

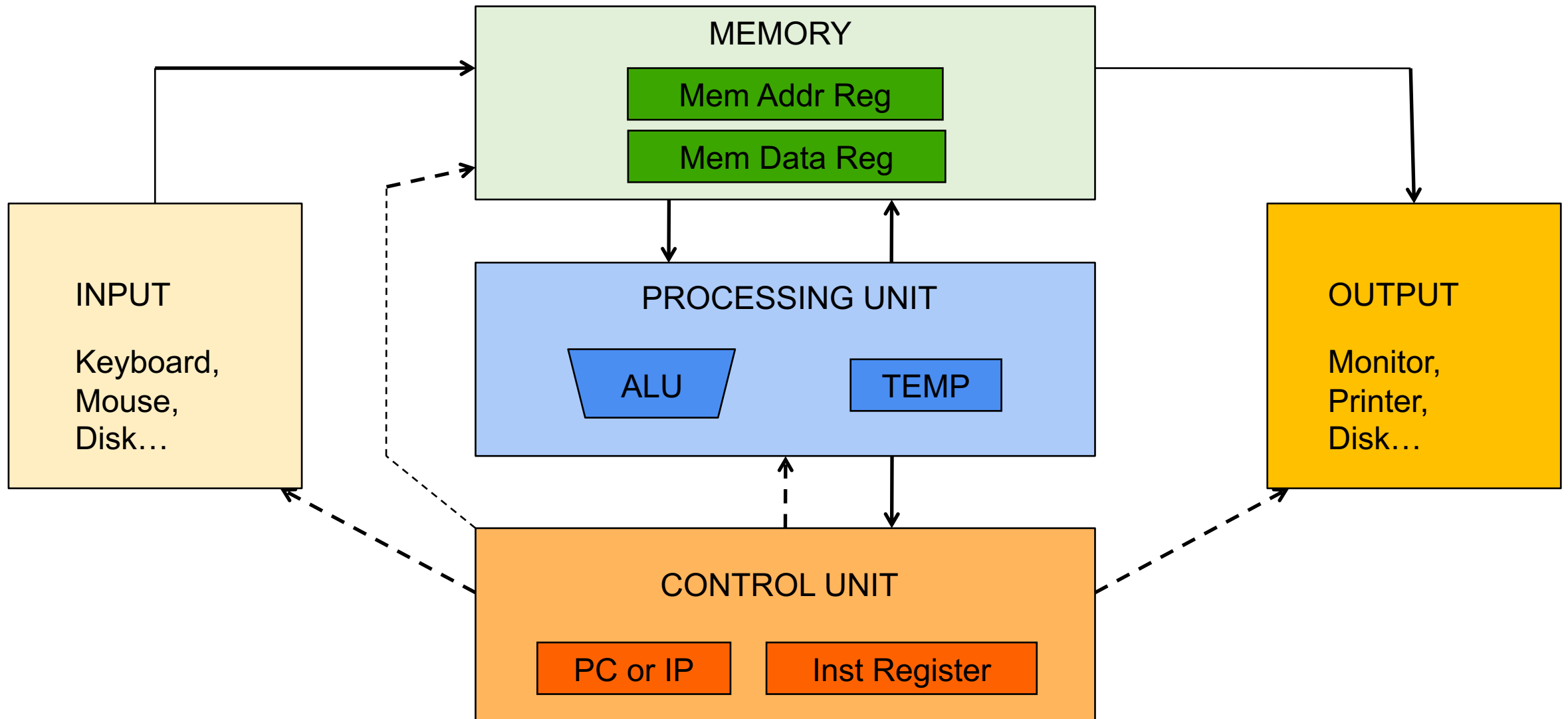
Hardware Design

Lecture 2: Instruction Set Architectures II

Dr. Haiyu Mao

29.01.2026

Recall: The von Neumann Model



Recall: The von Neumann Model

Stored program

Sequential instruction processing

Recall: LC-3 (Little Computer 3): A von Neumann Machine

- ❑ A simple educational 16-bit computer architecture

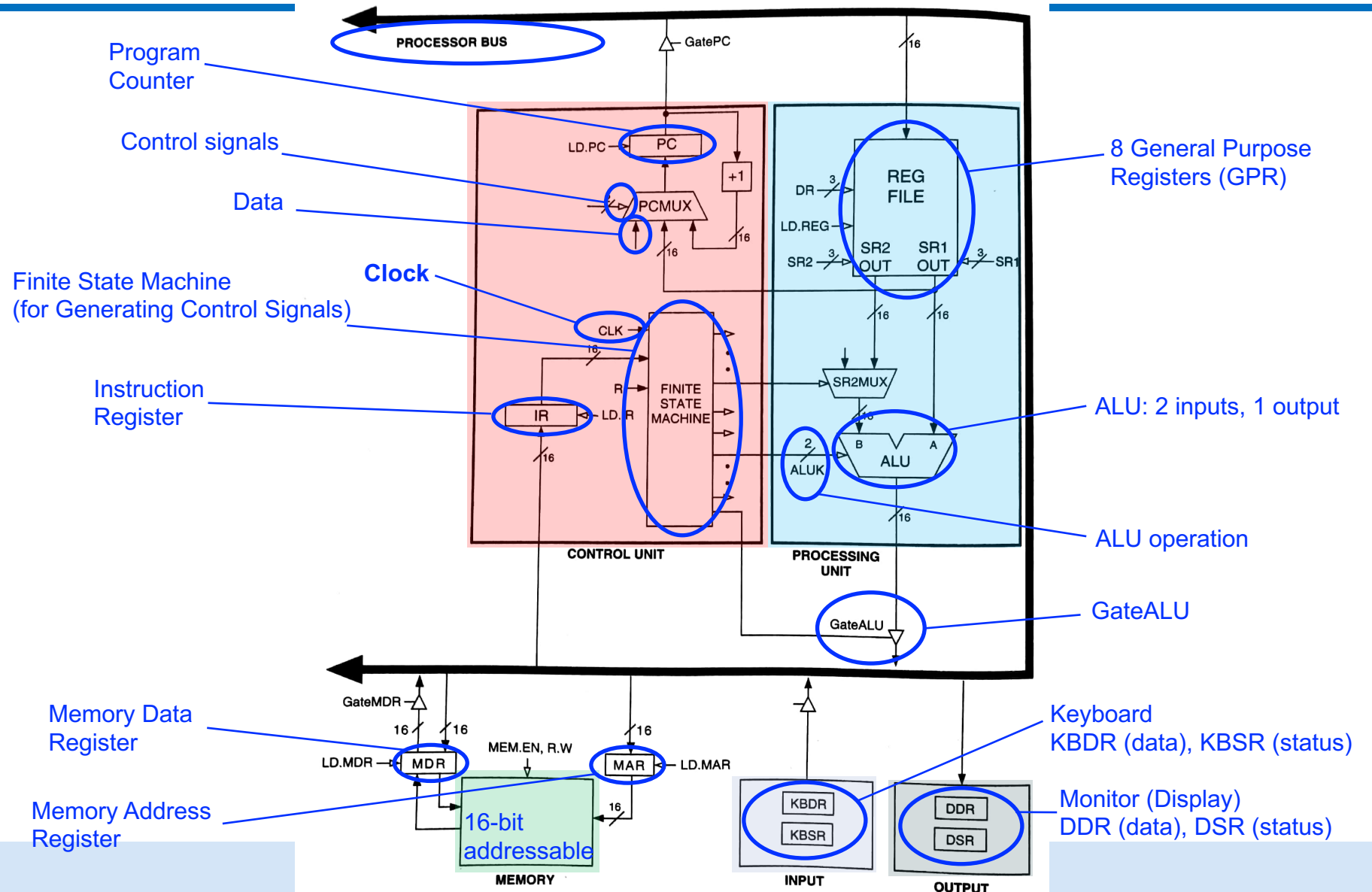
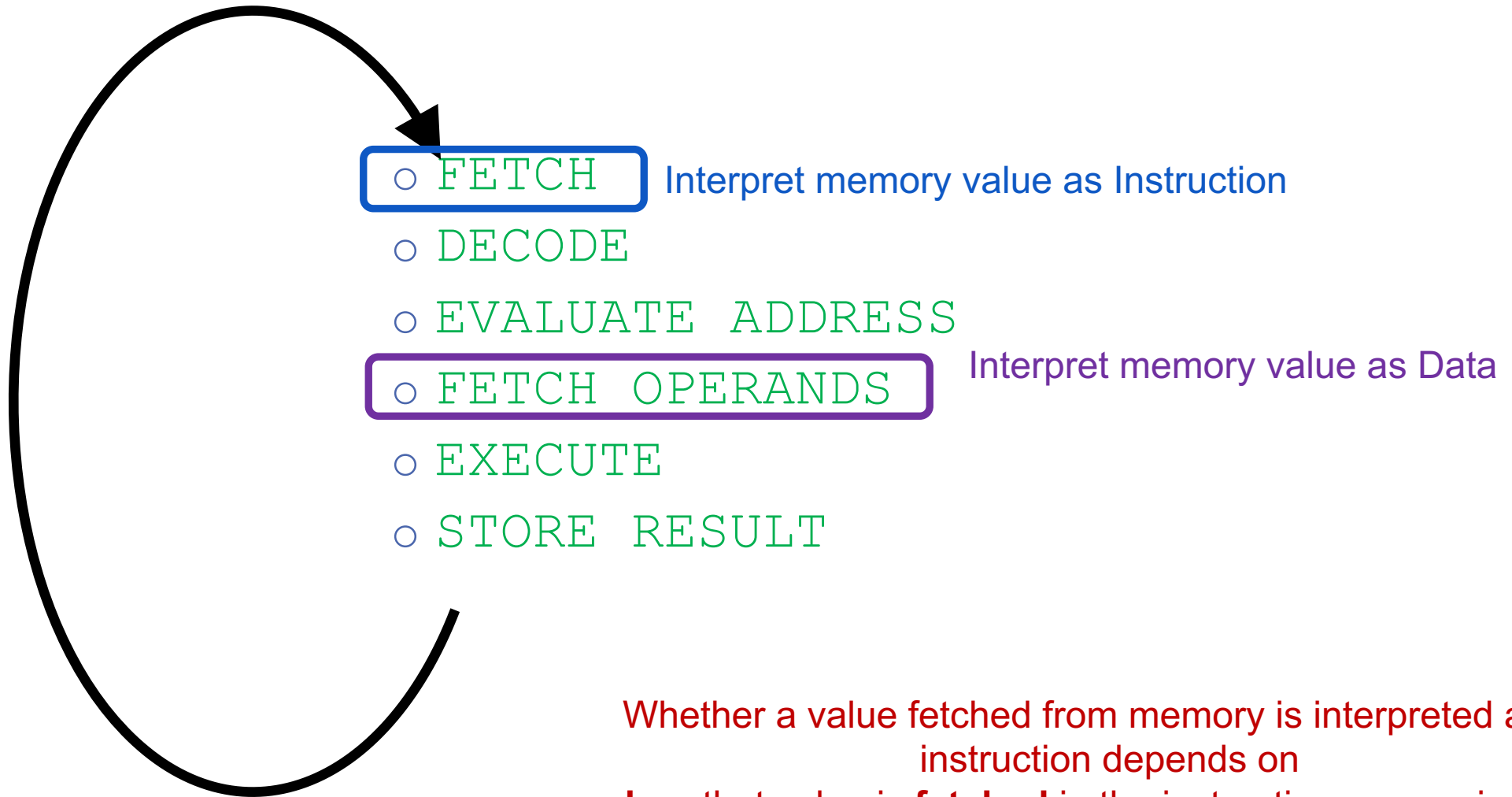


Figure 4.3 The LC-3 as an example of the von Neumann model

Recall: Instruction Types

- ❑ There are **three main types of instructions**
- ❑ Operate instructions
 - Execute operations in the ALU
- ❑ Data movement instructions
 - Read from or write to memory
- ❑ Control flow instructions
 - Change the sequence of execution

Recall: The Instruction Cycle



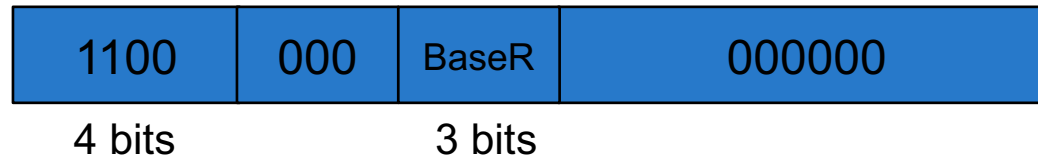
Whether a value fetched from memory is interpreted as an instruction depends on **when** that value is **fetched** in the instruction processing cycle

Recall: Jump in LC-3

□ Unconditional branch or jump

□ LC-3

JMP R2



- BaseR = Base register
- $PC \leftarrow R2$ (Register identified by BaseR)
- Variations
 - RET: special case of JMP where BaseR = R7
 - JSR, JSRR: jump to subroutine

This is register
addressing mode

Recall: PC UPDATE in LC-3

JMP loads **SR1** into PC

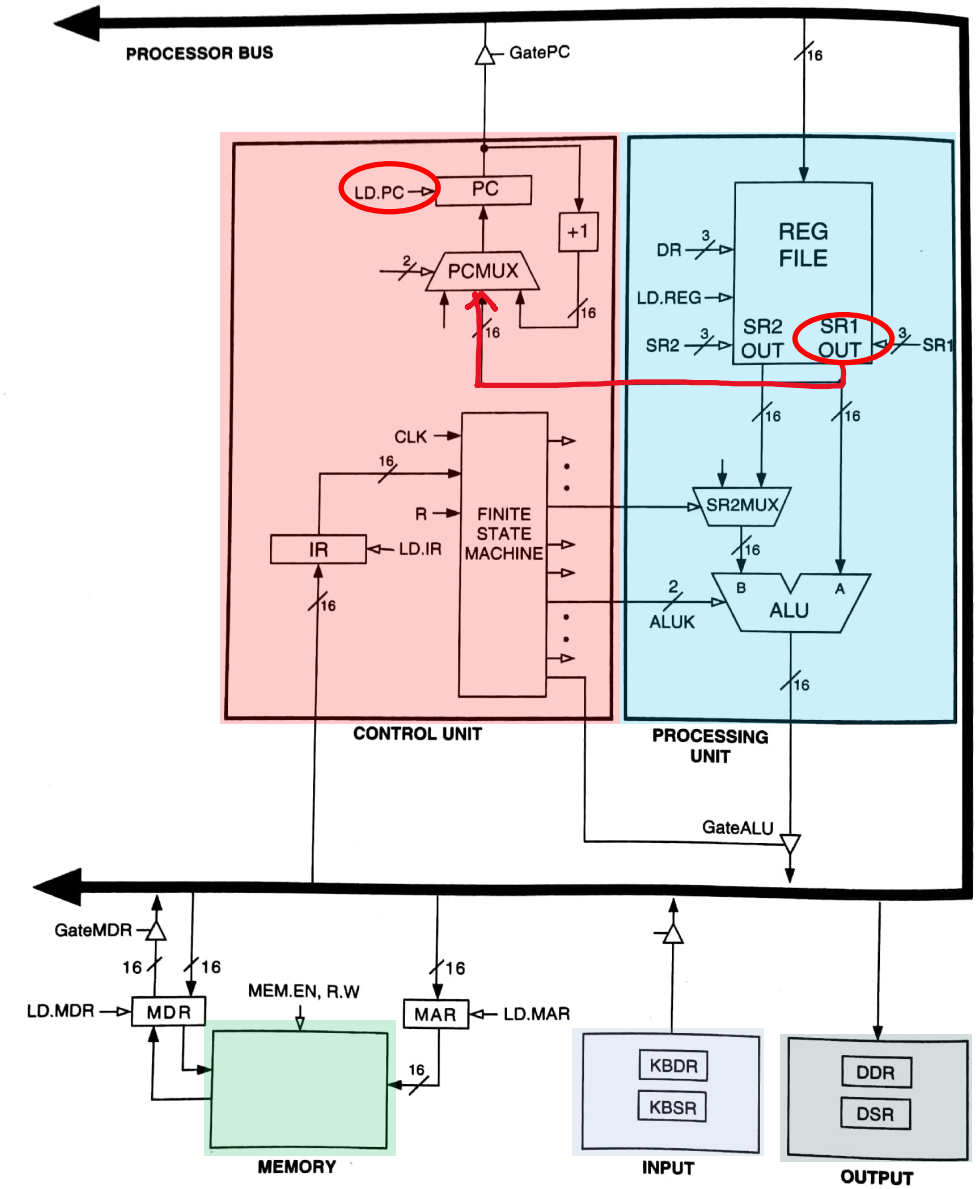
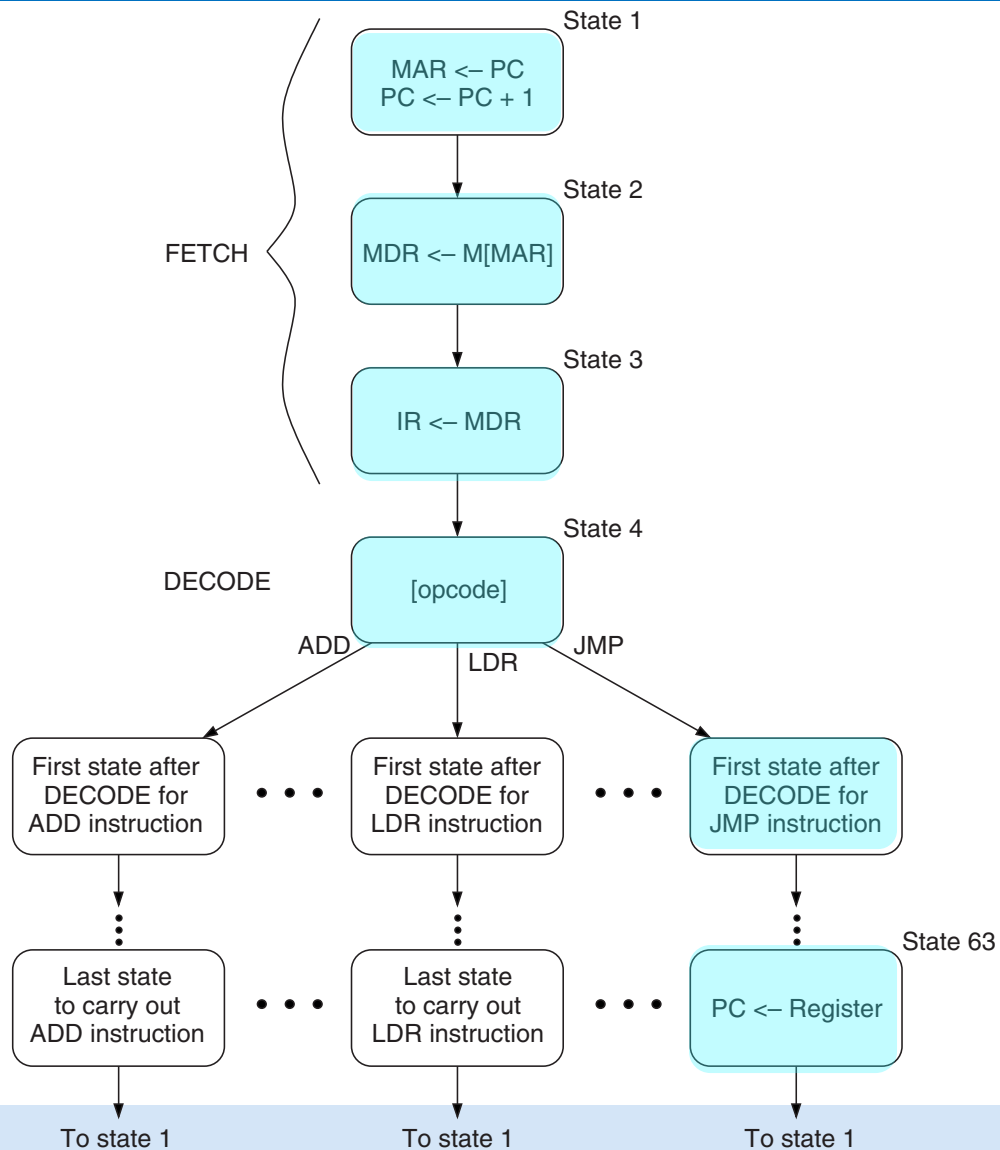


Figure 4.3 The LC-3 as an example of the von Neumann model

Control of the Instruction Cycle



- ❑ **State 1**
 - The FSM asserts GatePC and LD.MAR
 - It selects input (+1) in PCMUX and asserts LD.PC
- ❑ **State 2**
 - MDR is loaded with the instruction
- ❑ **State 3**
 - The FSM asserts GateMDR and LD.IR
- ❑ **State 4**
 - The FSM goes to the next state depending on opcode
- ❑ **State 63**
 - JMP loads register into PC
- ❑ Full state diagram in Patt&Pattel, Appendix C

To Come: Full State Machine for LC-3b

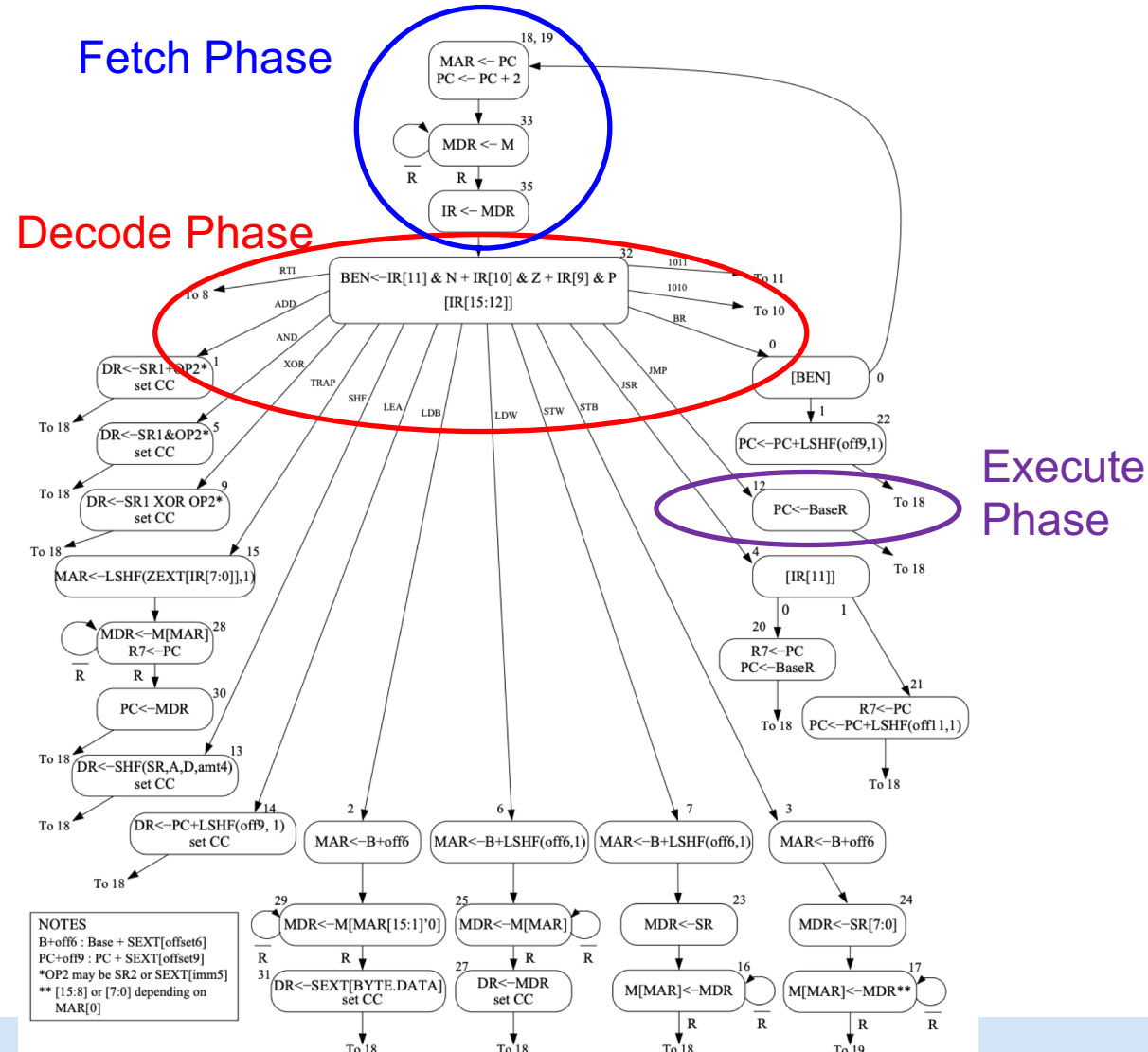
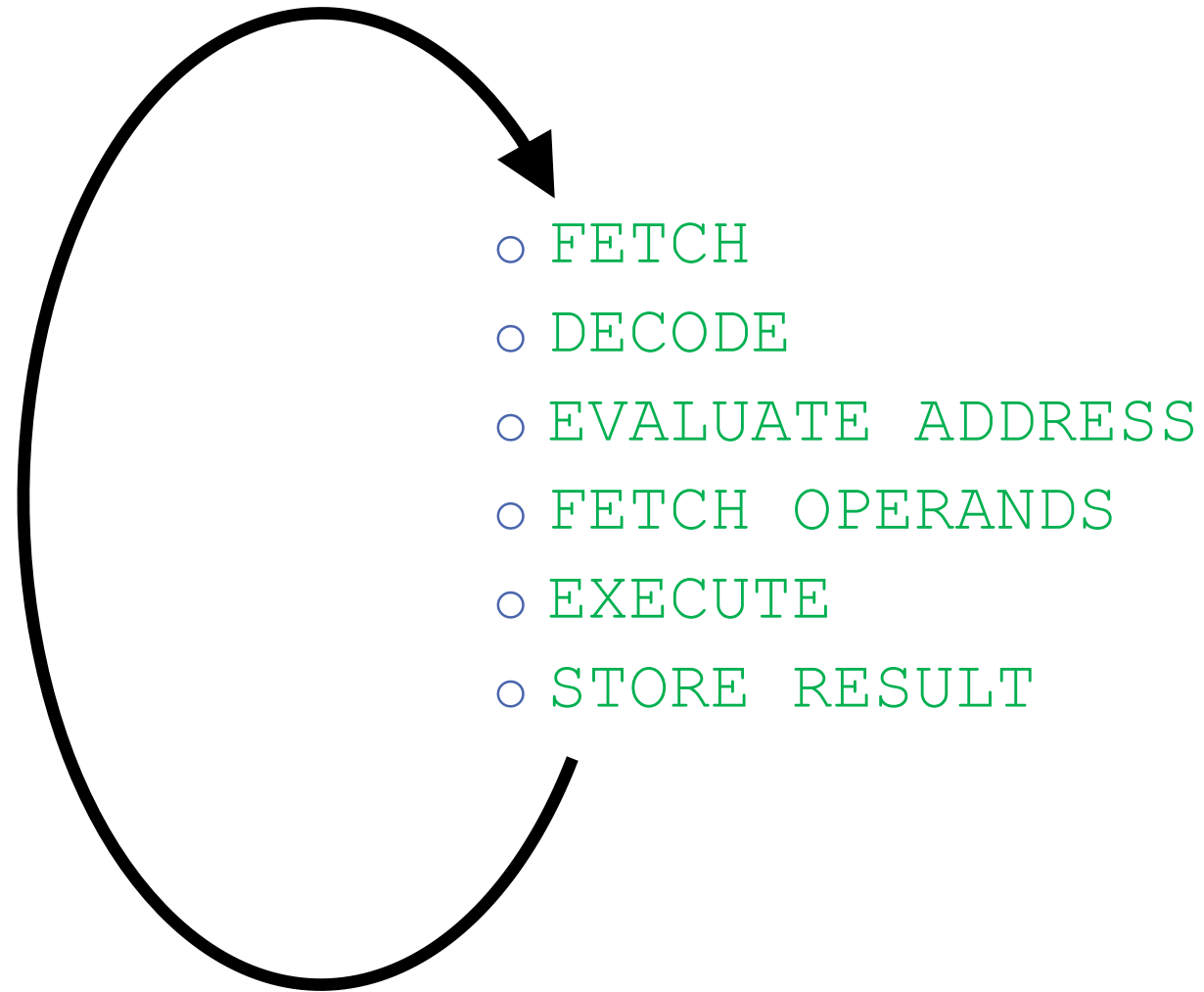


Figure C.2: A state machine for the LC-3b

The Instruction Cycle



Hardware Design

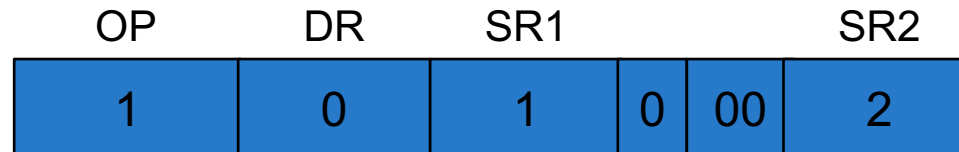
LC-3 and MIPS Instruction Set Architectures

	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		
J:	op	target address				

The Instruction Set

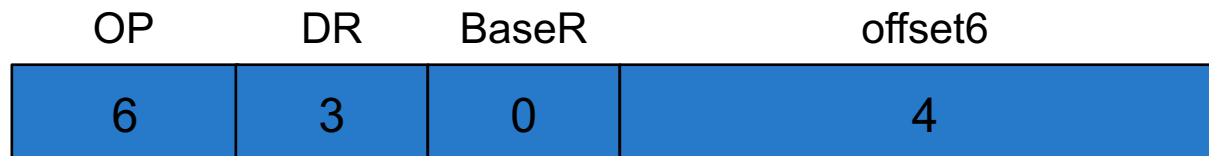
- ❑ It defines **opcodes**, **data types**, and **addressing modes**
- ❑ ADD and LDR have been our first examples

ADD



Register mode

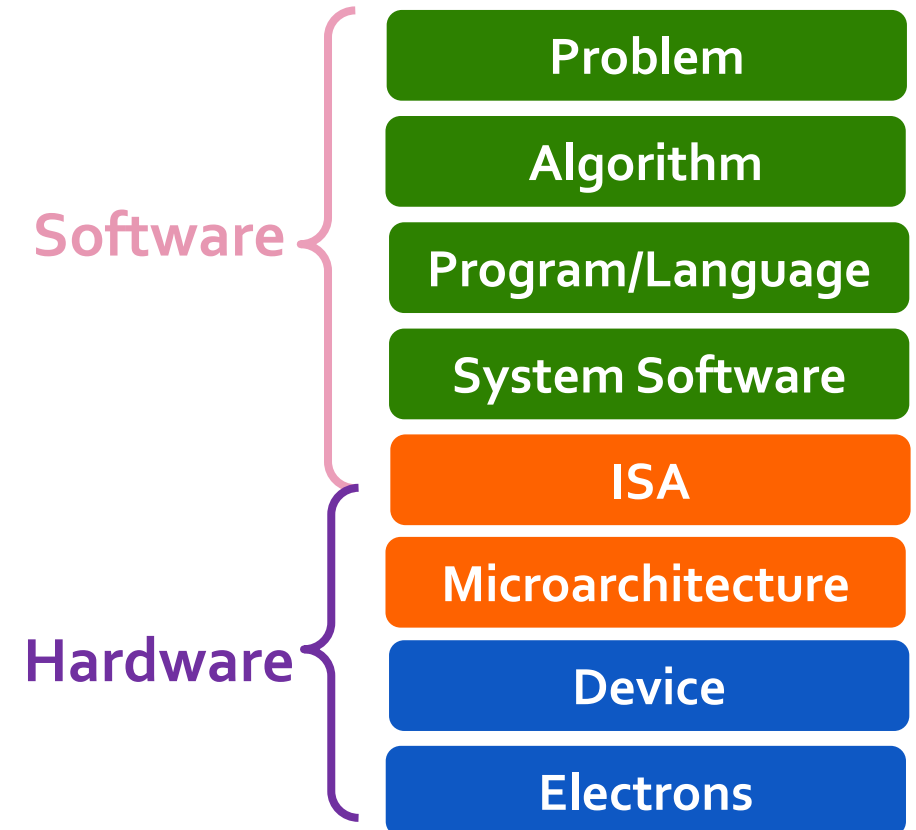
LDR



Base+offset mode

The Instruction Set Architecture

- ❑ The ISA is the **interface between** what the **software** commands and what the **hardware** carries out
- ❑ The ISA specifies
 - The **memory organization**
 - Address space (LC-3: 2^{16} , MIPS: 2^{32})
 - Addressability (LC-3: 16 bits, MIPS: 8 bits)
 - Word- or Byte-addressable
 - The **register set**
 - 8 registers (R0 to R7) in LC-3
 - 32 registers in MIPS
 - The **instruction set**
 - Opcodes
 - Data types
 - Addressing modes
 - Length and format of instructions



Hardware Design

Instructions (Opcodes)

0	rs	rt	rd	shamt	funct
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

R-type

opcode	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

opcode	immediate
6-bit	26-bit

J-type

Opcodes

- ❑ A large or small **set of opcodes** could be defined
 - E.g, HP Precision Architecture: an instruction for $A*B+C$
 - E.g, x86 ISA: multimedia extensions (MMX), later SSE, AVX, AMX
 - E.g, VAX ISA: opcode to save all information of one program before switching to another program
- ❑ **Tradeoffs** are involved. Examples:
 - Hardware complexity vs. software complexity
 - Latency of simple vs. complex instructions
- ❑ In LC-3 and in MIPS, there are three **types of opcodes**
 - Operate
 - Data movement
 - Control

Opcodes in LC-3

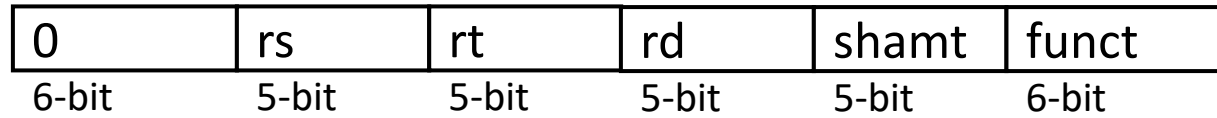
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD ⁺	0001				DR			SR1			0	00		SR2		
ADD ⁺	0001				DR			SR1			1	imm5				
AND ⁺	0101				DR			SR1			0	00		SR2		
AND ⁺	0101				DR			SR1			1	imm5				
BR	0000				n	z	p	PCoffset9								
JMP	1100				000			BaseR			000000					
JSR	0100				1	PCoffset11										
JSRR	0100				0	00		BaseR			000000					
LD ⁺	0010				DR			PCoffset9								
LDI ⁺	1010				DR			PCoffset9								
LDR ⁺	0110				DR			BaseR			offset6					
LEA ⁺	1110				DR			PCoffset9								
NOT ⁺	1001				DR			SR			111111					
RET	1100				000			111			000000					
RTI	1000				000000000000											
ST	0011				SR			PCoffset9								
STI	1011				SR			PCoffset9								
STR	0111				SR			BaseR			offset6					
TRAP	1111				0000			trapvect8								
reserved	1101															

Figure 5.3 Formats of the entire LC-3 instruction set. NOTE: + indicates instructions that modify condition codes

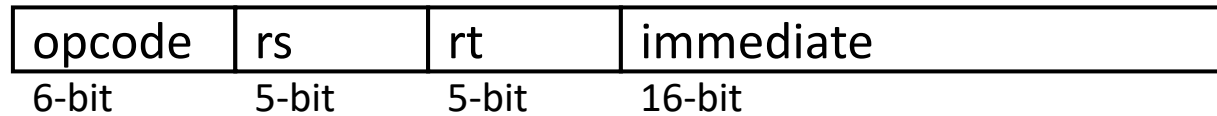
Opcodes in LC-3b

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD ⁺	0001				DR			SR1			A	op.spec				
AND ⁺	0101				DR			SR1			A	op.spec				
BR	0000				n	z	p	PCOffset9								
JMP	1100				000			BaseR			000000					
JSR(R)	0100				A	operand.specifier										
LDB ⁺	0010				DR			BaseR			boffset6					
LDW ⁺	0110				DR			BaseR			offset6					
LEA ⁺	1110				DR			PCOffset9								
RTI	1000				000000000000											
SHF ⁺	1101				DR			SR			A	D	amount4			
STB	0011				SR			BaseR			boffset6					
STW	0111				SR			BaseR			offset6					
TRAP	1111				0000			trapvect8								
XOR ⁺	1001				DR			SR1			A	op.spec				
not used	1010															
not used	1011															

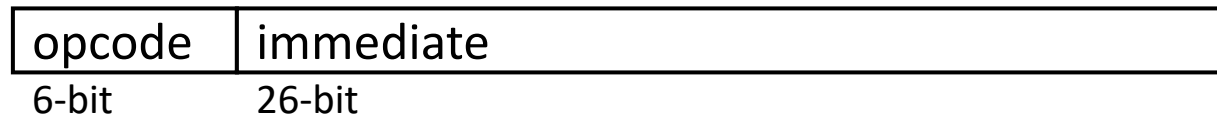
MIPS Instruction Types



R-type



I-type



J-type

Funct in MIPS R-Type Instructions (I)

Opcode is 0 in
MIPS R-Type instructions.

Funct defines the operation

Table B.2 R-type instructions, sorted by funct field

Funct	Name	Description	Operation
000000 (0)	sll rd, rt, shamt	shift left logical	[rd] = [rt] << shamt
000010 (2)	srl rd, rt, shamt	shift right logical	[rd] = [rt] >> shamt
000011 (3)	sra rd, rt, shamt	shift right arithmetic	[rd] = [rt] >>> shamt
000100 (4)	sllv rd, rt, rs	shift left logical variable	[rd] = [rt] << [rs] _{4:0}
000110 (6)	srlv rd, rt, rs	shift right logical variable	[rd] = [rt] >> [rs] _{4:0}
000111 (7)	srav rd, rt, rs	shift right arithmetic variable	[rd] = [rt] >>> [rs] _{4:0}
001000 (8)	jr rs	jump register	PC = [rs]
001001 (9)	jalr rs	jump and link register	\$ra = PC + 4, PC = [rs]
001100 (12)	syscall	system call	system call exception
001101 (13)	break	break	break exception
010000 (16)	mfhi rd	move from hi	[rd] = [hi]
010001 (17)	mthi rs	move to hi	[hi] = [rs]
010010 (18)	mflo rd	move from lo	[rd] = [lo]
010011 (19)	mtlo rs	move to lo	[lo] = [rs]
011000 (24)	mult rs, rt	multiply	{[hi], [lo]} = [rs] × [rt]
011001 (25)	multu rs, rt	multiply unsigned	{[hi], [lo]} = [rs] × [rt]
011010 (26)	div rs, rt	divide	[lo] = [rs]/[rt], [hi] = [rs]%[rt]
011011 (27)	divu rs, rt	divide unsigned	[lo] = [rs]/[rt], [hi] = [rs]%[rt]

(continued)

Funct in MIPS R-Type Instructions (II)

Table B.2 R-type instructions, sorted by funct field—Cont'd

Funct	Name	Description	Operation
100000 (32)	add rd, rs, rt	add	$[rd] = [rs] + [rt]$
100001 (33)	addu rd, rs, rt	add unsigned	$[rd] = [rs] + [rt]$
100010 (34)	sub rd, rs, rt	subtract	$[rd] = [rs] - [rt]$
100011 (35)	subu rd, rs, rt	subtract unsigned	$[rd] = [rs] - [rt]$
100100 (36)	and rd, rs, rt	and	$[rd] = [rs] \& [rt]$
100101 (37)	or rd, rs, rt	or	$[rd] = [rs] \mid [rt]$
100110 (38)	xor rd, rs, rt	xor	$[rd] = [rs] \wedge [rt]$
100111 (39)	nor rd, rs, rt	nor	$[rd] = \sim([rs] \mid [rt])$
101010 (42)	slt rd, rs, rt	set less than	$[rs] < [rt] ? [rd] = 1 : [rd] = 0$
101011 (43)	sltu rd, rs, rt	set less than unsigned	$[rs] < [rt] ? [rd] = 1 : [rd] = 0$

❑ More complete list of instructions is in H&H Appendix B

Hardware Design

Data Types

0	rs	rt	rd	shamt	funct
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

R-type

opcode	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

opcode	immediate
6-bit	26-bit

J-type

Data Types

- ❑ An ISA supports one or several data types
- ❑ LC-3 only supports 2's complement integers
 - Negative of a 2's complement binary value $X = \text{NOT}(X) + 1$
- ❑ MIPS supports
 - 2's complement integers
 - Unsigned integers
 - Floating point
- ❑ Tradeoffs are involved. Examples:
 - Hardware complexity vs. software complexity
 - Latency of operations on supported vs. unsupported data types

Why Have Different Data Types in ISA?

- ❑ An example of a programmer vs. microarchitect tradeoff

- ❑ Advantage of more data types:
 - Enables better mapping of high-level programming constructs to hardware
 - Hardware can directly operate on data types present in programming languages → small number of instructions and code size
 - Matrix operations vs. individual multiply/add/load/store instructions
 - Graph operations vs. individual load/store/add/... instructions

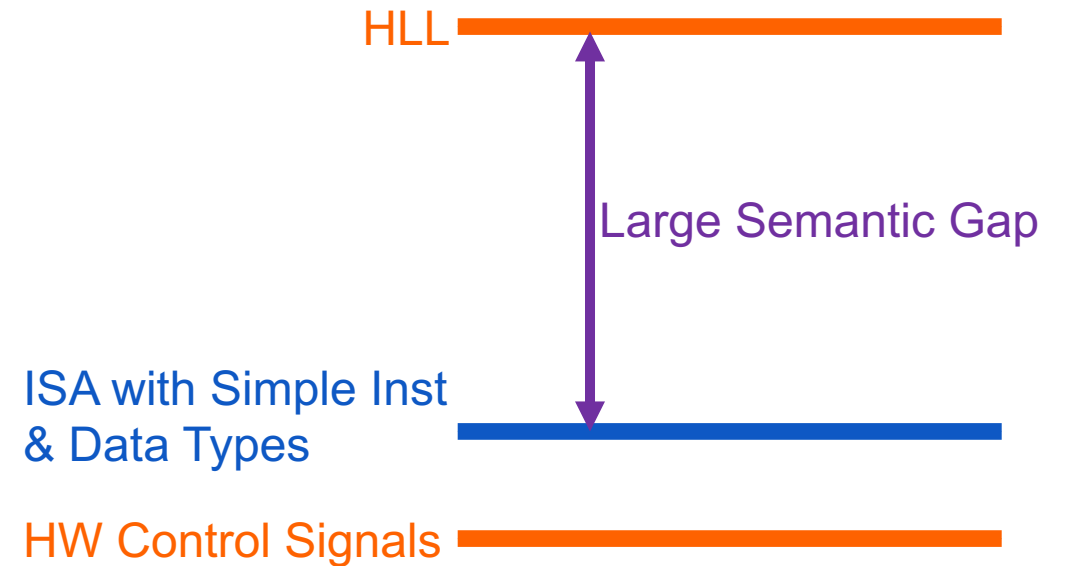
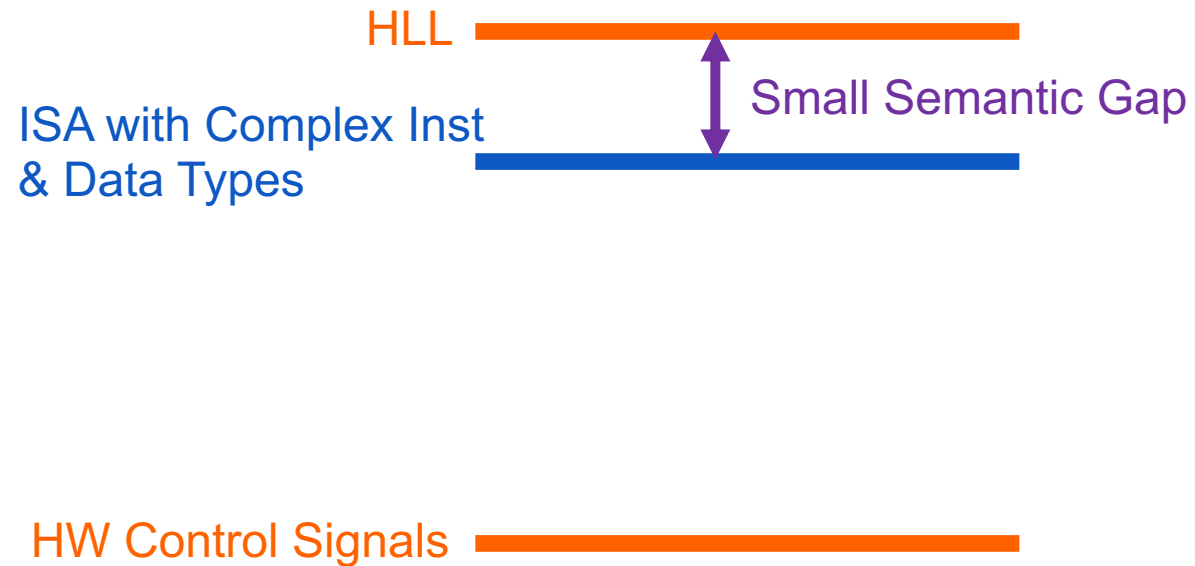
- ❑ Disadvantage:
 - More work for the microarchitect
 - Who needs to implement the data types and instructions that operate on data types

Data Types and Instruction Complexity

- ❑ Data types are coupled tightly to the semantic level, or **the complexity of instructions**
- ❑ Concept of **semantic gap**
 - How close instructions & data types are to high-level language
- ❑ **Complex instructions + data types → small semantic gap**
 - E.g., insert into a doubly linked list, multiply two matrices
 - VAX ISA: doubly-linked list, multi-dimensional arrays
- ❑ **Simple instructions + data types → large semantic gap**
 - E.g., primitive operations: load, store, multiply, add, nor
 - Early RISC machines: Only integer data type, simple operations

Semantic Gap

- How close instructions & data types are to high-level language (HLL)



Complex vs. Simple Instructions+Data Types

- ❑ **Complex instruction:** An instruction **does a lot of work**, e.g. many operations
 - Insert into a doubly linked list
 - Compute FFT
 - String copy
 - Matrix multiply
 - ...

- ❑ **Simple instruction:** An instruction **does little work** -- it is a primitive using which complex operations can be built
 - Add
 - XOR
 - Multiply
 - ...

Complex vs. Simple Instructions+Data Types

❑ Advantages of Complex Instructions + Data Types

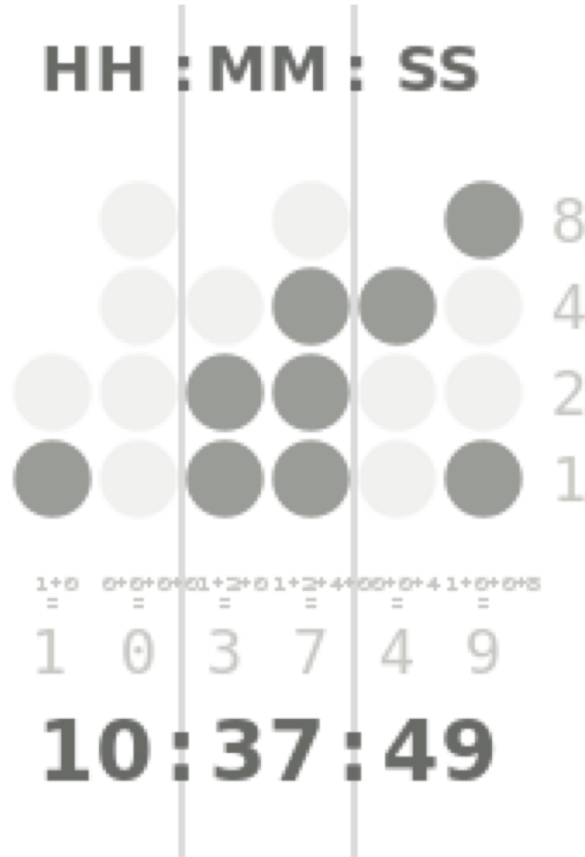
- + **Denser encoding** → smaller code size → better memory utilization, saves off-chip bandwidth, better cache hit rate (better packing of instructions)
- + **Simpler compiler**: no need to optimize small instructions as much

❑ Disadvantages of Complex Instructions + Data Types

- **Larger chunks of work** → compiler has less opportunity to optimize (limited in fine-grained optimizations it can do)
- **More complex hardware** → translation from a high level to control signals and optimization needs to be done by hardware

Aside: An Example: BinaryCodedDecimal

- ❑ Each decimal digit is encoded with a fixed number of bits



"Binary clock" by Alexander Jones & Eric Pierce - Own work, based on Wapcaplet's Binary clock.png on the English Wikipedia. Licensed under CC BY-SA 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Binary_clock.svg#mediaviewer/File:Binary_clock.svg

"Digital-BCD-clock" by Julo - Own work. Licensed under Public Domain via Wikimedia Commons - <http://commons.wikimedia.org/wiki/File:Digital-BCD-clock.jpg#mediaviewer/File:Digital-BCD-clock.jpg>

Aside: An Example: BinaryCodedDecimal

- ❑ Each decimal digit is encoded with a fixed number of bits



"Binary clock" by Alexander Jones & Eric Pierce - Own work, based on Wapcaplet's Binary clock.png on the English Wikipedia. Licensed under CC BY-SA 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Binary_clock.svg#mediaviewer/File:Binary_clock.svg

"Digital-BCD-clock" by Julo - Own work. Licensed under Public Domain via Wikimedia Commons - <http://commons.wikimedia.org/wiki/File:Digital-BCD-clock.jpg#mediaviewer/File:Digital-BCD-clock.jpg>

Hardware Design

Addressing Modes

0	rs	rt	rd	shamt	funct
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

R-type

opcode	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

opcode	immediate
6-bit	26-bit

J-type

Addressing Modes

- ❑ An addressing mode is a mechanism to specify where an operand is
- ❑ There are five addressing modes in LC-3
 - Immediate or literal (constant)
 - The operand is in some bits of the instruction
 - Register
 - The operand is in one of R0 to R7 registers
 - Three memory addressing modes
 - PC-relative
 - Indirect
 - Base+offset
- ❑ MIPS has pseudo-direct addressing (for j and jal), but does **not** have indirect addressing

Why Have Different Addressing Modes?

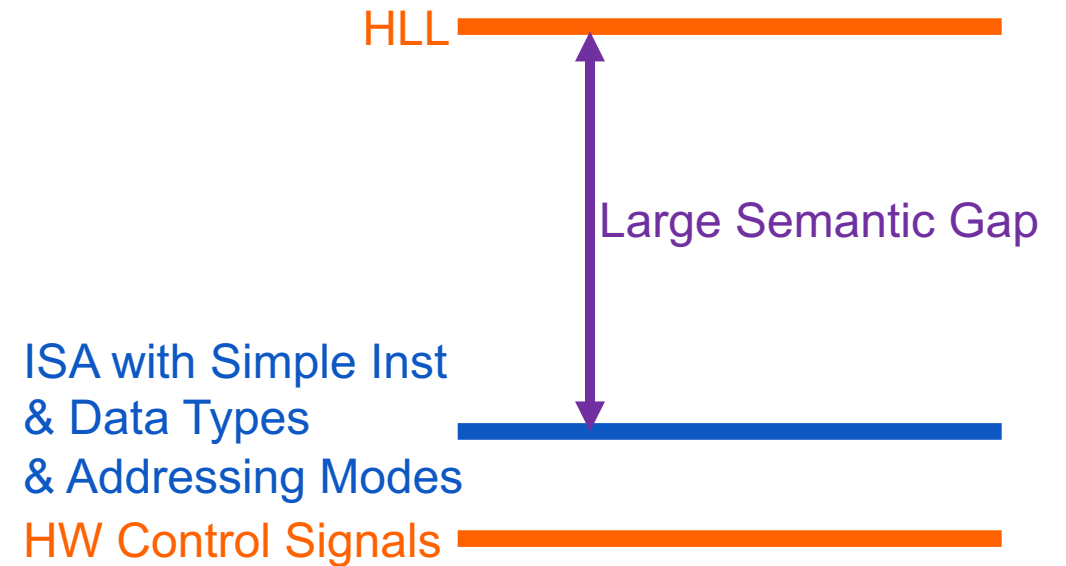
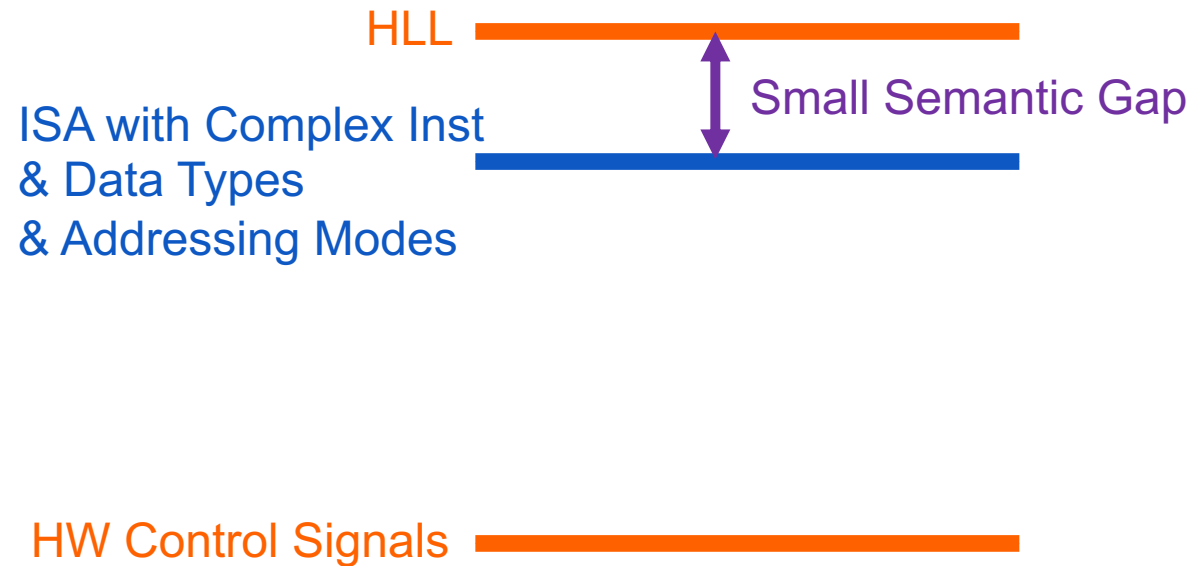
- ❑ Another example of programmer vs. microarchitect tradeoff

- ❑ Advantage of more addressing modes:
 - Enables better mapping of high-level programming constructs to hardware
 - Some accesses are better expressed with a different mode → reduced number of instructions and code size
 - Array indexing (one or multi-dimensional)
 - Pointer-based accesses (indirection)
 - Matrix element indexing; sparse matrix element indexing

- ❑ Disadvantages:
 - More work for the microarchitect
 - More options for the compiler to decide what to use

Semantic Gap Applies to Addressing Modes

- ❑ How close instructions & data types & addressing modes are to high-level language (HLL)

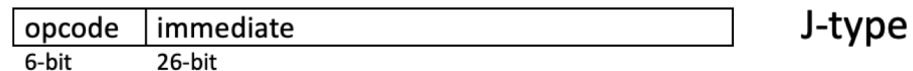
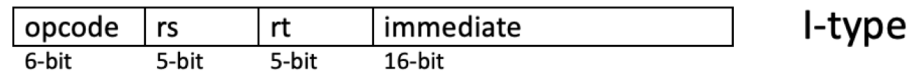
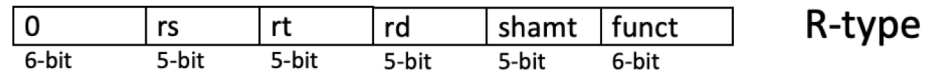


Many Tradeoffs in ISA Design...

- ☐ Execution model – sequencing model and processing style
- ☐ Instruction length
- ☐ Instruction format
- ☐ Instruction types
- ☐ Instruction complexity vs. simplicity
- ☐ Data types
- ☐ Number of registers
- ☐ Addressing mode types
- ☐ Memory organization (address space, addressability, endianness, ...)
- ☐ Memory access restrictions and permissions
- ☐ Support for multiple instructions to execute in parallel?
- ☐ ...

Hardware Design

Operate Instructions



Operate Instructions

- ❑ In **LC-3**, there are three operate instructions
 - NOT is a **unary operation** (one source operand)
 - It executes bitwise NOT
 - ADD and AND are **binary operations** (two source operands)
 - ADD is 2's complement addition
 - AND is bitwise SR1 & SR2

- ❑ In **MIPS**, there are many more
 - Many **R-type** instructions (they are **binary operations**)
 - E.g., add, and, nor, xor...
 - **I-type** versions (i.e., with one immediate operand) of the R-type operate instructions
 - **F-type** operations, i.e., floating-point operations

NOT in LC-3

❑ NOT assembly and machine code

LC-3 assembly

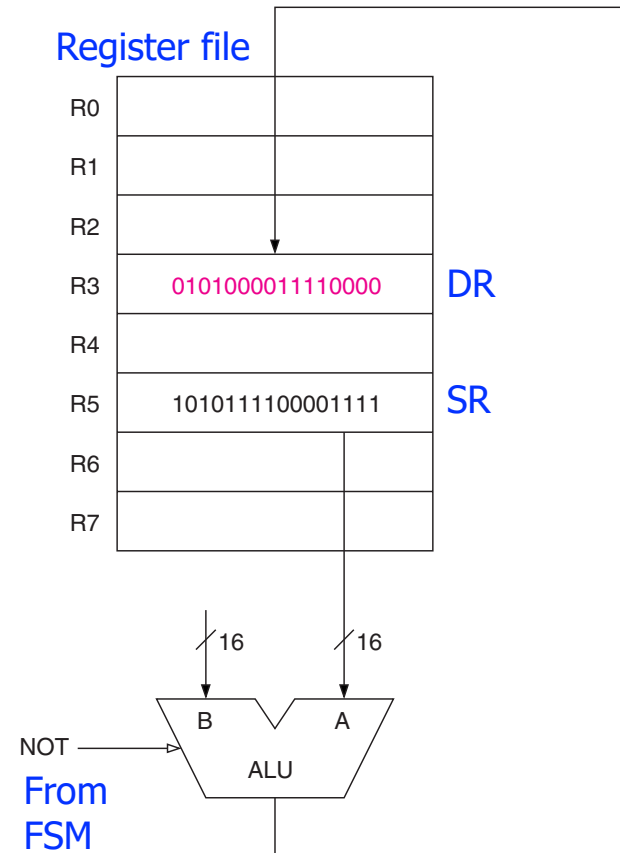
```
NOT R3, R5
```

Field Values

OP	DR	SR	
9	3	5	1 1 1 1 1 1

Machine Code

OP	DR	SR	
1 0 0 1	0 1 1	0 0 1	1 1 1 1 1 1
15	12	11 9	8 6 5 0



There is **no**
NOT in MIPS.
How is it
implemented?

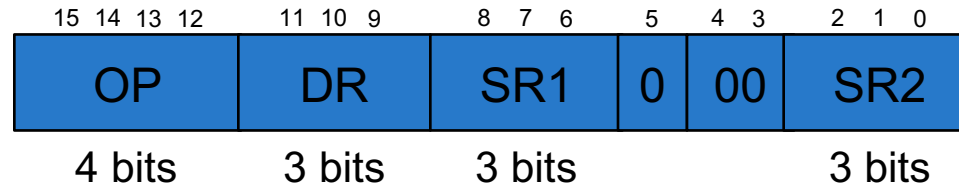
There is **no**
NOT in LC-3b.
How is it
implemented?

Operate Instructions

- We are already familiar with LC-3's ADD and AND with register mode (R-type in MIPS)
- Now, let us see the versions with one literal (i.e., immediate) operand
- We will use Subtraction as an example
 - How is it implemented in LC-3 and MIPS?

Recall: LC-3 Operate Instruction Format

□ LC-3 Operate Instruction Format (Register OP Register)



○ OP = **opcode** (what the instruction does)

▪ E.g., ADD = 0001

◦ **Semantics:** $DR \leftarrow SR1 + SR2$

▪ E.g., AND = 0101

◦ **Semantics:** $DR \leftarrow SR1 \text{ AND } SR2$

○ SR1, SR2 = source registers

○ DR = destination register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	0	0	0	1	1	0
ADD				R6			R2			R6					

Operate Instr. with one Literal in LC-3

□ ADD and AND



- OP = operation
 - E.g., **ADD** = 0001 (same OP as the register-mode ADD)
 - $DR \leftarrow SR1 + \text{sign-extend}(\text{imm5})$
 - E.g., **AND** = 0101 (same OP as the register-mode AND)
 - $DR \leftarrow SR1 \text{ AND } \text{sign-extend}(\text{imm5})$
- SR1 = source register
- DR = destination register
- **imm5** = Literal or immediate (sign-extend to 16 bits)

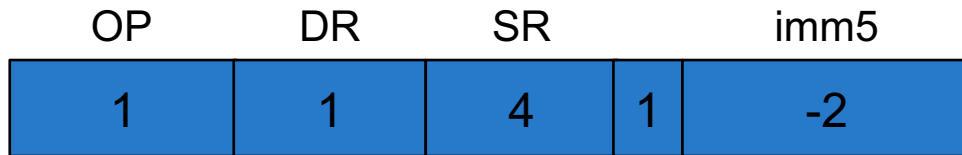
ADD with one Literal in LC-3

ADD assembly and machine code

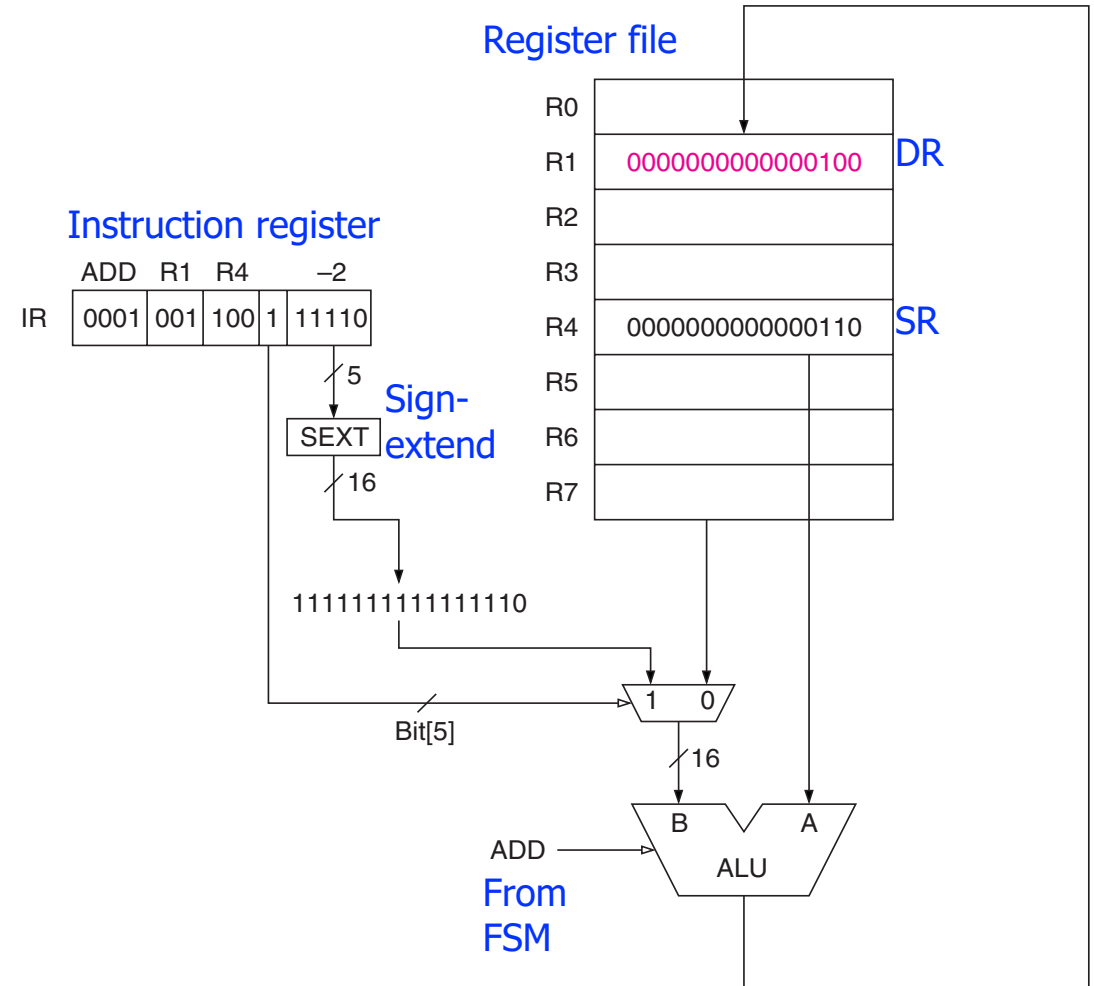
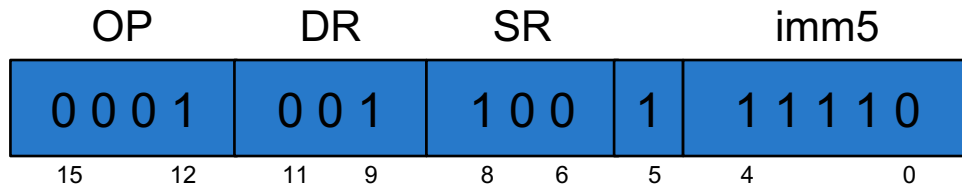
LC-3 assembly

```
ADD R1, R4, #-2
```

Field Values



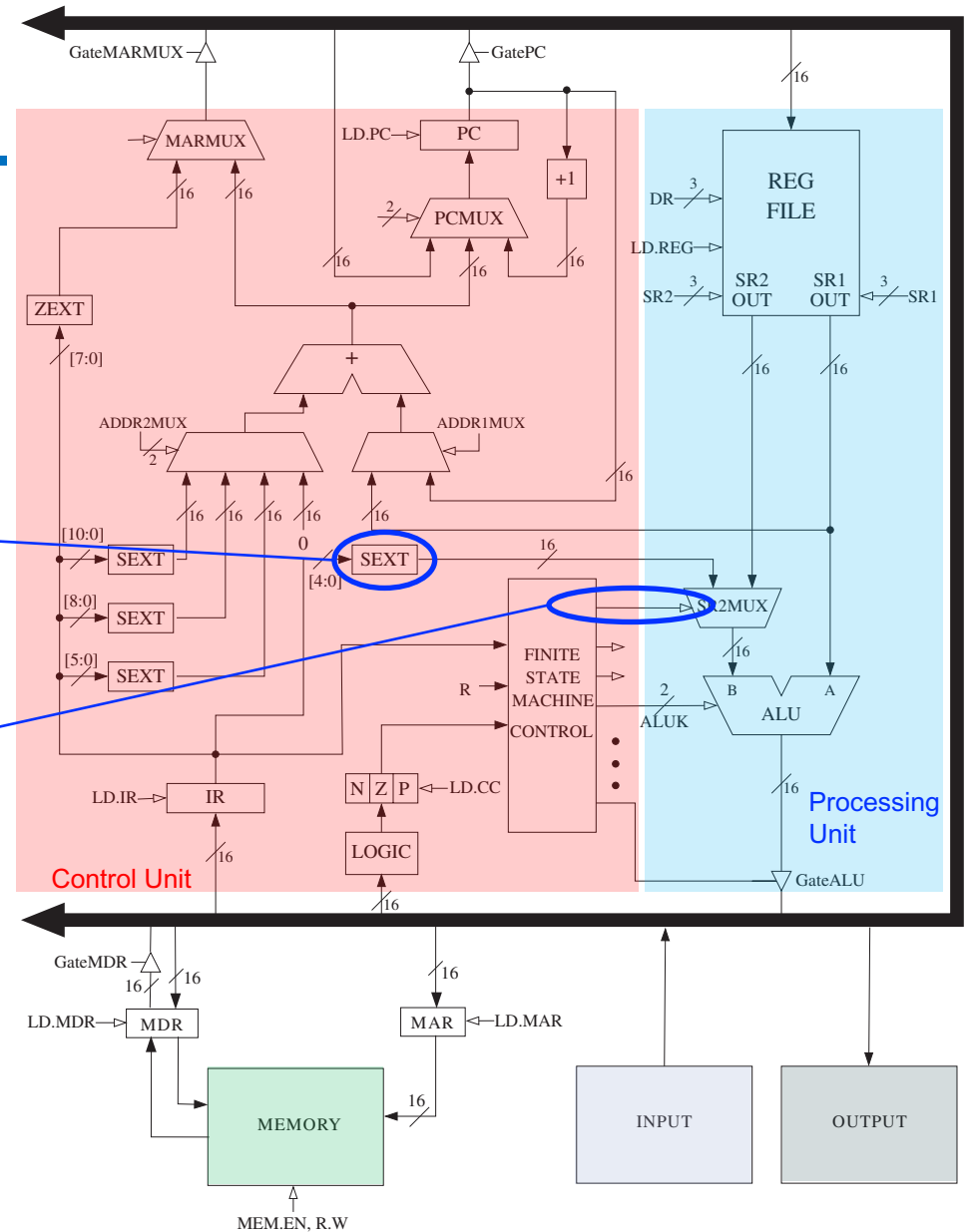
Machine Code



ADD with one Literal in LC-3 Data Path

Sign extension (Operand)

Select Immediate or Register (as the 2nd input to instruction)

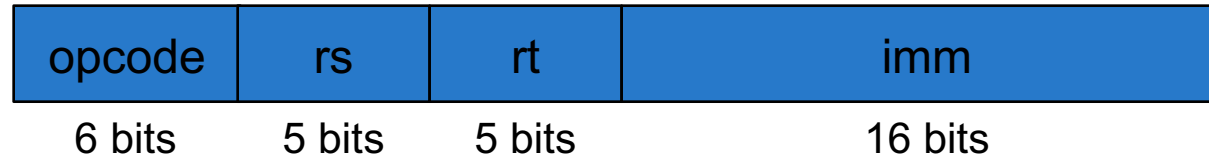


Instructions with one Literal in MIPS

❑ I-type MIPS Instructions

- 2 register operands and immediate

❑ Some operate and data movement instructions



- opcode = operation
- rs = source register
- rt =
 - destination register in some instructions (e.g., `addi`, `lw`)
 - source register in others (e.g., `sw`)
- imm = Literal or immediate

ADD with one Literal in MIPS

❑ Add immediate

MIPS assembly

```
addi $s0, $s1, 5
```

Field Values

op	rs	rt	imm
8	17	16	5

$rt \leftarrow rs + \text{sign-extend}(\text{imm})$

Machine Code

op	rs	rt	imm
001000	10001	10010	0000 0000 0000 0101

0x22300005

Subtraction in MIPS vs. LC-3

❑ MIPS assembly

High-level code

```
a = b + c - d;
```

MIPS assembly

```
add    $t0, $s0, $s1  
sub    $s3, $t0, $s2
```

❑ LC-3 assembly

High-level code

```
a = b + c - d;
```

LC-3 assembly

```
ADD    R2, R0, R1  
NOT    R4, R3  
ADD    R5, R4, #1  
ADD    R6, R2, R5
```

2's complement of R3

❑ Tradeoff in LC-3

- More instructions
- But, simpler control logic

Subtract Immediate

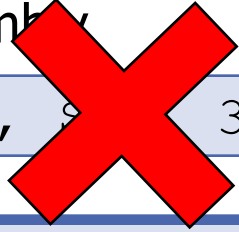
❑ MIPS assembly

High-level code

```
a = b - 3;
```

MIPS assembly

```
subi $s1, $s0, 3
```



Is **subi** necessary in MIPS?

❑ LC-3

High-level code

```
a = b - 3;
```

MIPS assembly

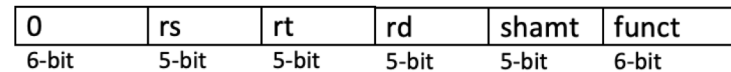
```
addi $s1, $s0, -3
```

LC-3 assembly

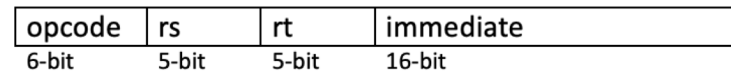
```
ADD R1, R0, #-3
```

Hardware Design

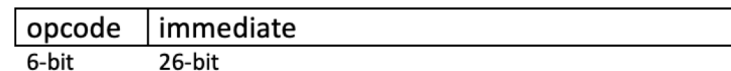
Data Movement Instructions and Addressing Modes



R-type



I-type



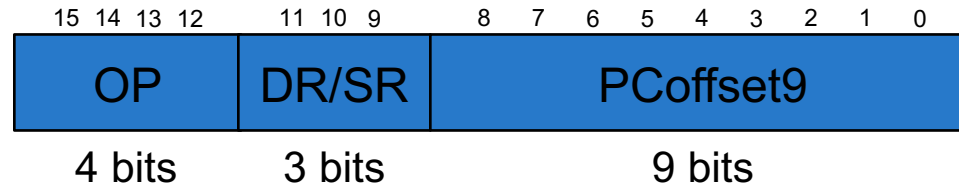
J-type

Data Movement Instructions

- ❑ In **LC-3**, there are seven data movement instructions
 - LD, LDR, LDI, LEA, ST, STR, STI
- ❑ Format of load and store instructions
 - Opcode (bits [15:12])
 - DR or SR (bits [11:9])
 - Address generation bits (bits [8:0])
 - Four ways to interpret bits, called **addressing modes**
 - PC-Relative Mode
 - Indirect Mode
 - Base+Offset Mode
 - Immediate Mode
- ❑ In **MIPS**, there are only **Base+offset** and **Immediate modes** for load and store instructions

PC-Relative Addressing Mode

LD (Load) and ST (Store)



- OP = opcode
 - E.g., LD = 0010
 - E.g., ST = 0011
- DR = destination register in LD
- SR = source register in ST
- LD: $DR \leftarrow \text{Memory}[PC^+ + \text{sign-extend}(\text{PCOffset9})]$
- ST: $\text{Memory}[PC^+ + \text{sign-extend}(\text{PCOffset9})] \leftarrow SR$

LD in LC-3

LD assembly and machine code

LC-3 assembly

```
LD R2, 0x1AF
```

Field Values

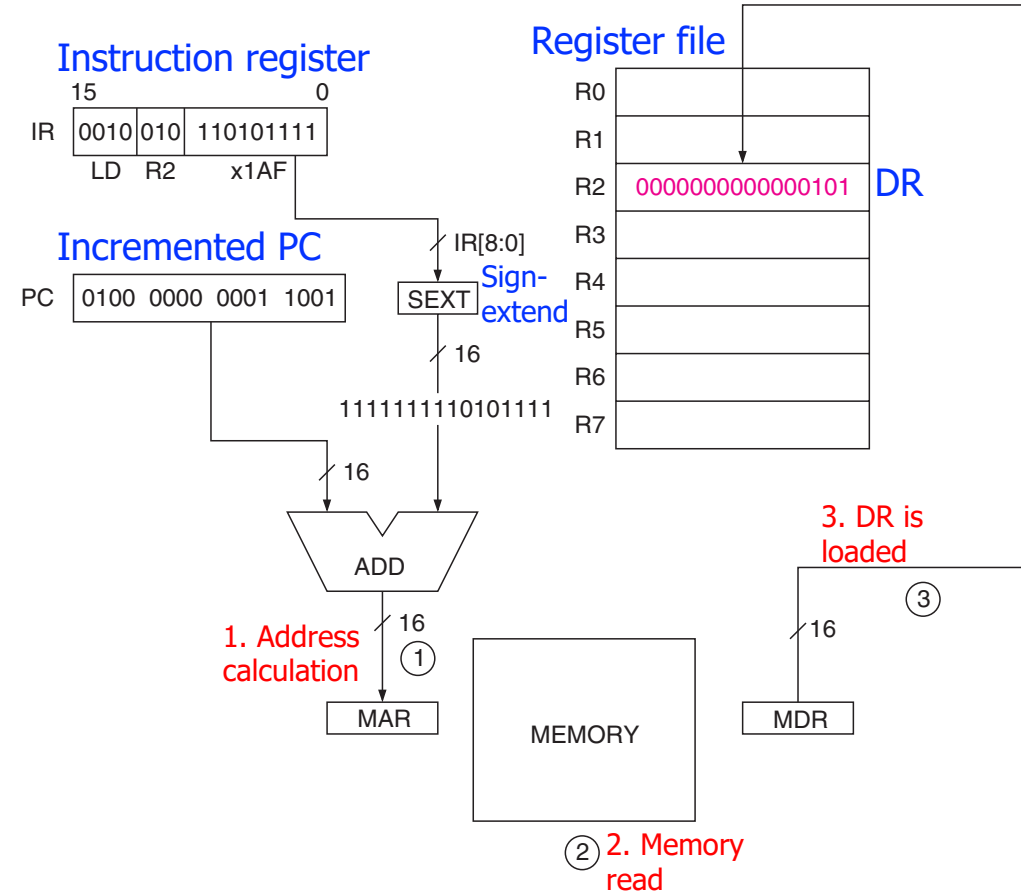
OP	DR	PCOffset9
2	2	0x1AF

Machine Code

OP	DR	PCOffset9
0010	010	110101111

15 12 11 9 8 0

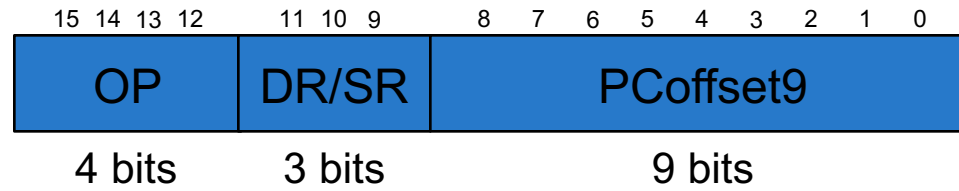
The memory address is **only +255 to -256** locations away from the **LD or ST instruction**



Limitation: The **PC-relative addressing mode** cannot address far away from the instruction

Indirect Addressing Mode

□ LDI (Load Indirect) and STI (Store Indirect)



- OP = opcode
 - E.g., LDI = 1010
 - E.g., STI = 1011
- DR = destination register in LDI
- SR = source register in STI
- LDI: $DR \leftarrow \text{Memory}[\text{Memory}[PC^+ + \text{sign-extend}(\text{PCOffset9})]]$
- STI: $\text{Memory}[\text{Memory}[PC^+ + \text{sign-extend}(\text{PCOffset9})]] \leftarrow SR$

LDI in LC-3

LDI assembly and machine code

LC-3 assembly

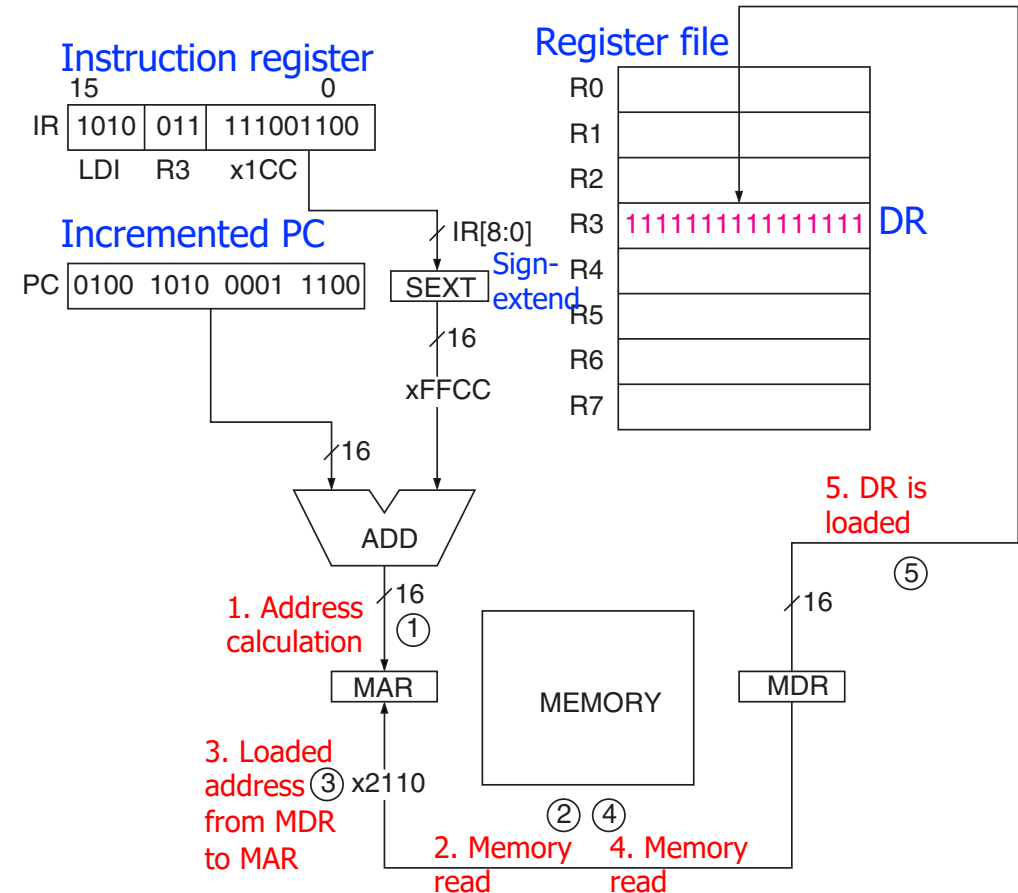
```
LDI R3, 0x1CC
```

Field Values

OP	DR	PCOffset9
A	3	0x1CC

Machine Code

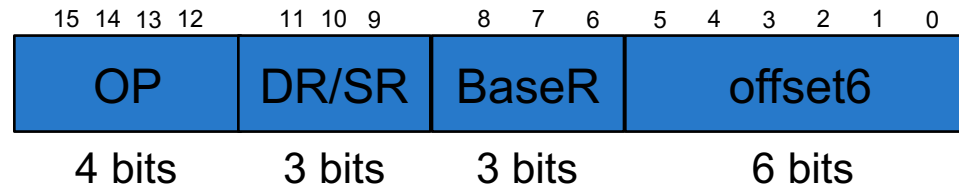
OP	DR	PCOffset9
1 0 1 0	0 1 1	1 1 1 0 0 1 1 0 0
15	12	11 9 8 0



Now the address of the operand can be **anywhere in the memory**

Base+Offset Addressing Mode

□ LDR (Load Register) and STR (Store Register)



- OP = opcode
 - E.g., LDR = 0110
 - E.g., STR = 0111
- DR = destination register in LDR
- SR = source register in STR
- LDR: $DR \leftarrow \text{Memory}[\text{BaseR} + \text{sign-extend}(\text{offset6})]$
- STR: $\text{Memory}[\text{BaseR} + \text{sign-extend}(\text{offset6})] \leftarrow SR$

LDR in LC-3

❑ LDR assembly and machine code

LC-3 assembly

```
LDR R1, R2, 0x1D
```

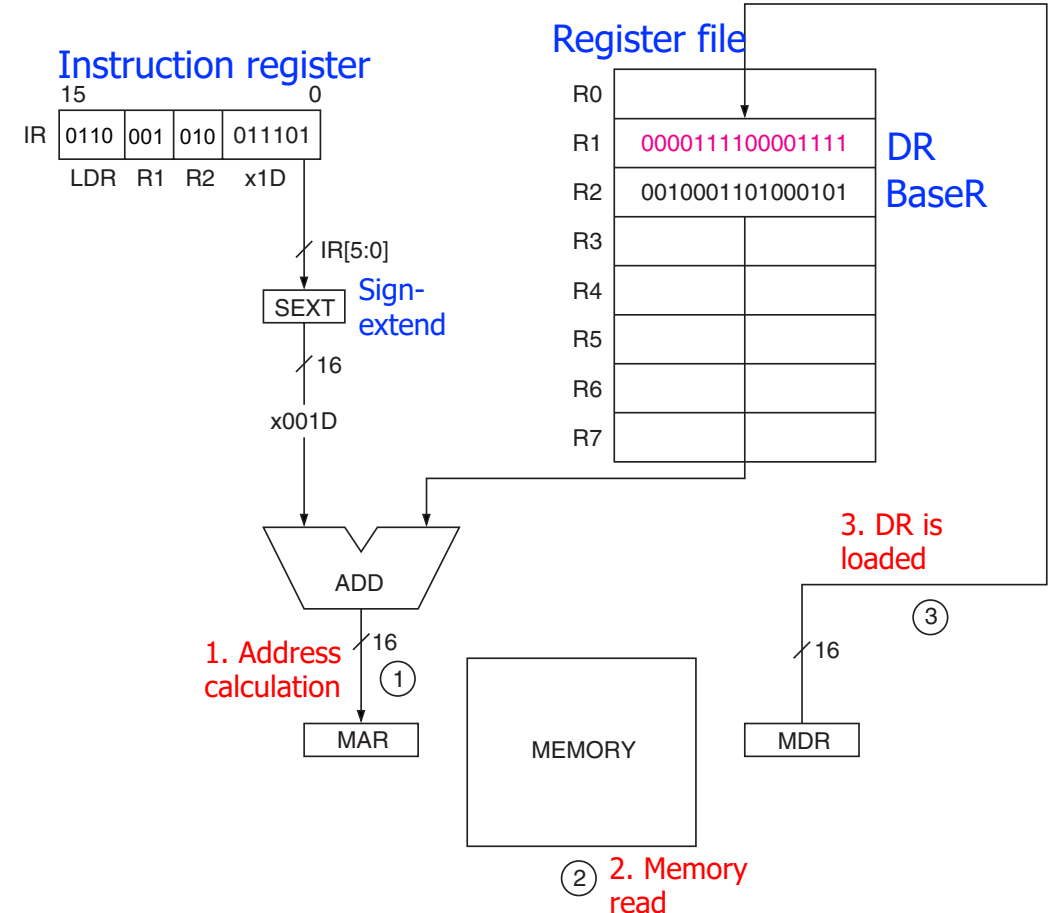
Field Values

OP	DR	BaseR	offset6
6	1	2	0x1D

Machine Code

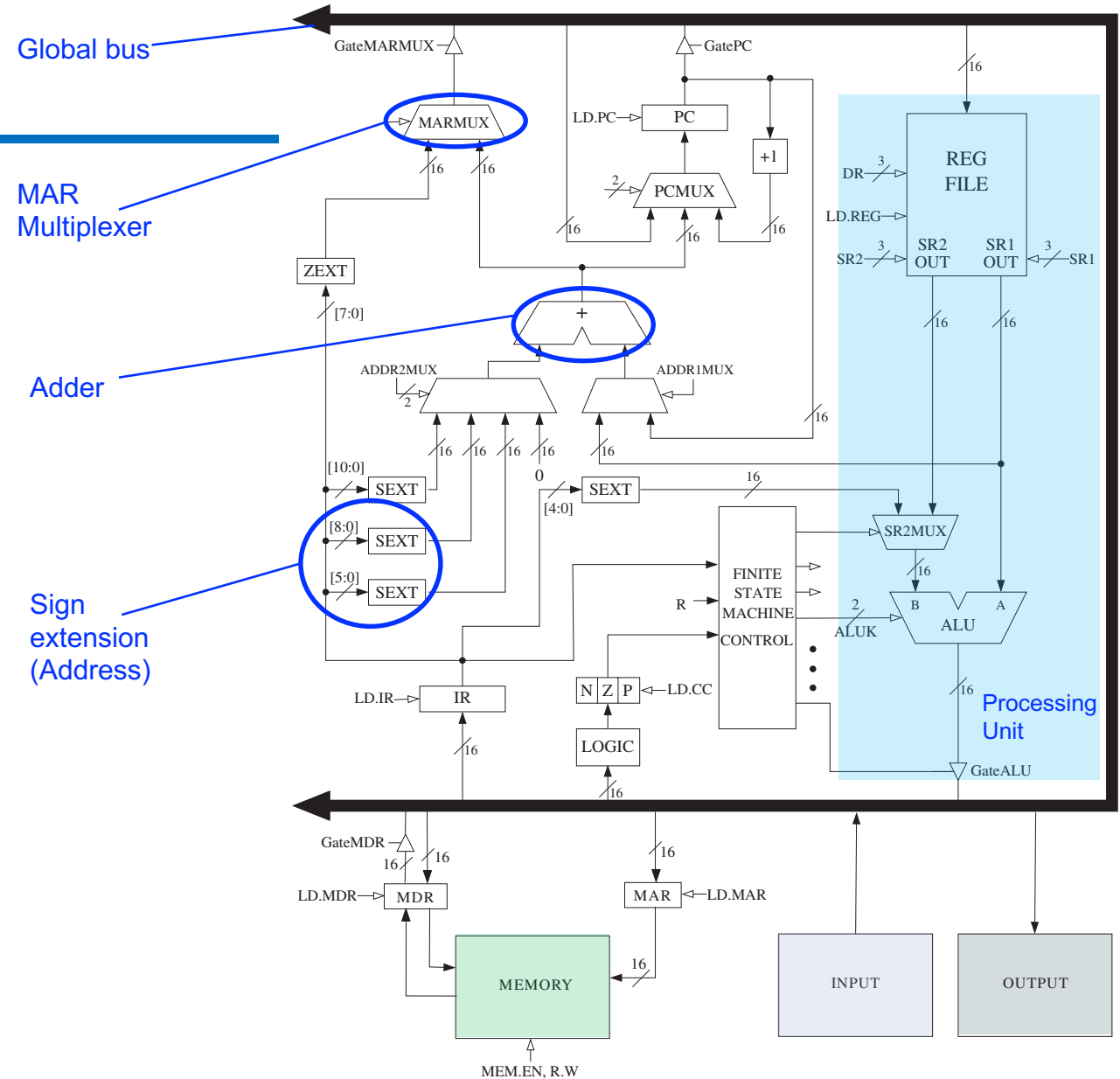
OP	DR	BaseR	offset6
0110	001	010	011101

15 12 11 9 8 6 5 0



Again, the address of the operand can be **anywhere in the memory**

Address Calculation in LC-3 Data Path



Base+Offset Addressing Mode in MIPS

- In MIPS, `lw` and `sw` use base+offset mode (or **base addressing mode**)

High-level code

```
A[2] = a;
```

MIPS assembly

```
sw    $s3, 8($s0)
```

Memory[\$s0 + 8] ← \$s3

Field Values

op	rs	rt	imm
43	16	19	8

- `imm` is the 16-bit offset, which is **sign-extended to 32 bits**

An Example Program in MIPS and LC-3

High-level code

```
a    = A[0];  
c    = a + b - 5;  
B[0] = c;
```

MIPS registers

```
A = $s0  
b = $s2  
B = $s1
```

LC-3 registers

```
A = R0  
b = R2  
B = R1
```

MIPS assembly

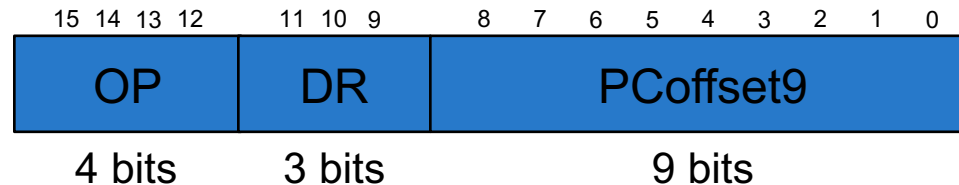
```
lw    $t0, 0($s0)  
add   $t1, $t0, $s2  
addi  $t2, $t1, -5  
sw    $t2, 0($s1)
```

LC-3 assembly

```
LDR   R5, R0, #0  
ADD   R6, R5, R2  
ADD   R7, R6, #-5  
STR   R7, R1, #0
```

Immediate Addressing Mode (in LC-3)

❑ LEA (Load Effective Address)



- OP = 1110
- DR = destination register
- LEA: $DR \leftarrow PC^{\dagger} + \text{sign-extend}(\text{PCOffset9})$

What is the **difference from PC-Relative** addressing mode?

Answer: Instructions with **PC-Relative** mode load from memory, but **LEA** does not → Hence the name *Load Effective Address*

LEA in LC-3

LEA assembly and machine code

LC-3 assembly

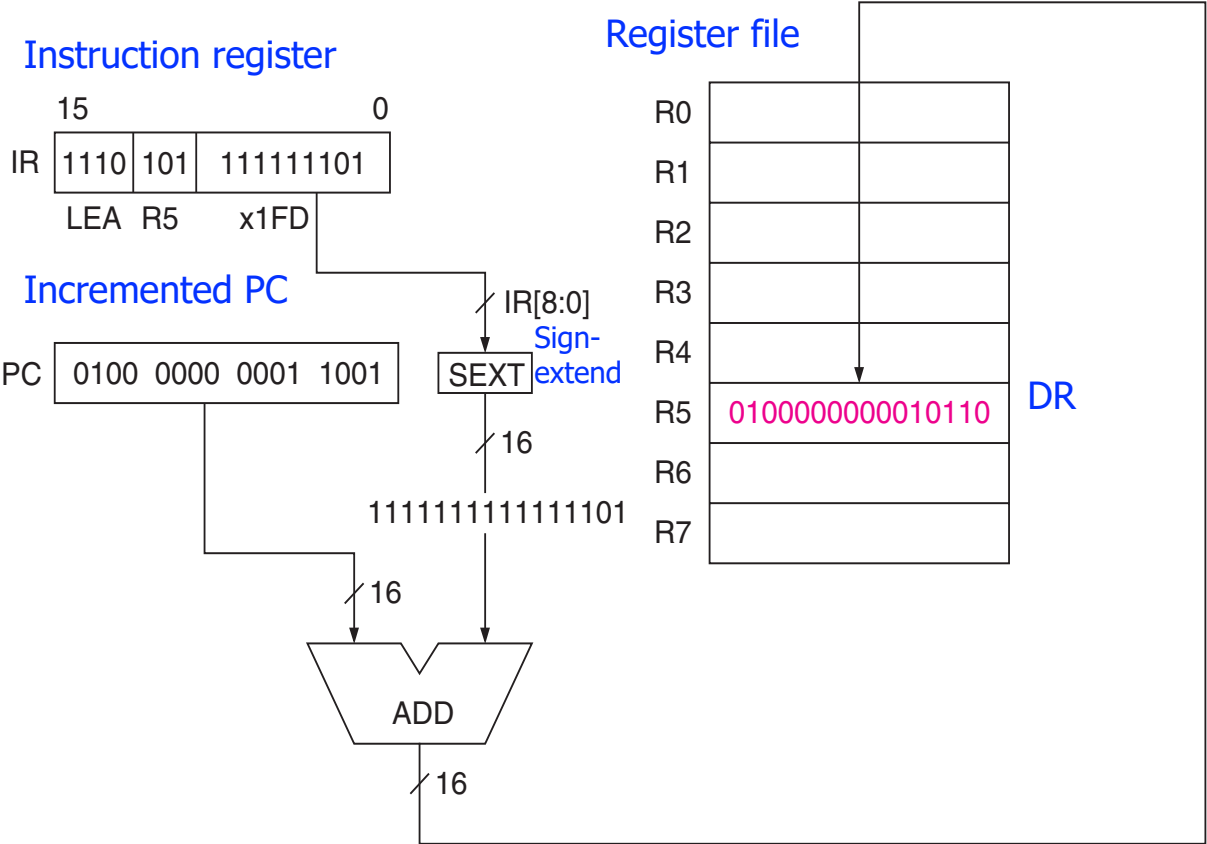
```
LEA R5, #-3
```

Field Values

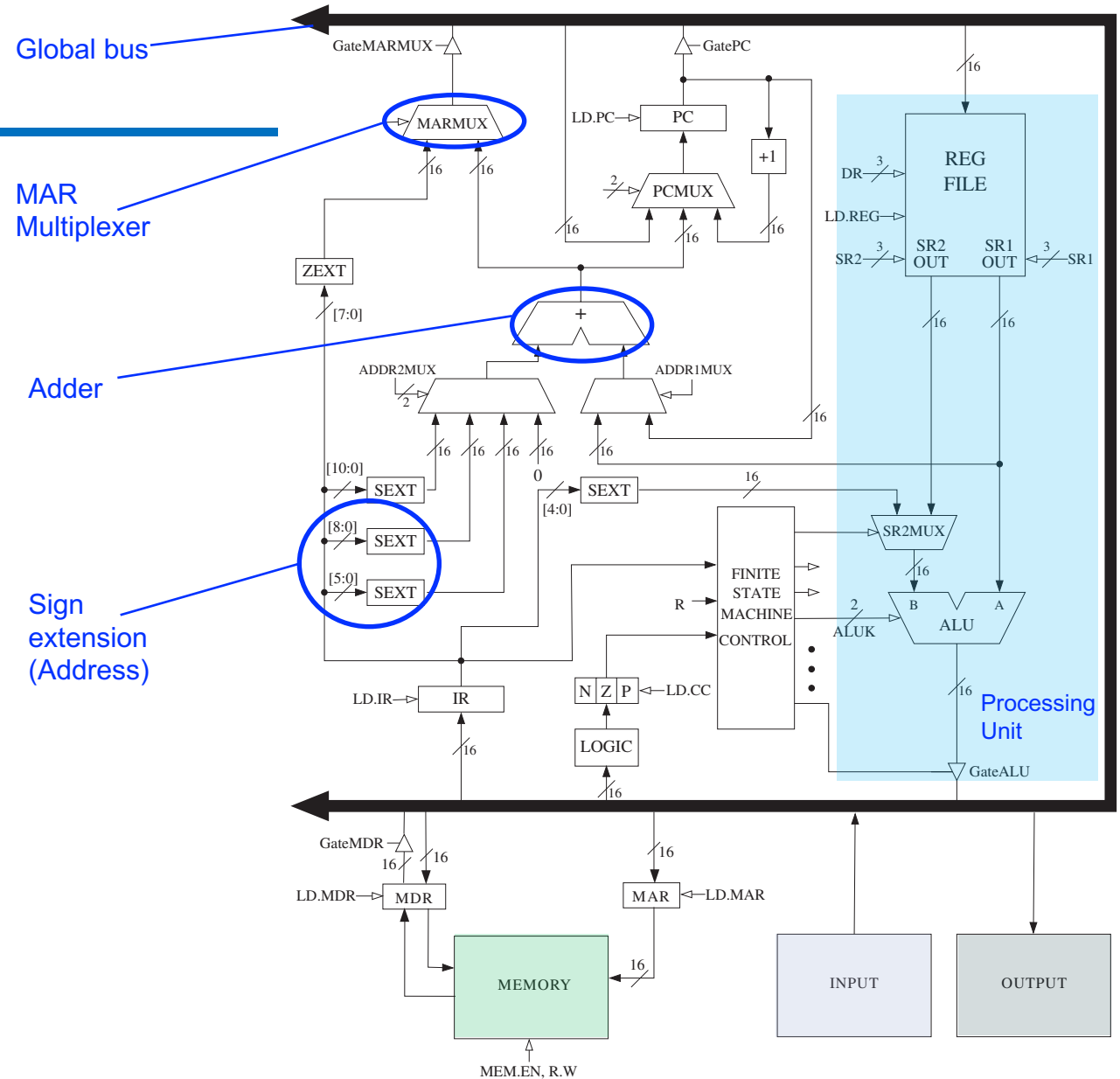
OP	DR	PCoffset9
E	5	0x1FD

Machine Code

OP	DR	PCoffset9
1 1 1 0	1 0 1	1 1 1 1 1 1 1 0 1
15	12	11 9 8 0



Address Calculation in LC-3 Data Path



Immediate Addressing Mode in MIPS

- ❑ In MIPS, `lui` (load upper immediate) loads a 16-bit immediate into the upper half of a register and sets the lower half to 0
- ❑ It is used to assign 32-bit constants to a register

High-level code

```
a = 0x6d5e4f3c;
```

MIPS assembly

```
# $s0 = a  
lui   $s0, 0x6d5e  
ori   $s0, 0x4f3c
```

Addressing Example in LC-3

□ What is the final value of R3?

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x30F6	1	1	1	0	0	0	1	1	1	1	1	1	1	1	0	1	R1 ← PC - 3
x30F7	0	0	0	1	0	1	0	0	0	1	1	0	1	1	1	0	R2 ← R1 + 14
x30F8	0	0	1	1	0	1	0	1	1	1	1	1	1	0	1	1	M[x30F4] ← R2
x30F9	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 ← 0
x30FA	0	0	0	1	0	1	0	0	1	0	1	0	0	1	0	1	R2 ← R2 + 5
x30FB	0	1	1	1	0	1	0	0	0	1	0	0	1	1	1	0	M[R1 + 14] ← R2
x30FC	1	0	1	0	0	1	1	1	1	1	1	1	0	1	1	1	R3 ← M[M[x30F4]]

LC-ISA Instruction Encodings

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD ⁺	0001				DR			SR1			0	00		SR2		
ADD ⁺	0001				DR			SR1			1	imm5				
AND ⁺	0101				DR			SR1			0	00		SR2		
AND ⁺	0101				DR			SR1			1	imm5				
BR	0000				n	z	p	PCoffset9								
JMP	1100				000			BaseR			000000					
JSR	0100				1	PCoffset11										
JSRR	0100				0	00		BaseR			000000					
LD ⁺	0010				DR			PCoffset9								
LDI ⁺	1010				DR			PCoffset9								
LDR ⁺	0110				DR			BaseR			offset6					
LEA ⁺	1110				DR			PCoffset9								
NOT ⁺	1001				DR			SR			111111					
RET	1100				000			111			000000					
RTI	1000				000000000000											
ST	0011				SR			PCoffset9								
STI	1011				SR			PCoffset9								
STR	0111				SR			BaseR			offset6					
TRAP	1111				0000			trapvect8								
reserved	1101															

Figure 5.3 Formats of the entire LC-3 instruction set. NOTE: + indicates instructions that modify condition codes

Addressing Example in LC-3

❑ What is the final value of R₃?

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x30F6	LEA	1	1	0	0	0	1	1	1	1	1	1	1	1	1	0	$R1 = PC - 3 = 0x30F7 - 3 = 0x30F4$
x30F7	ADD	0	0	1	0	1	0	0	0	1	1	0	1	1	1	0	$R2 = R1 + 14 = 0x30F4 + 14 = 0x3102$
x30F8	ST	0	1	1	0	1	0	1	1	1	1	1	1	0	1	0	$M[PC - 5] = M[0x30F4] = 0x3102$
x30F9	AND	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	$R2 = 0$
x30FA	ADD	0	0	1	0	1	0	0	1	0	1	0	1	0	1	0	$R2 = R2 + 5 = 5$
x30FB	STR	1	1	1	0	1	0	0	0	1	1	1	1	1	1	0	$M[R1 + 14] = M[0x30F4 + 14] = M[0x3102] = 5$
x30FC	LDI	0	1	0	0	1	1	1	1	1	1	1	1	1	1	0	$R3 = M[M[PC - 9]] = M[M[0x30FD - 9]] =$ $M[M[0x30F4]] = M[0x3102] = 5$

❑ The final value of R₃ is 5

Hardware Design

Control Flow Instructions

0	rs	rt	rd	shamt	funct
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

R-type

opcode	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

opcode	immediate
6-bit	26-bit

J-type

Control Flow Instructions

- ❑ Allow a program to execute **out of sequence**

- ❑ Conditional branches and unconditional jumps
 - **Conditional branches** are used to **make decisions**
 - E.g., if-else statement
 - In LC-3, three **condition codes** are used

 - **Jumps** are used to implement
 - **Loops**
 - **Function calls**
 - **JMP** in LC-3 and **j** in MIPS
 - We have already seen these

Hardware Design

Conditional Control Flow (Conditional Branching)

0	rs	rt	rd	shamt	funct
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

R-type

opcode	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

opcode	immediate
6-bit	26-bit

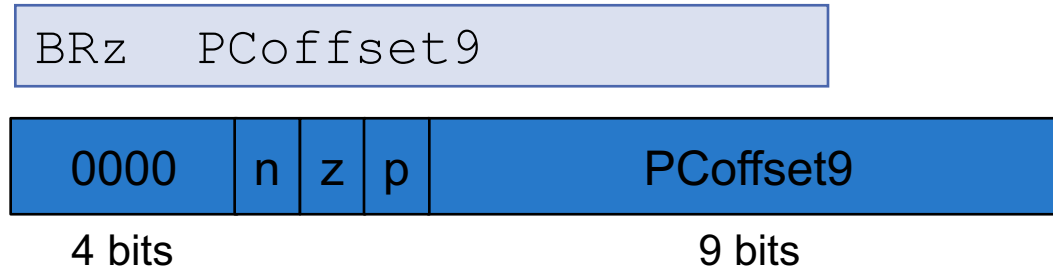
J-type

Condition Codes in LC-3

- ❑ Each time one GPR (R0-R7) is written, **three single-bit registers** are updated
- ❑ Each of these **condition codes** is either set (set to 1) or cleared (set to 0)
 - If the written value is **negative**
 - **N** is set, Z and P are cleared
 - If the written value is **zero**
 - **Z** is set, N and P are cleared
 - If the written value is **positive**
 - **P** is set, N and Z are cleared
- ❑ x86 and SPARC are examples of ISAs that use condition codes

Conditional Branches in LC-3

❑ BRz (Branch if Zero)



- n, z, p = which condition code is tested (N, Z, and/or P)
 - n, z, p : instruction bits to identify the condition codes to be tested
 - N, Z, P : values of the corresponding condition codes
- $PCoffset9$ = immediate or constant value
- if $((n \text{ AND } N) \text{ OR } (p \text{ AND } P) \text{ OR } (z \text{ AND } Z))$
 - then $PC \leftarrow PC^{\dagger} + \text{sign-extend}(PCoffset9)$
- Variations: BRn, BRz, BRp, BRzp, BRnp, BRnz, BRnzp

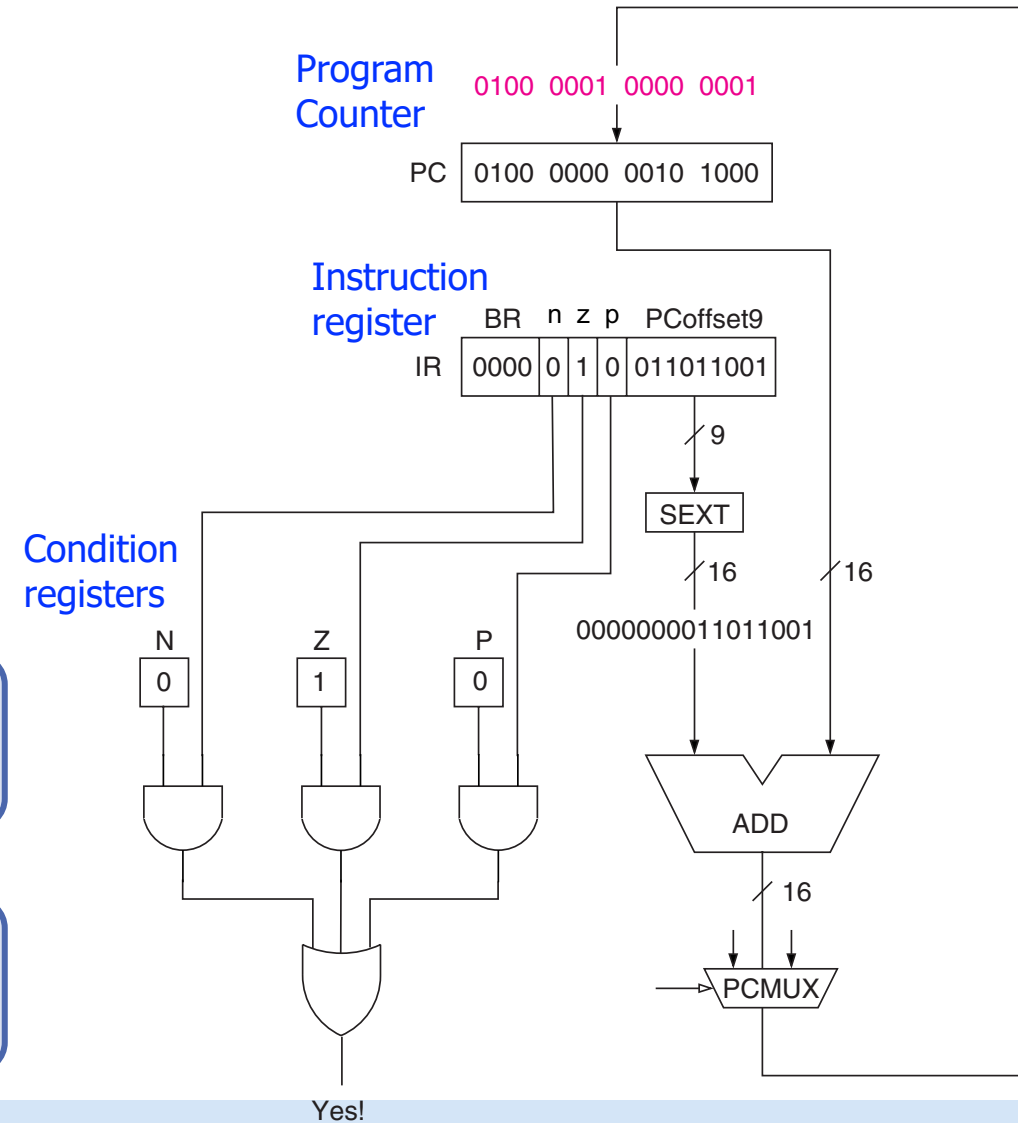
Conditional Branches in LC-3

BRz

BRz 0x0D9

What if $n = z = p = 1$?* (i.e., BRnzp)

And what if $n = z = p = 0$?

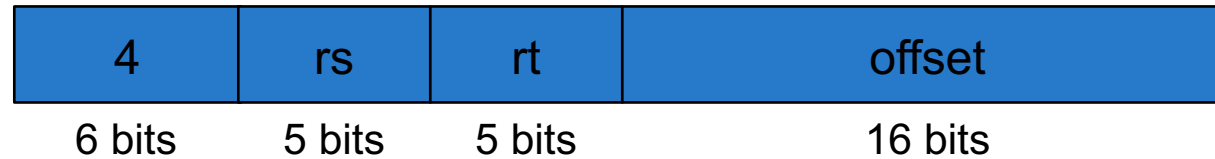


*n, z, p are the instruction bits to identify the condition codes to be tested

Conditional Branches in MIPS

❑ beq (Branch if Equal)

```
beq  $s0, $s1, offset
```



- 4 = opcode
- rs, rt = source registers
- offset = immediate or constant value
- if $rs == rt$
 - then $PC \leftarrow PC^+ + \text{sign-extend}(\text{offset}) * 4$
- Variations: beq, bne, blez, bgtz

Branch If Equal in MIPS and LC-3

MIPS assembly

```
beq $s0, $s1, offset
```

LC-3 assembly

```
NOT   R2, R1  
ADD   R3, R2, #1  
ADD   R4, R3, R0  
BRz   offset
```

**Subtract
(R0 - R1)**

- ❑ This is an example of a **tradeoff** in the instruction set
 - The same functionality requires **more instructions in LC-3**
 - But the **control logic** requires **more complexity in MIPS**

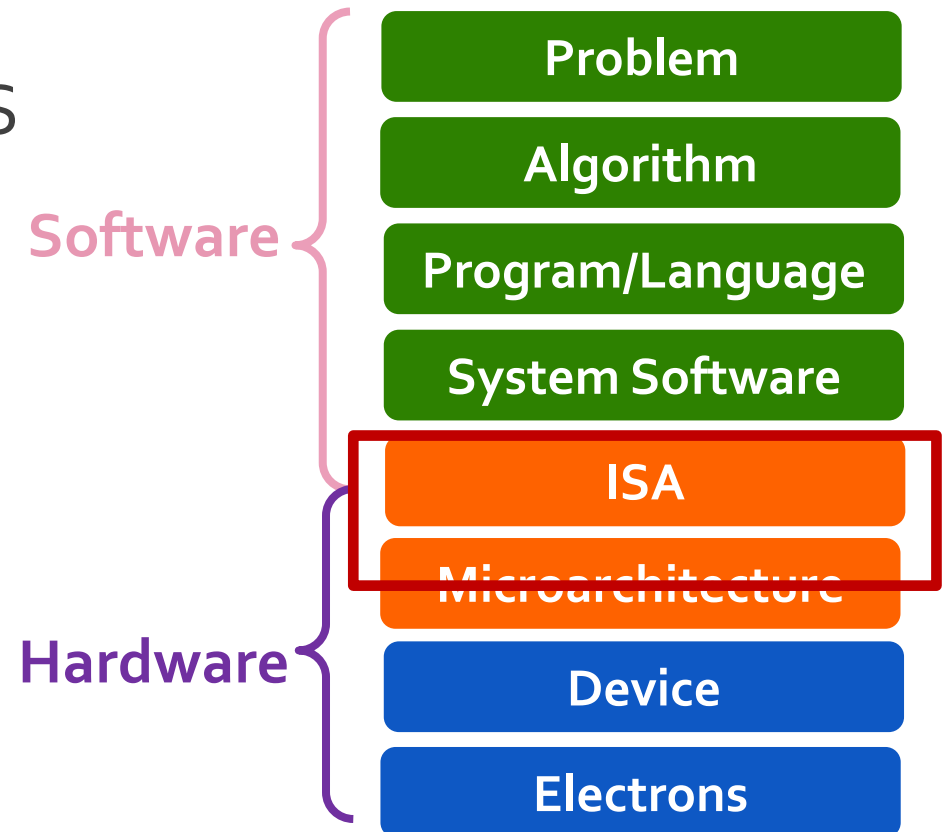
What We Learned

- ❑ Basic elements of a computer & the von Neumann model
 - LC-3: An example von Neumann machine

- ❑ Instruction Set Architectures: LC-3 and MIPS
 - Operate instructions
 - Data movement instructions
 - Control instructions

- ❑ Instruction formats

- ❑ Addressing modes



Many Different ISAs Over Decades

- ❑ x86
- ❑ PDP-x: Programmed Data Processor (PDP-11)
- ❑ VAX
- ❑ IBM 360
- ❑ CDC 6600
- ❑ SIMD ISAs: CRAY-1, Connection Machine
- ❑ VLIW ISAs: Multiflow, Cydrome, IA-64 (EPIC)
- ❑ PowerPC, POWER
- ❑ RISC ISAs: Alpha, MIPS, SPARC, ARM, RISC-V, ...
- ❑ What are the fundamental differences?
 - E.g., how instructions are specified and what they do
 - E.g., how complex are instructions, data types, addr. modes

Complex vs. Simple Instructions+Data Types

- ❑ **Complex instruction:** An instruction **does a lot of work**, e.g. many operations
 - Insert into a doubly linked list
 - Compute FFT
 - String copy
 - Matrix multiply
 - ...

- ❑ **Simple instruction:** An instruction **does little work** -- it is a primitive using which complex operations can be built
 - Add
 - XOR
 - Multiply
 - ...

Complex vs. Simple Instructions+Data Types

□ Advantages of Complex Instructions + Data Types

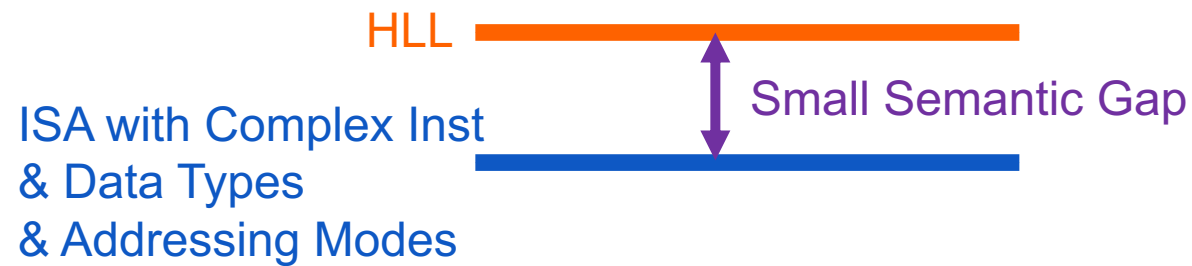
- + **Denser encoding** → smaller code size → better memory utilization, saves off-chip bandwidth, better cache hit rate (better packing of instructions)
- + **Simpler compiler**: no need to optimize small instructions as much

□ Disadvantages of Complex Instructions + Data Types

- **Larger chunks of work** → compiler has less opportunity to optimize (limited in fine-grained optimizations it can do)
- **More complex hardware** → translation from a high level to control signals and optimization needs to be done by hardware

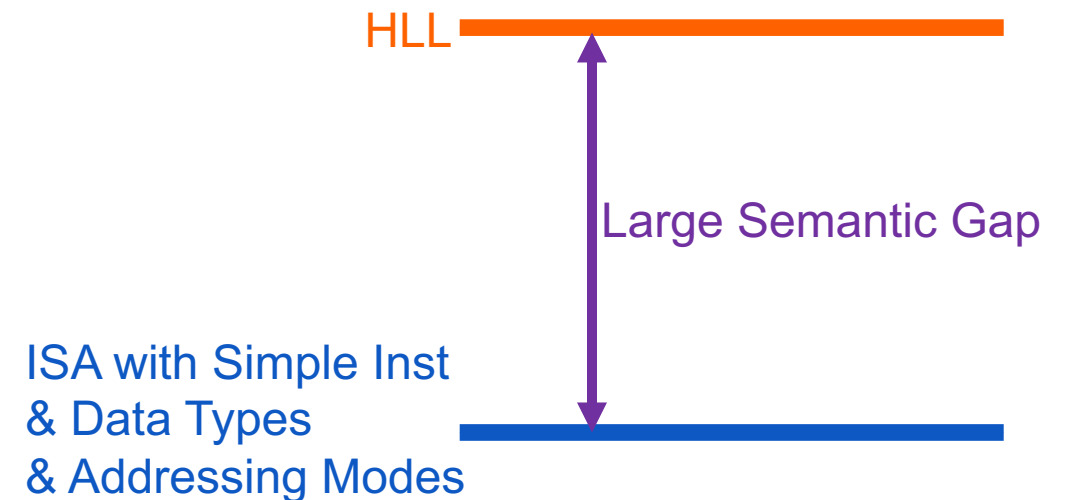
Semantic Gap

- How close are instructions & data types & addressing modes to high-level language (HLL)



HW Control Signals

Easier mapping of HLL to ISA
Less work for software designer
More work for hardware designer
Optimization burden on HW



HW Control Signals

Harder mapping of HLL to ISA
More work for software designer
Less work for hardware designer
Optimization burden on SW

Hardware Design

Lecture 2: Instruction Set Architectures II

Dr. Haiyu Mao

29.01.2026