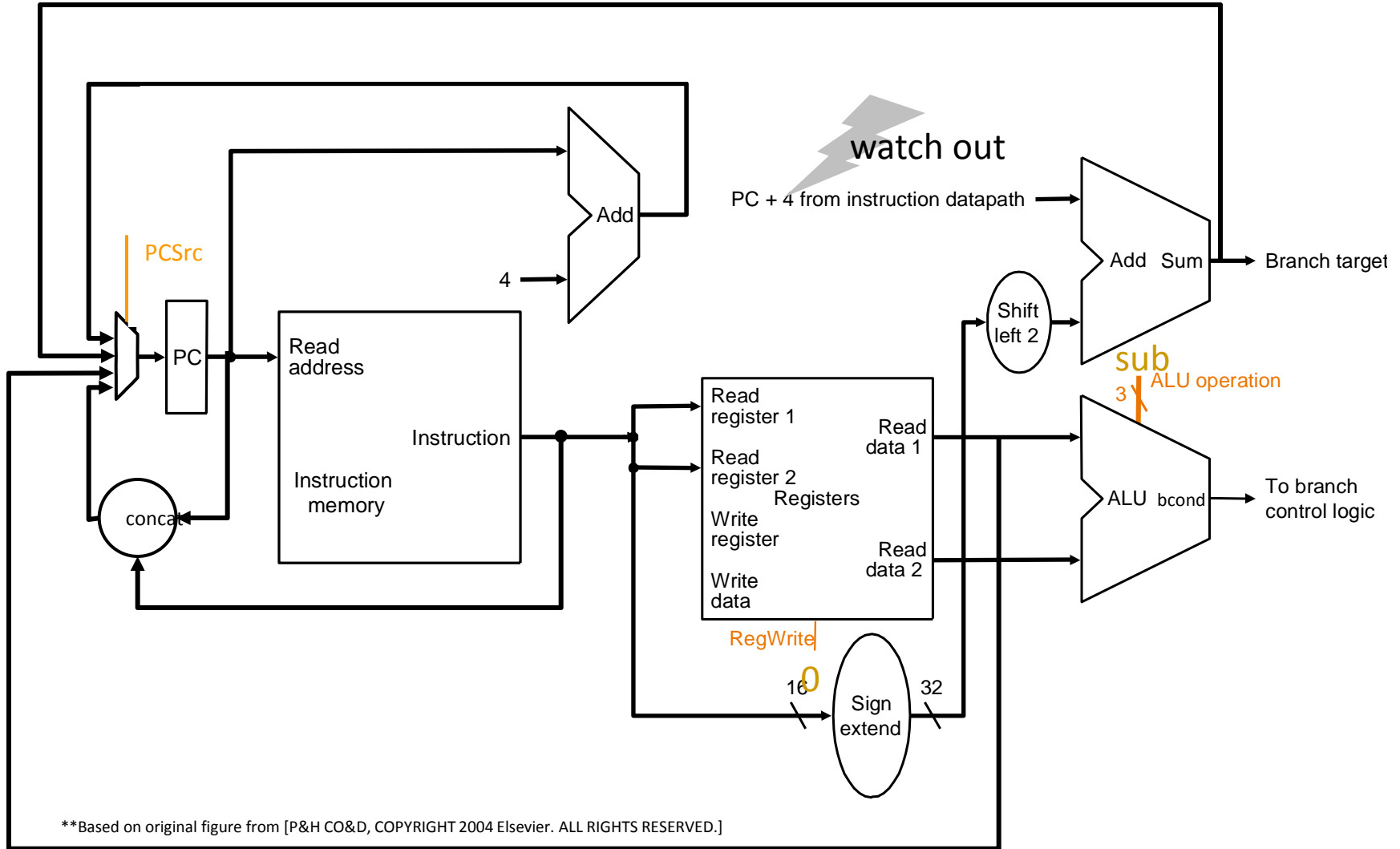
if MEM[PC] == BEQ rs rt immediate₁₆

target = PC + 4 + sign-extend(immediate) x 4

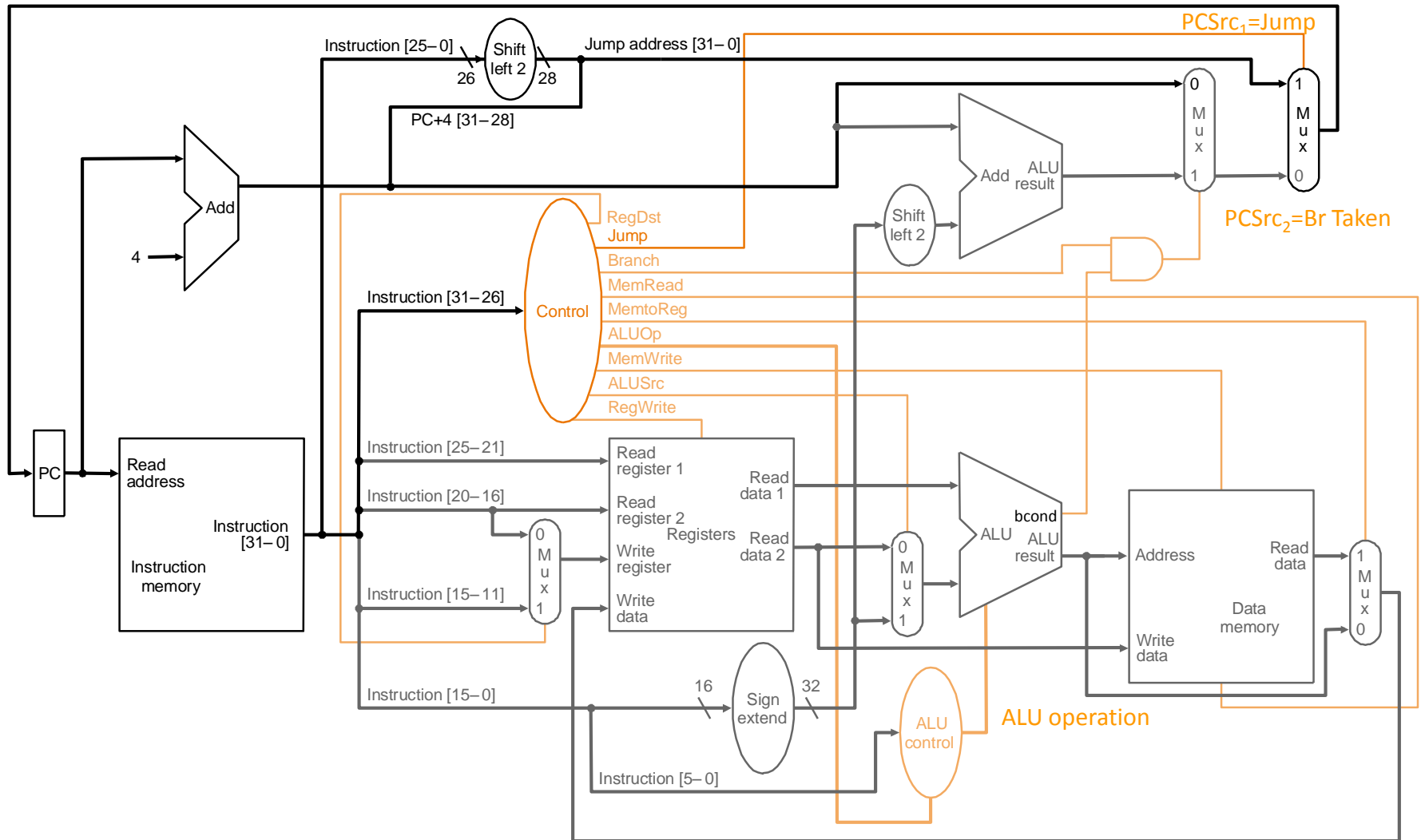
if GPR[rs] == GPR[rt] then PC ← target

else PC ← PC + 4

Conditional Branch Datapath (For You to Fix)



Putting It All Together



**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]