

Design of Digital Circuits

Lecture 10a: Instruction Set Architecture

Prof. Onur Mutlu

ETH Zurich

Spring 2019

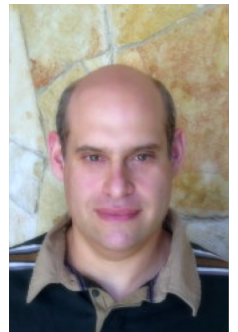
22 March 2019

Talk Announcement (Today)

- 22 March 2019, Friday, 16:00-17:00, CAB G51
- **Cross-Layer Architecture for Deep Learning**
- Prof. Mattan Erez, University of Texas at Austin

- High-performance DNN inference and training is essential for the ongoing ML revolution. Training of DNNs requires massive memory capacity and bandwidth, and is generally a huge pain, especially for researchers. While significant research effort has been dedicated to inference accelerator, less work has been done on training, especially work that crosses the algorithmic and implementation layers. The result is a very limited number of high-cost accelerators available, in particular, with very expensive high-bandwidth memories. I will motivate and discuss some of our recent work on accelerating training (of CNNs) that combines understanding of and changes to the algorithm with matching hardware architecture modifications.

Optional Review



Extra Assignment 2: Moore's Law (I)

- **Paper review**
- G.E. Moore. "Cramming more components onto integrated circuits," Electronics magazine, 1965

- **Optional Assignment – for 1% extra credit**
 - **Write a 1-page review**
 - Upload PDF file to Moodle – Deadline: Friday, March 22

- I strongly recommend that you **follow my guidelines for (paper) review** (see next slide)

Extra Assignment 2: Moore's Law (II)

■ Guidelines on how to review papers critically

- **Guideline slides:** [pdf](#) [ppt](#)
- **Video:** <https://www.youtube.com/watch?v=tOL6FANAJ8c>
- Example reviews on "Main Memory Scaling: Challenges and Solution Directions" ([link to the paper](#))
 - [Review 1](#)
 - [Review 2](#)
- Example review on "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems" ([link to the paper](#))
 - [Review 1](#)

Agenda for Today & Next Few Lectures

- LC-3 and MIPS Instruction Set Architectures
- LC-3 and MIPS assembly and programming
- Introduction to microarchitecture and single-cycle microarchitecture
- Multi-cycle microarchitecture

Required Readings

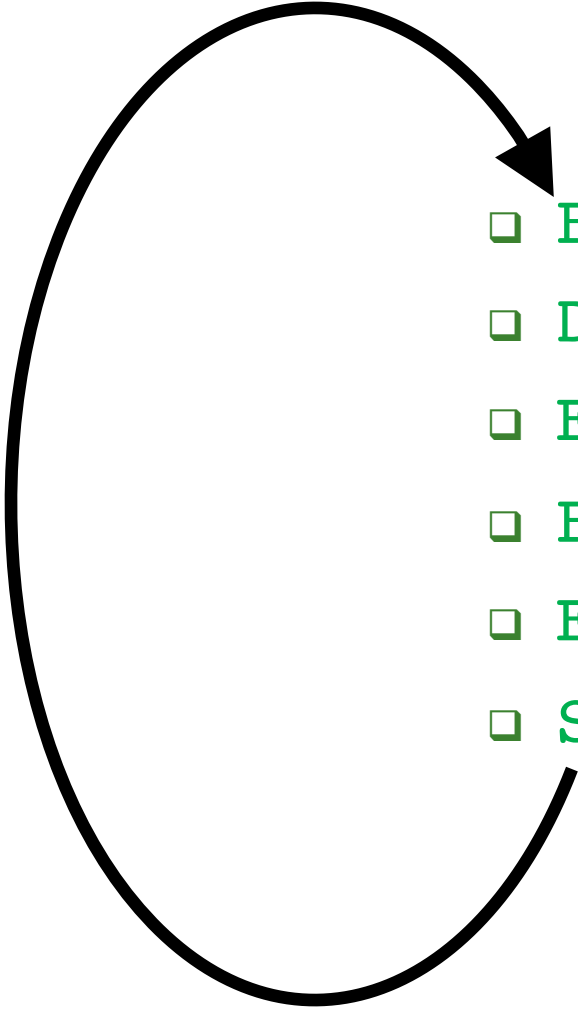
■ This week

- Von Neumann Model, LC-3, and MIPS
 - P&P, Chapters 4, 5
 - H&H, Chapter 6
 - P&P, Appendices A and C (ISA and microarchitecture of LC-3)
 - H&H, Appendix B (MIPS instructions)
- Programming
 - P&P, Chapter 6
- **Recommended:** H&H Chapter 5, especially 5.1, 5.2, 5.4, 5.5

■ Next week

- Introduction to microarchitecture and single-cycle microarchitecture
 - H&H, Chapter 7.1-7.3
 - P&P, Appendices A and C
- Multi-cycle microarchitecture
 - H&H, Chapter 7.4
 - P&P, Appendices A and C

Recall: The Instruction Cycle

- 
- ❑ FETCH
 - ❑ DECODE
 - ❑ EVALUATE ADDRESS
 - ❑ FETCH OPERANDS
 - ❑ EXECUTE
 - ❑ STORE RESULT

Instruction Set Architectures

Recall: 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: 32 bits)
 - Word- or Byte-addressable
 - The **register set**
 - R0 to R7 in LC-3
 - 32 registers in MIPS
 - The **instruction set**
 - Opcodes
 - Data types
 - Addressing modes

Problem
Algorithm
Program
ISA
Microarchitecture
Circuits
Electrons

Recall: 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

Recall: 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															

Recall: Funct in MIPS R-Type Instructions (I)

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

Opcode is 0
in MIPS R-
Type
instructions.
Funct defines
the operation

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)

Recall: 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	$[rs] < [rt] ? [rd] = 1 : [rd] = 0$

- Find the complete list of instructions in the appendix

Data Types

- An ISA supports one or several data types
- LC-3 only supports 2's complement integers
- MIPS supports
 - 2's complement integers
 - Unsigned integers
 - Floating point
- Again, **tradeoffs** are involved

Data Type Tradeoffs

- What is the benefit of **having more or high-level data types** in the ISA?
- What is the disadvantage?

- Think compiler/programmer vs. microarchitect

- Concept of **semantic gap**
 - Data types coupled tightly to the semantic level, or complexity of instructions

- Example: Early RISC architectures vs. Intel 432
 - Early RISC (e.g., MIPS): Only integer data type
 - Intel 432: Object data type, capability based machine

Addressing Modes

- An addressing mode is a mechanism for specifying where an operand is located
- There 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 of them are memory addressing modes
 - PC-relative
 - Indirect
 - Base+offset
- In addition, MIPS has pseudo-direct addressing (for j and jal), but does not have indirect addressing

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
 - Most of **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

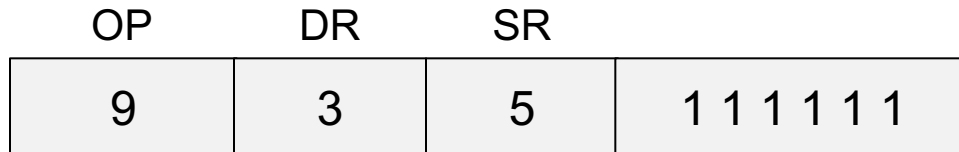
NOT in LC-3

NOT assembly and machine code

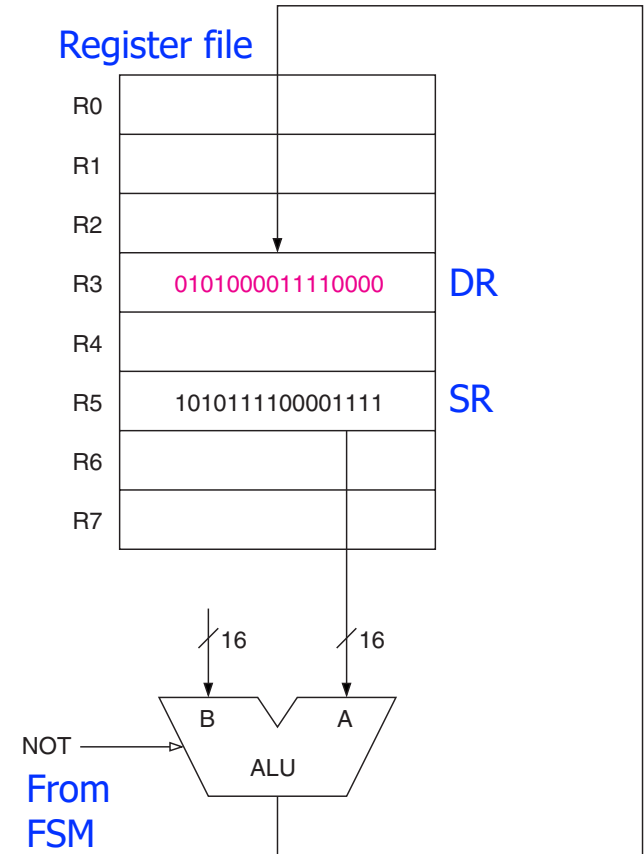
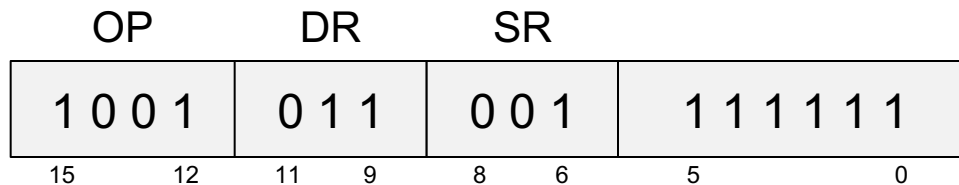
LC-3 assembly

```
NOT R3, R5
```

Field Values



Machine Code



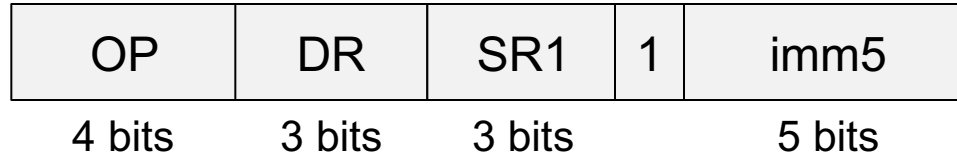
There is **no NOT in MIPS**. How is it implemented?

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
- **Subtraction** is another necessary operation
 - How is it implemented in LC-3 and MIPS?

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** ← **SR1** + **sign-extend(imm5)**
 - E.g., **AND** = **0101** (same OP as the register-mode AND)
 - **DR** ← **SR1** **AND** **sign-extend(imm5)**
- SR1 = source register
- DR = destination register
- **imm5** = Literal or immediate (sign-extend to 16 bits)

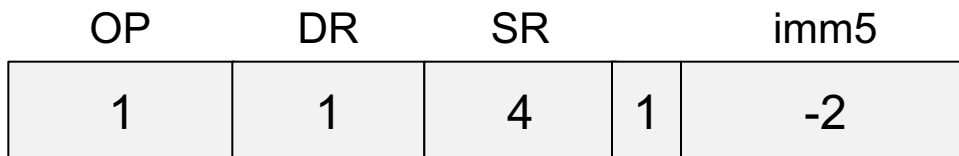
ADD with one Literal in LC-3

- ADD assembly and machine code

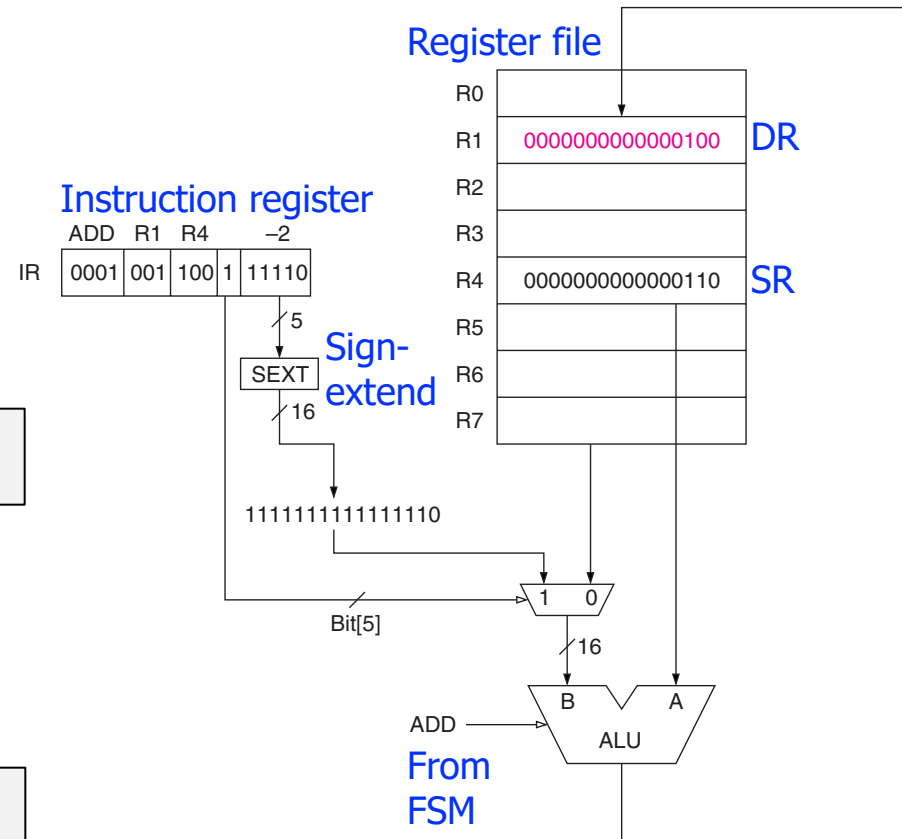
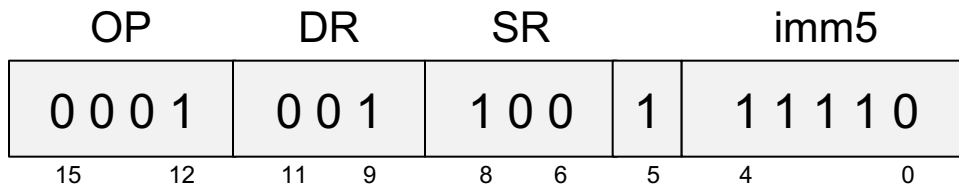
LC-3 assembly

ADD R1, R4, #-2

Field Values

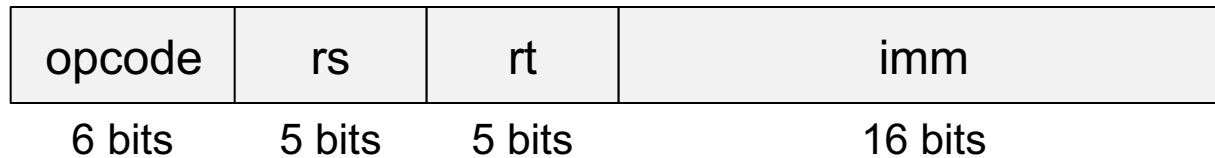


Machine Code



Instructions with one Literal in MIPS

- I-type
 - 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
0	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

Subtract in 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 R4

■ Tradeoff in LC-3

- More instructions
- But, simpler control logic

Subtract Immediate

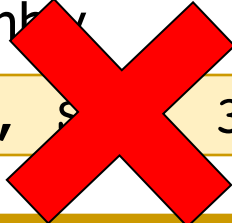
- MIPS assembly

High-level code

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

MIPS assembly

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



Is **subi** necessary in MIPS?

MIPS assembly

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

- LC-3

High-level code

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

LC-3 assembly

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

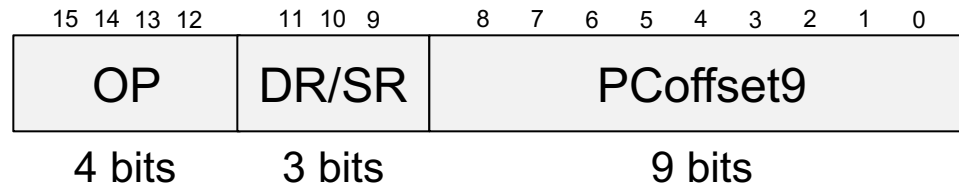
Data Movement Instructions and Addressing Modes

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: $DR \leftarrow \text{Memory}[PC^\dagger + \text{sign-extend}(\text{PCoffset9})]$
- ST: $\text{Memory}[PC^\dagger + \text{sign-extend}(\text{PCoffset9})] \leftarrow SR$

[†] This is the incremented PC

LD in LC-3

LD assembly and machine code

LC-3 assembly

LD R2, 0x1AF

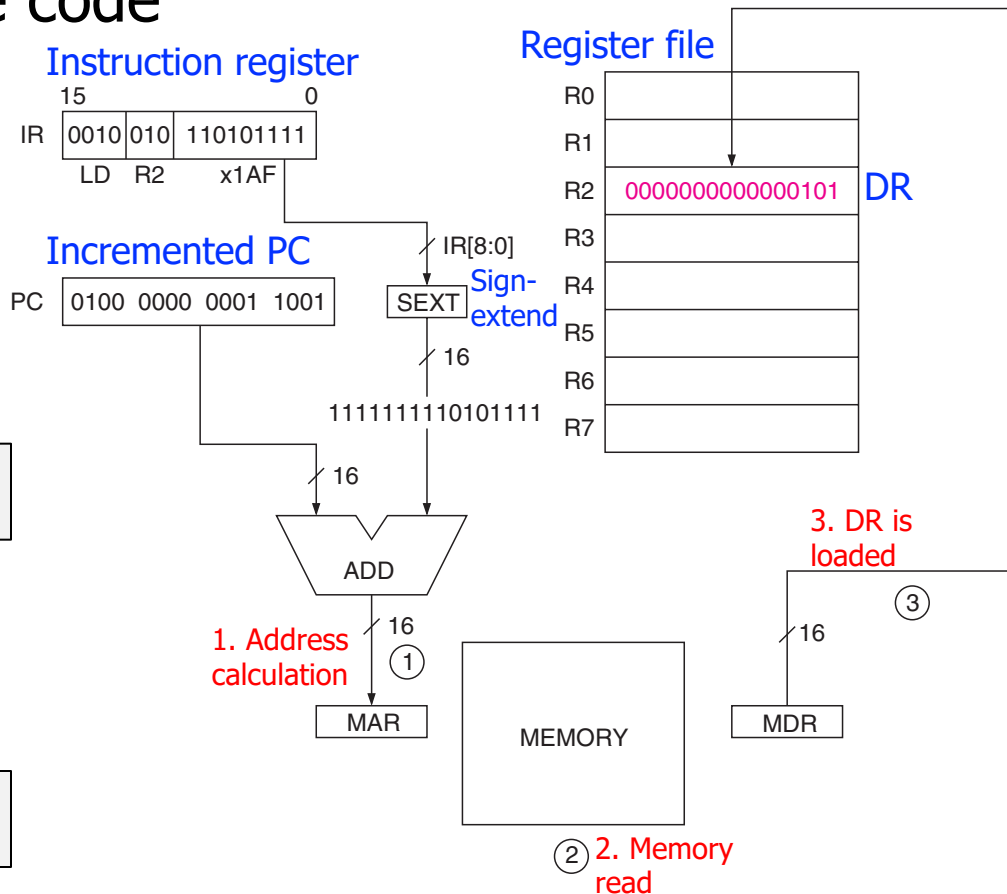
Field Values

OP	DR	PCoffset9
2	2	0x1AF

Machine Code

OP	DR	PCoffset9
0010	010	110101111

15 12 11 9 8 0

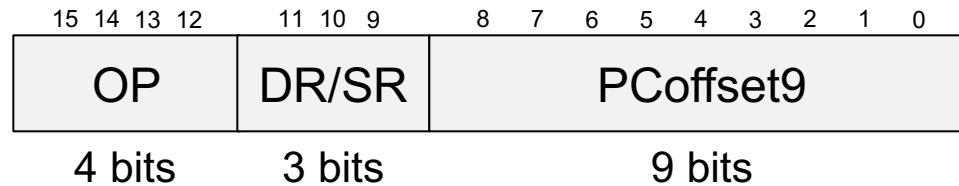


The memory address is **only +256 to -255** locations away of the **LD or ST instruction**

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: $DR \leftarrow \text{Memory}[\text{Memory}[\text{PC}^\dagger + \text{sign-extend}(\text{PCoffset9})]]$
- STI: $\text{Memory}[\text{Memory}[\text{PC}^\dagger + \text{sign-extend}(\text{PCoffset9})]] \leftarrow SR$

[†] This is the incremented PC

LDI in LC-3

LDI assembly and machine code

LC-3 assembly

LDI R3, 0x1CC

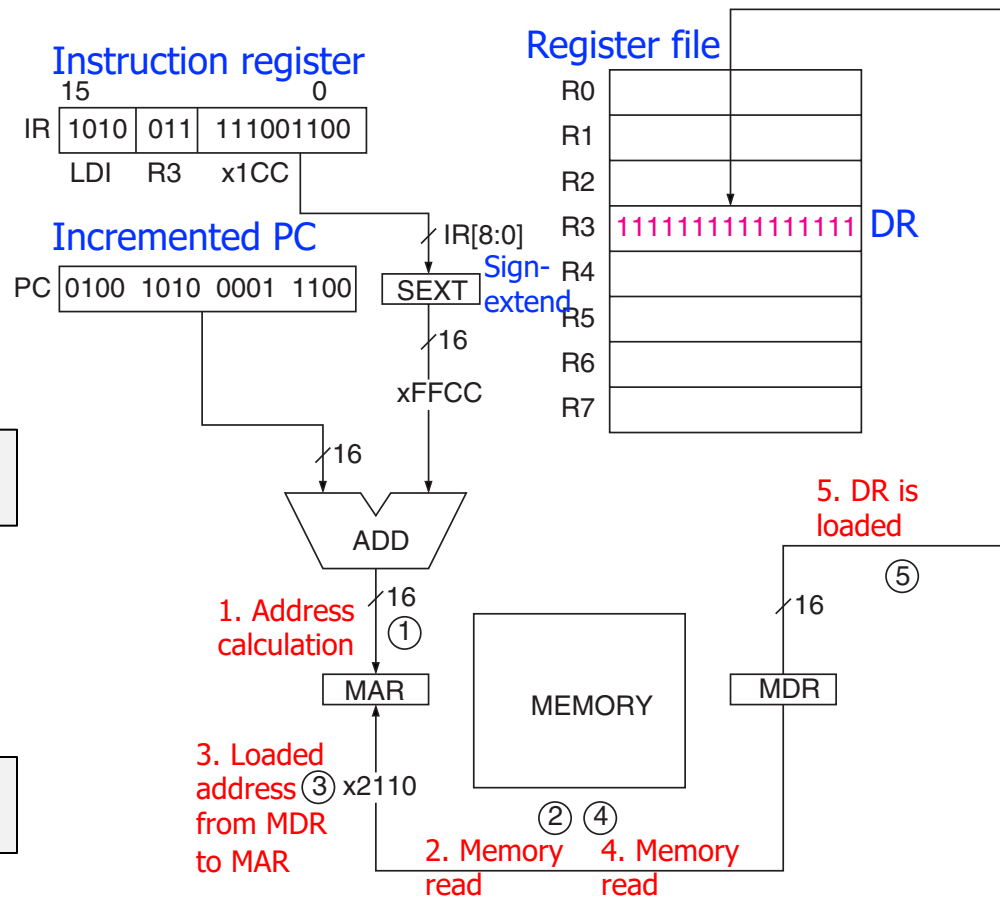
Field Values

OP	DR	PCoffset9
A	3	0x1CC

Machine Code

OP	DR	PCoffset9
1010	011	111001100

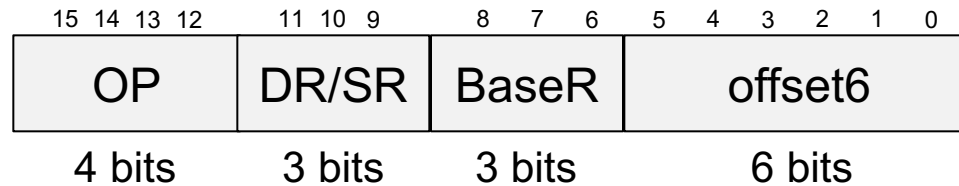
15 12 11 9 8 0



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

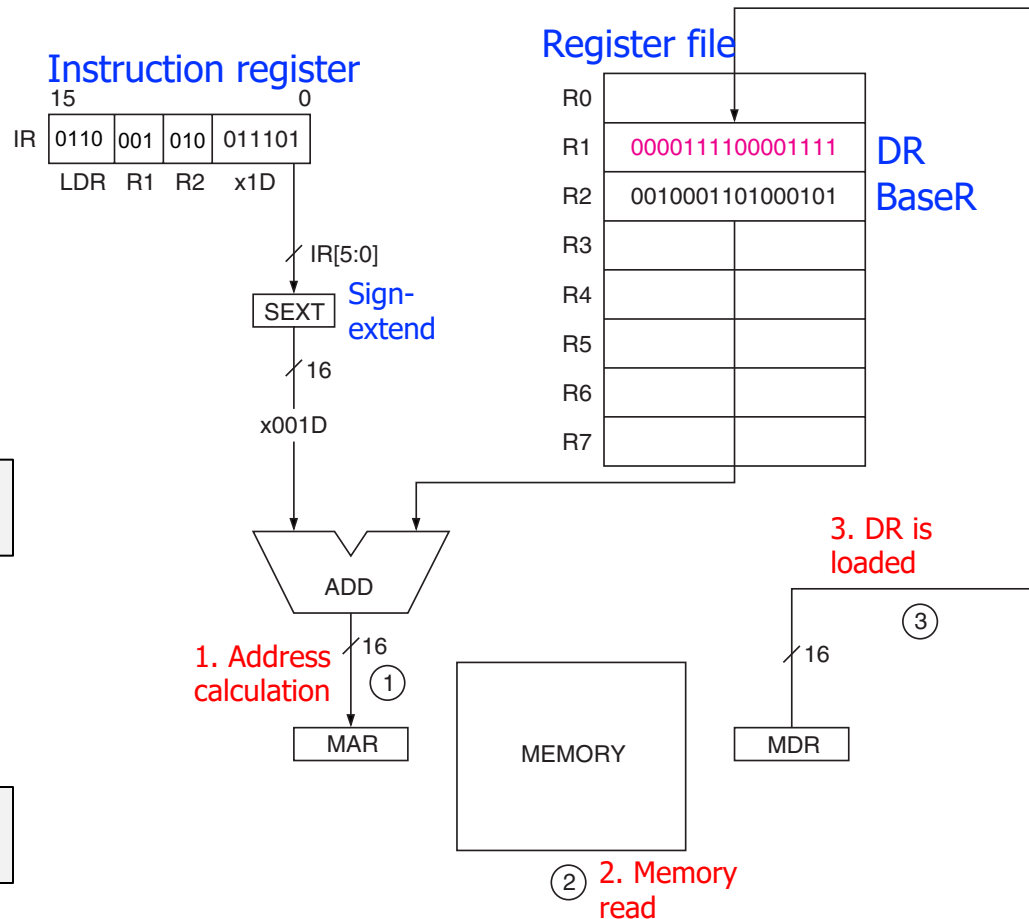
LDR R1, R2, 0x1D

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

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

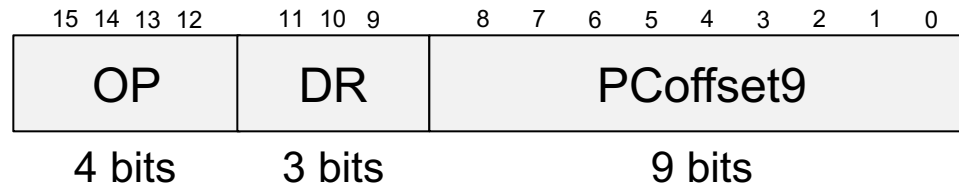
```
lw    $t0, 0($s0)  
add   $t1, $t0, $s2  
addi  $t2, $t1, -5  
sw    $t2, 0($s1)
```

LC-3 assembly

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

Immediate Addressing Mode

■ 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 **access memory**, but **LEA does not** → Hence the name *Load Effective Address*

[†] This is the incremented PC

LEA in LC-3

LEA assembly and machine code

LC-3 assembly

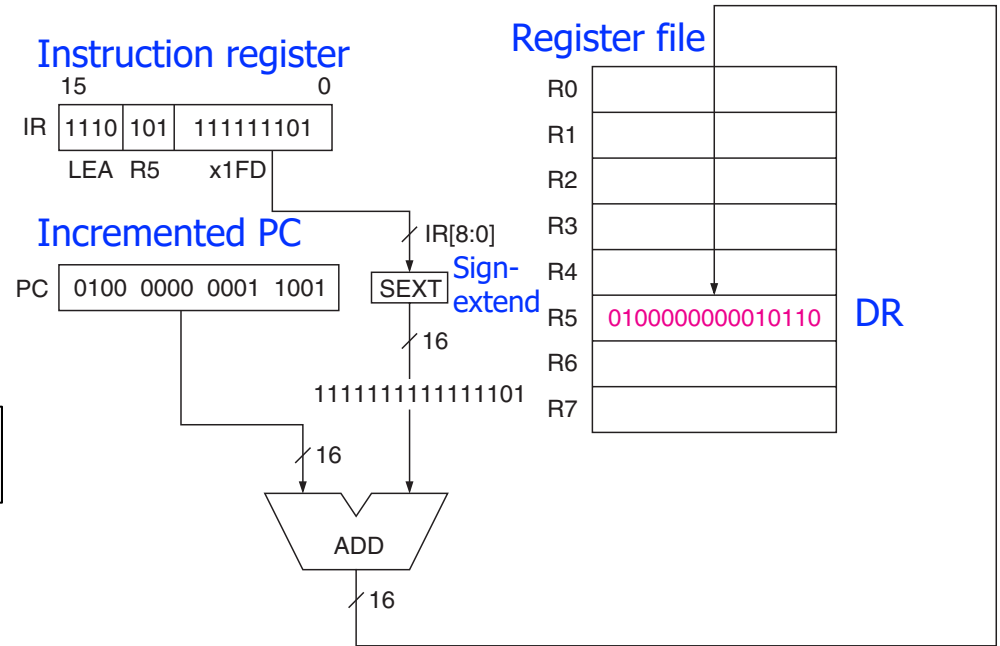
```
LEA R5, #-3
```

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



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?

P&P, Chapter 5.3.5

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]]

Addressing Example in LC-3

- What is the final value of R3?

P&P, Chapter 5.3.5

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x30F6	1	1	0	0	0	1	3	1	1	1	1	1	1	1	1	0	R1 = PC - 3 = 0x30F7 - 3 = 0x30F4
x30F7	0	0	1	0	1	0	0	0	1	1	0	1	1	0	1	0	R2 = R1 + 14 = 0x30F4 + 14 = 0x3102
x30F8	0	1	1	0	1	0	5	1	1	1	1	1	0	1	0	1	M[PC - 5] = M[0x030F4] = 0x3102
x30F9	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	0	R2 = 0
x30FA	0	0	1	0	1	0	0	1	0	1	5	0	1	0	1	0	R2 = R2 + 5 = 5
x30FB	1	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0	M[R1 + 14] = M[0x30F4 + 14] = M[0x3102] = 5
x30FC	0	1	0	0	1	1	9	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**

Control Flow Instructions

Control Flow Instructions

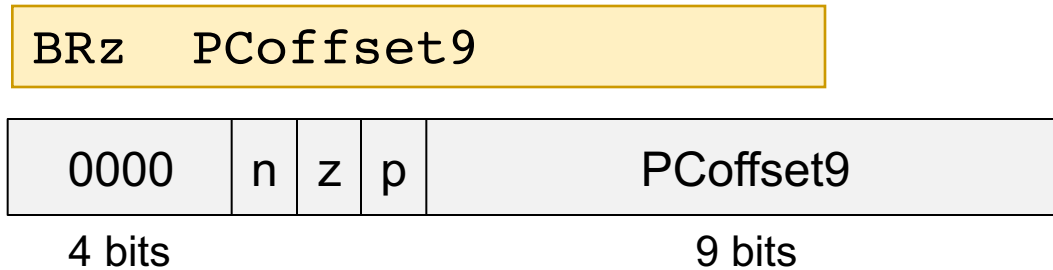
- Allow a program to execute **out of sequence**
- Conditional branches and jumps
 - **Conditional branches** are used to **make decisions**
 - E.g., if-else statement
 - In LC-3, three **condition codes** are used
 - **Jumps** are used to implement
 - **Loops**
 - **Function calls**
 - **JMP** in LC-3 and **j** in MIPS

Condition Codes in LC-3

- Each time one GPR (R0-R7) is written, **three single-bit registers** are updated
- Each of these **condition codes** are 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 P are cleared
- SPARC and x86 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 AND N) OR (p AND P) OR (z AND Z))**
 - **then $PC \leftarrow PC^\dagger + \text{sign-extend}(PCoffset9)$**
- Variations: BRn, BRz, BRp, BRzp, BRnp, BRnz, BRnzp

[†] This is the incremented PC

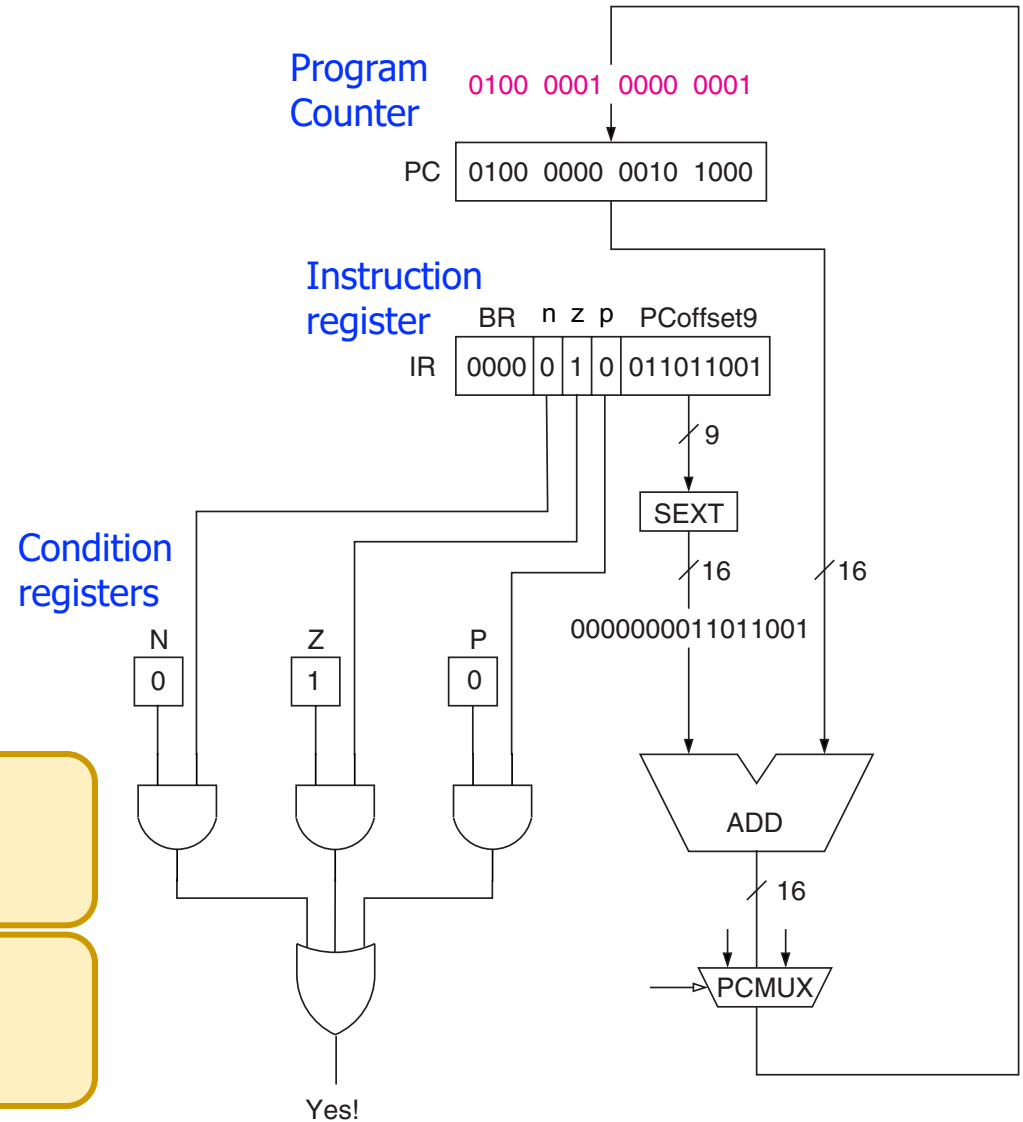
Conditional Branches in LC-3

BRz

BRz 0x0D9

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

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

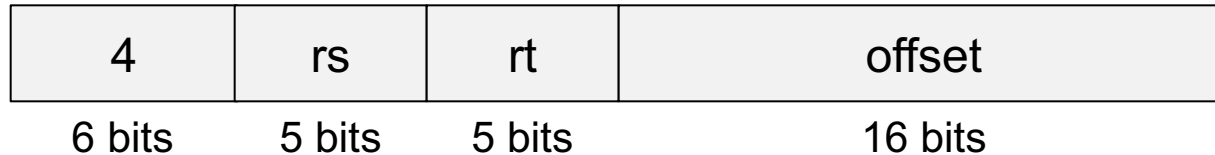


*n, z, p are the instruction bits to identify the condition codes to be tested

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 $PC \leftarrow PC^{\dagger} + \text{sign-extend}(\text{offset}) * 4$
- Variations: beq, bne, blez, bgtz

[†] This is the incremented PC

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 **tradeoff** in the instruction set
 - The same functionality requires **more instructions in LC-3**
 - But, the **control logic** requires **more complexity in MIPS**

Lecture Summary

- The von Neumann model
 - LC-3: An example of von Neumann machine
- Instruction Set Architectures: LC-3 and MIPS
 - Operate instructions
 - Data movement instructions
 - Control instructions
- Instruction formats
- Addressing modes

Design of Digital Circuits

Lecture 10a: Instruction Set Architecture

Prof. Onur Mutlu

ETH Zurich

Spring 2019

22 March 2019