

Design of Digital Circuits

Lecture 10: ISA (II)

and Assembly Programming

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zurich

Spring 2018

23 March 2018

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
- Microprogramming

Readings

■ This week

- Von Neumann Model, LC-3, and MIPS
 - P&P, Chapter 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
- Digital Building Blocks
 - H&H, Chapter 5

■ Next week

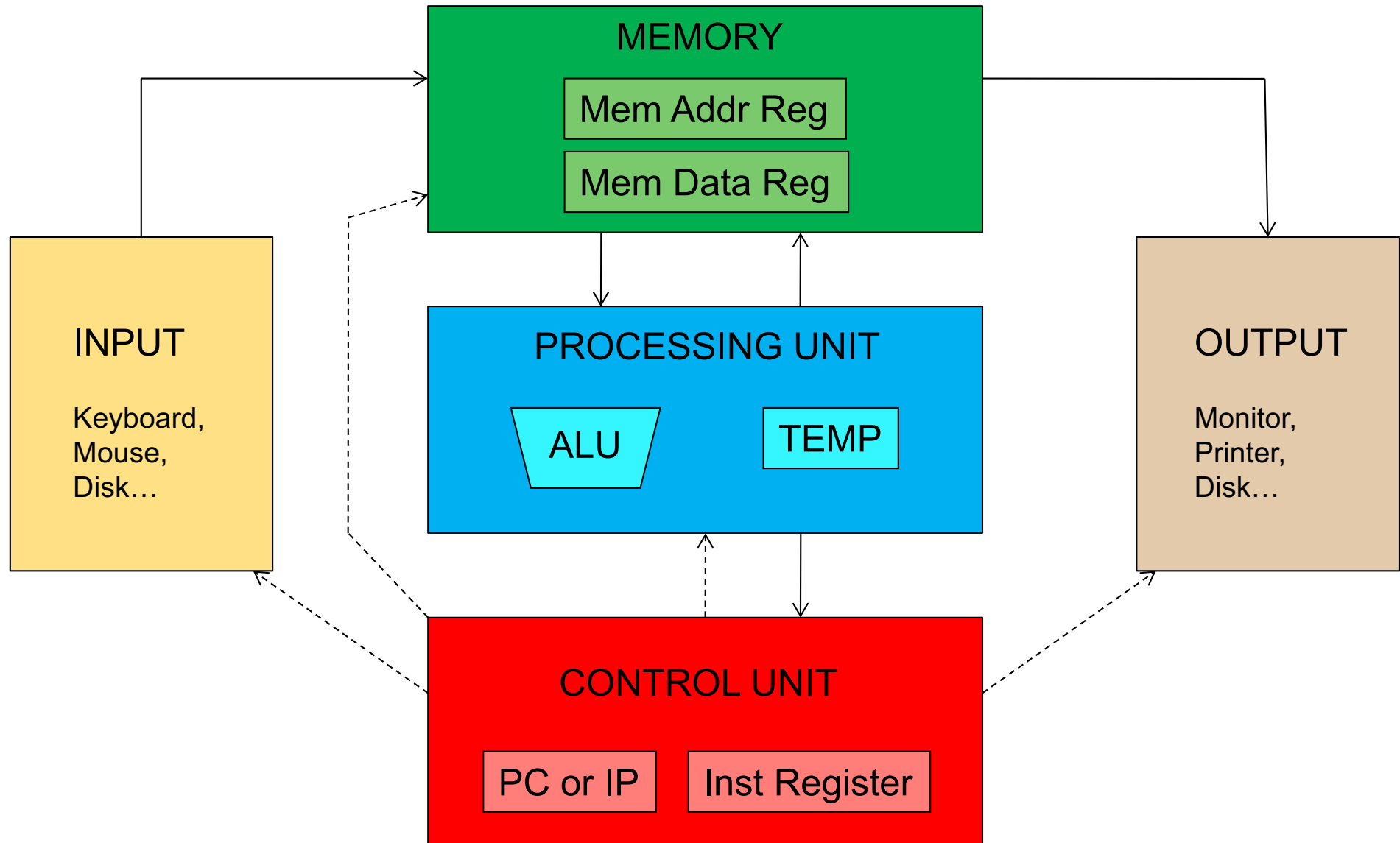
- Introduction to microarchitecture and single-cycle microarchitecture
 - P&P, Appendices A and C
 - H&H, Chapter 7.1-7.3
- Multi-cycle microarchitecture
 - P&P, Appendices A and C
 - H&H, Chapter 7.4
- Microprogramming
 - P&P, Appendices A and C

What Will We Learn Today?

- Instruction Set Architectures: LC-3 and MIPS
 - Operate instructions
 - Data movement instructions
 - Control instructions
- Instruction formats
- Addressing modes

- Assembly Programming
 - Programming constructs
 - Debugging
 - Conditional statements and loops in MIPS assembly
 - Arrays in MIPS assembly
 - Function calls
 - The stack

Recall: The Von Neumann Model



Recall: LC-3: A Von Neumann Machine

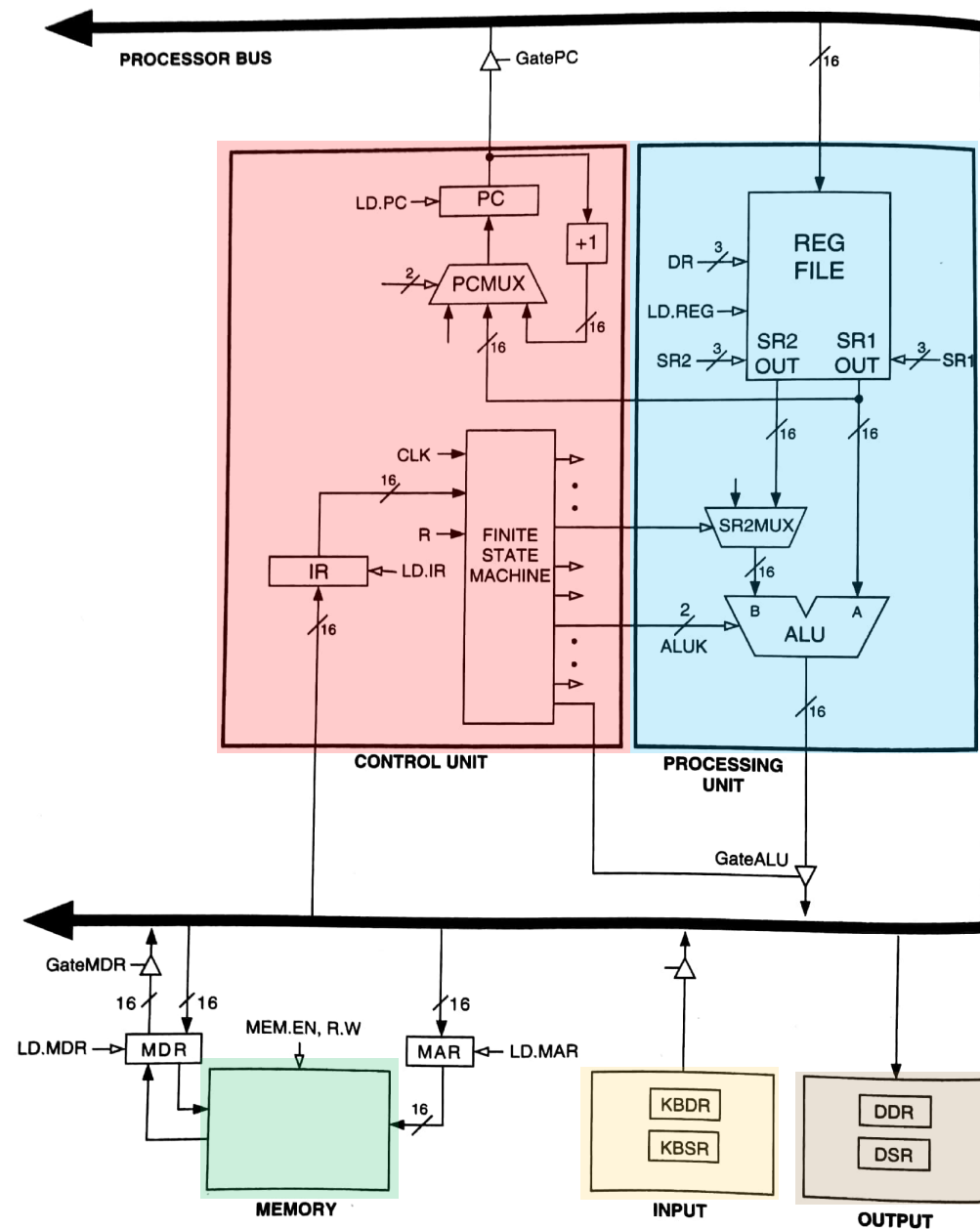
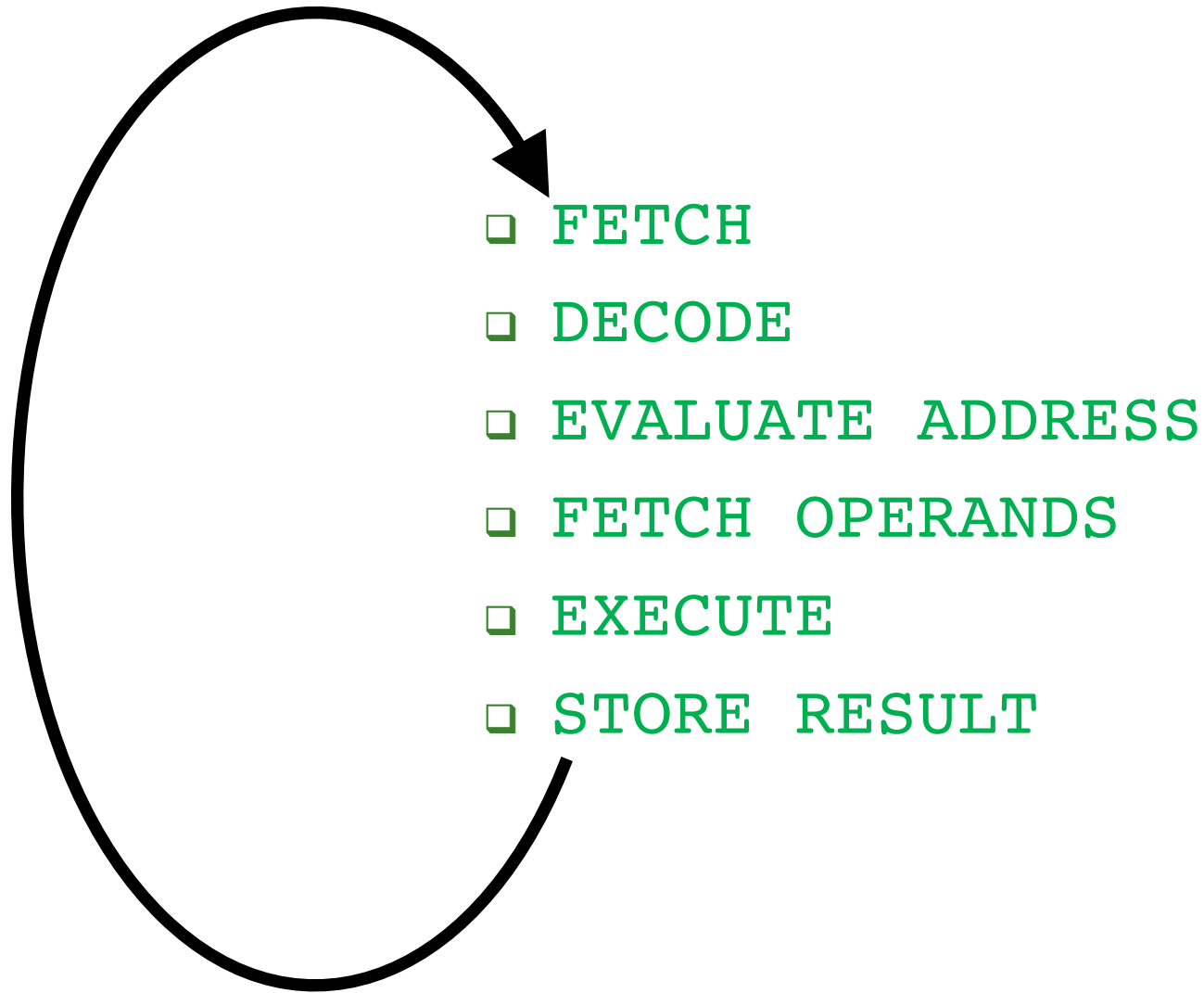


Figure 4.3 The LC-3 as an example of the von Neumann model

Recall: The Instruction Cycle



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

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 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

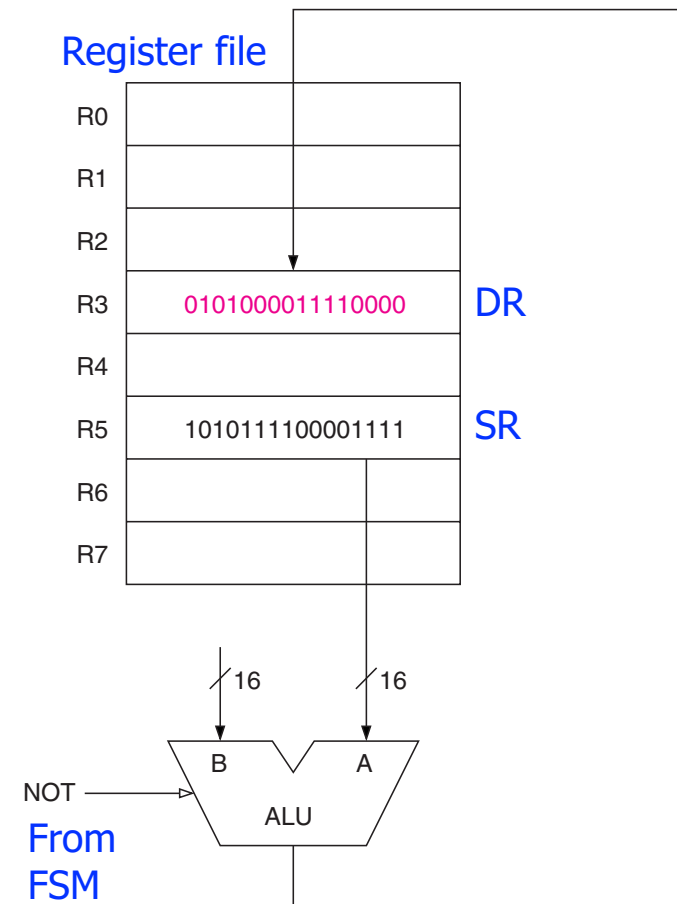
NOT R3, R5

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



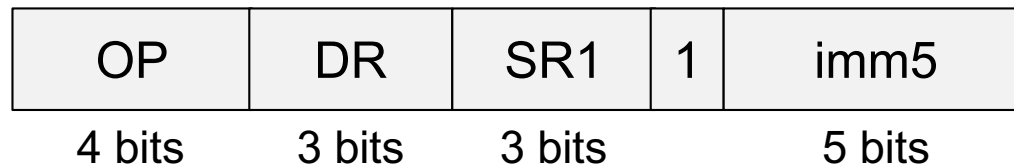
There is **no NOT in MIPS**. How is it implemented?

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** \leftarrow **SR1** + sign-extend(imm5)
 - E.g., **AND** = **0101** (same OP as the register-mode AND)
 - **DR** \leftarrow **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

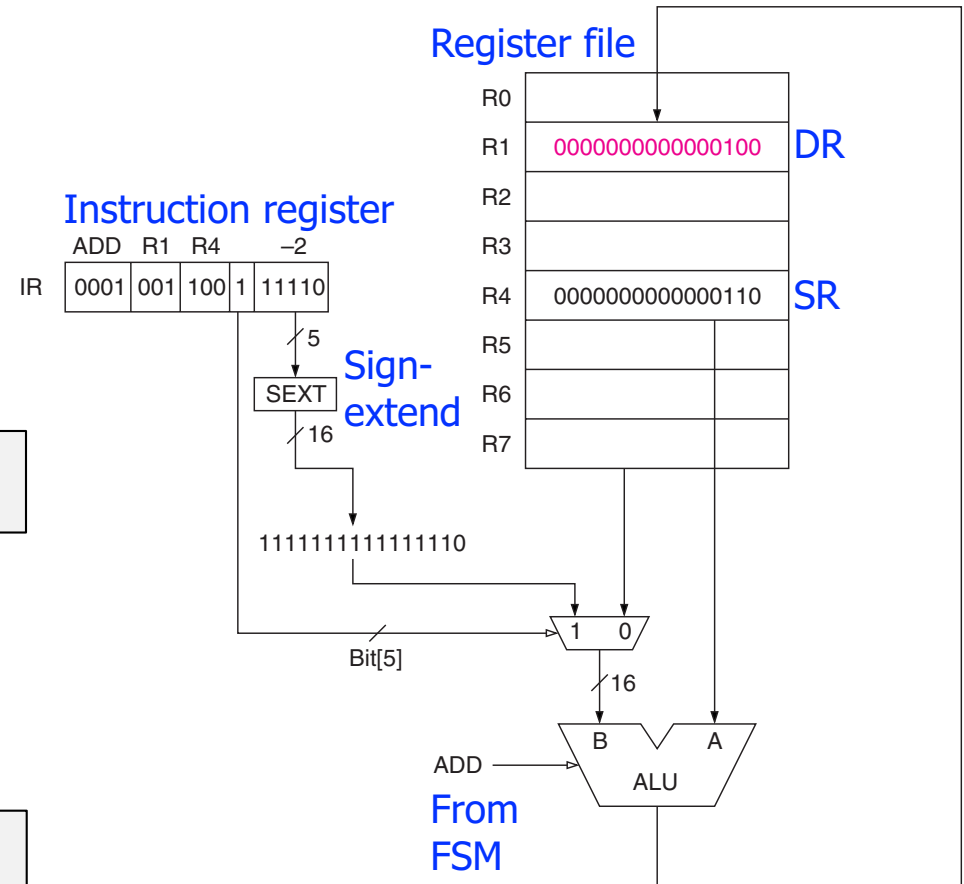
```
ADD R1, R4, #-2
```

Field Values

OP	DR	SR		imm5
1	1	4	1	-2

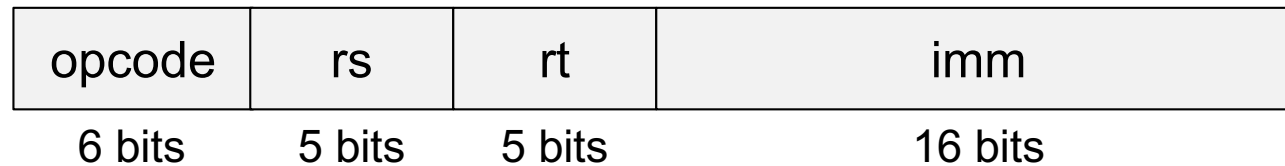
Machine Code

OP		DR		SR		imm5		
0001		001		100		1	11110	
15	12	11	9	8	6	5	4	0



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

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

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

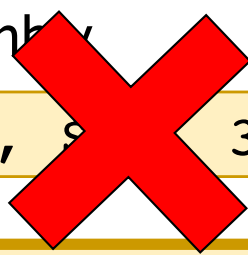
- MIPS assembly

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
```

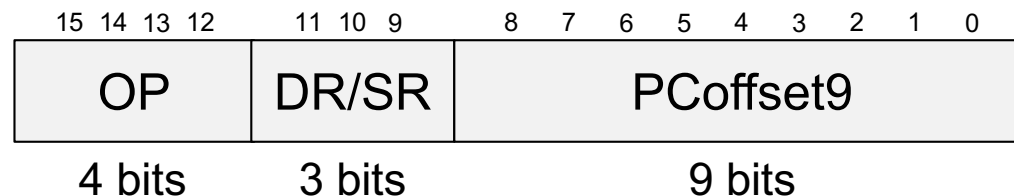
Data Movement Instructions and Addressing Modes

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
```

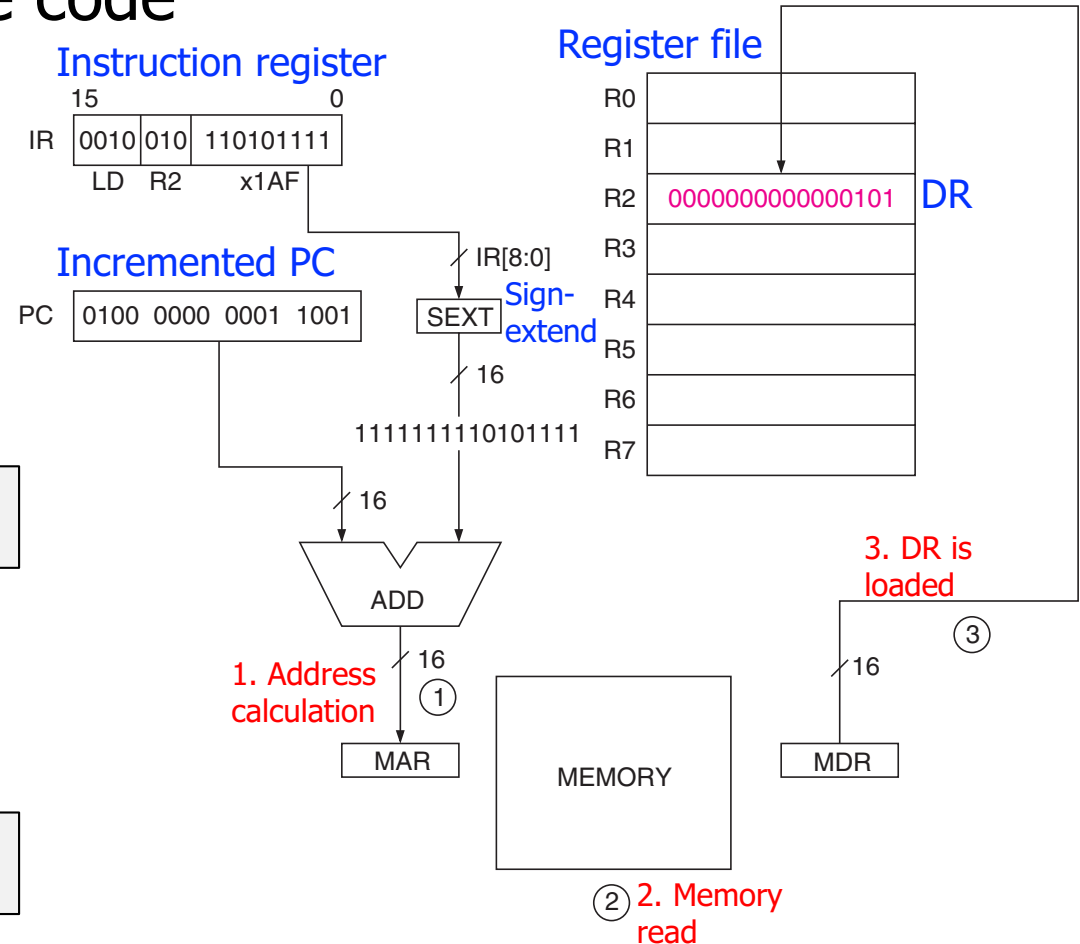
Field Values

OP	DR	PCOffset9
2	2	0x1AF

Machine Code

OP	DR	PCOffset9
0010	010	110101111

15 12 11 9 8 0

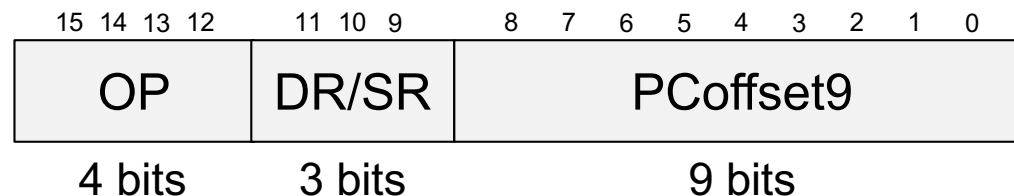


The memory address is **only +256 to -255** locations away of the **LD or ST** instruction

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: $\text{Memory}[\text{Memory}[\text{PC}^\dagger + \text{sign-extend}(\text{PCOffset9})]] \leftarrow \text{SR}$

[†] This is the incremented PC

LDI in LC-3

■ LDI assembly and machine code

LC-3 assembly

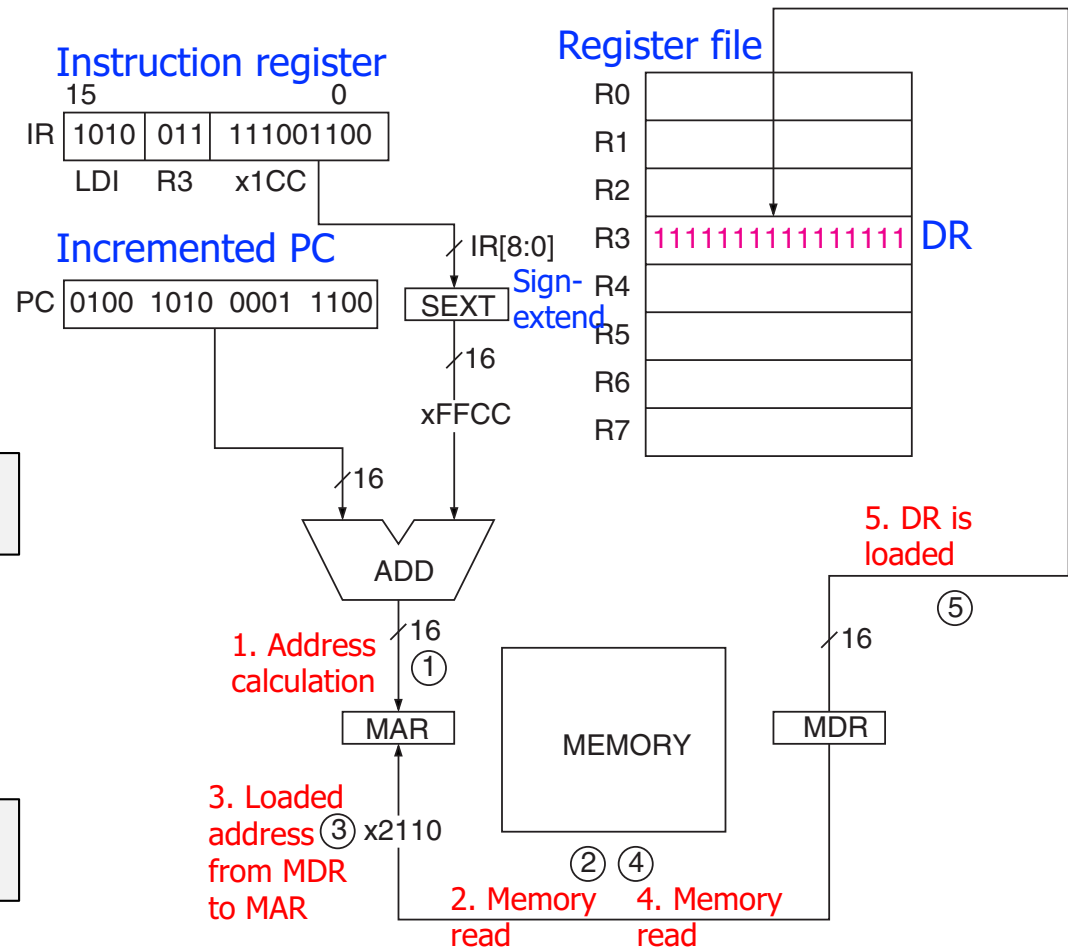
```
LDI R3, 0x1CC
```

Field Values

OP	DR	PCOffset9
A	3	0x1CC

Machine Code

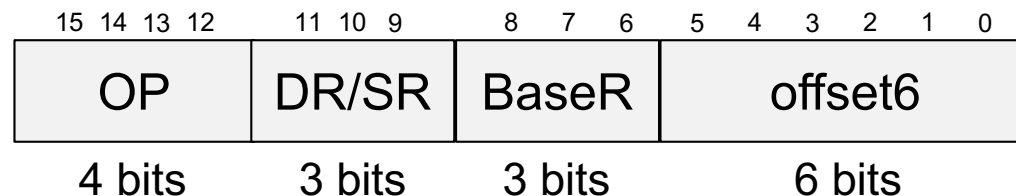
OP	DR	PCOffset9
1 0 1 0	0 1 1	1 1 1 0 0 1 1 0 0
15	12	11 9 8 0



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

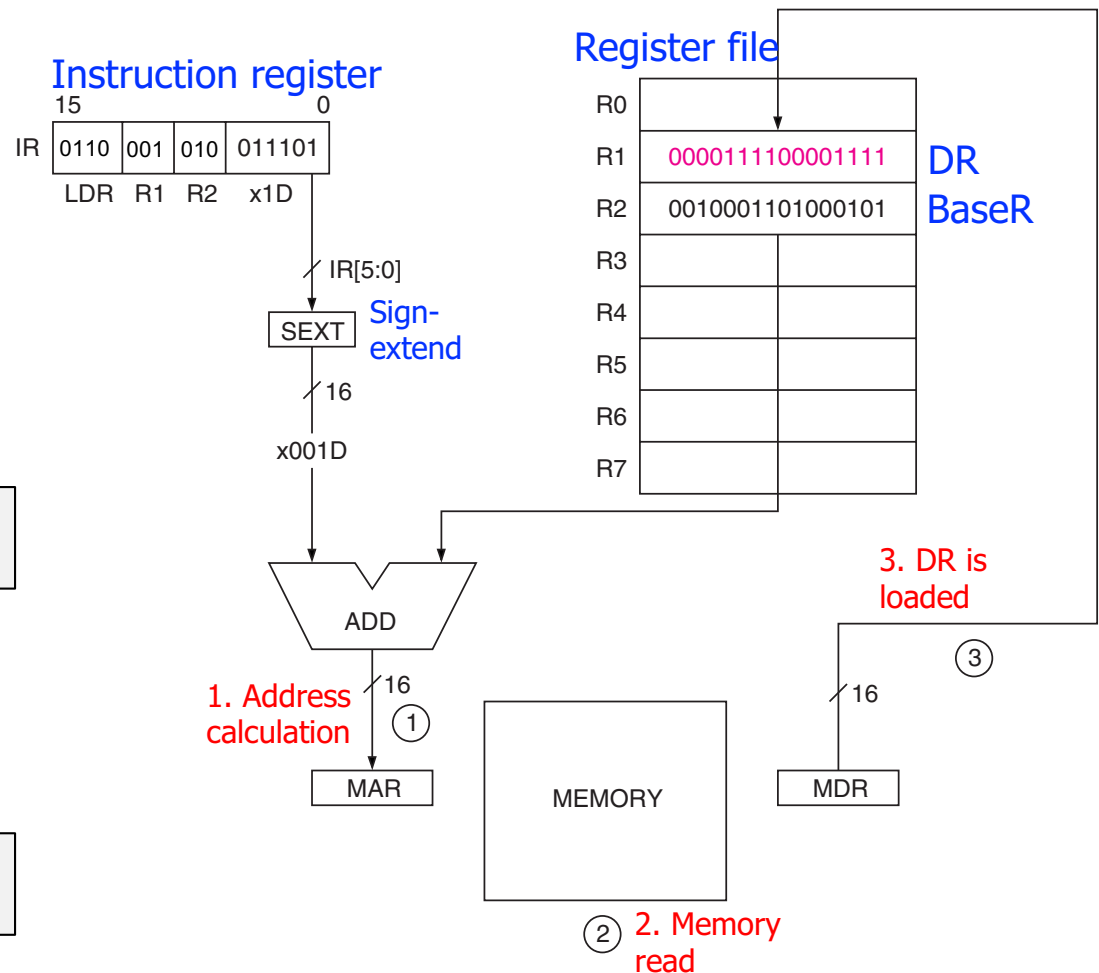
```
LDR R1, R2, 0x1D
```

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



The address of the operand can also 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

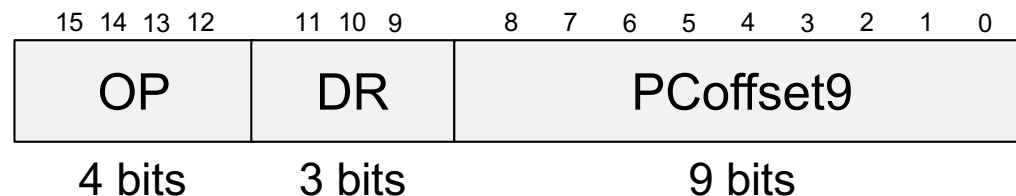
```
lw    $t0, 0($s0)  
add   $t1, $t0, $s2  
addi  $t2, $t1, -5  
sw    $t2, 0($s1)
```

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**

[†] This is the incremented PC

LEA in LC-3

■ LEA assembly and machine code

LC-3 assembly

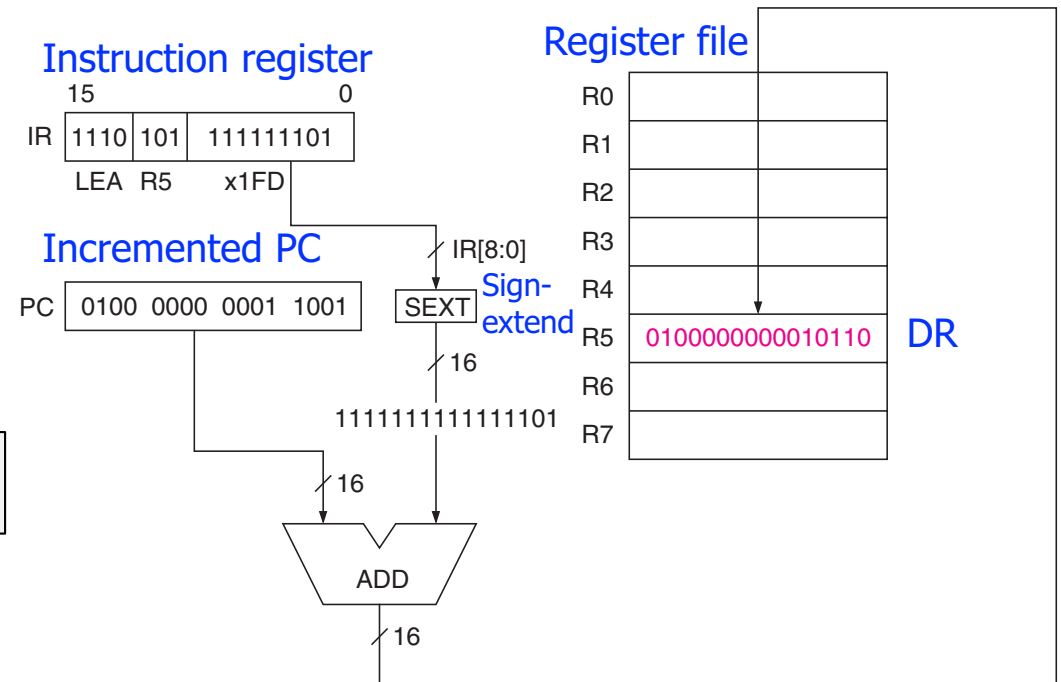
```
LEA R5, #-3
```

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?

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	0	1	1	1	1	R3 ← M[M[x30F4]]

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	3	1	1	1	1	1	1	0	1	$R1 = PC^+ - 3 = 0x30F7 - 3 = 0x30F4$
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	5	1	1	1	1	1	1	0	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	5	0	1	0	1	$R2 = R2 + 5 = 5$
x30FB	STR	1	1	1	0	1	0	0	0	1	0	0	0	1	1	0	$M[R1 + 14] = M[0x30F4 + 14] = M[0x3102] = 5$
x30FC	LDI	0	1	0	0	1	1	9	1	1	1	1	0	1	1	1	$R3 = M[M[PC^+ - 9]] = M[M[0x30FD - 9]] =$ $M[M[0x30F4]] = M[0x3102] = 5$

- The final value of **R3** is 5

Control Flow Instructions

Control Flow Instructions

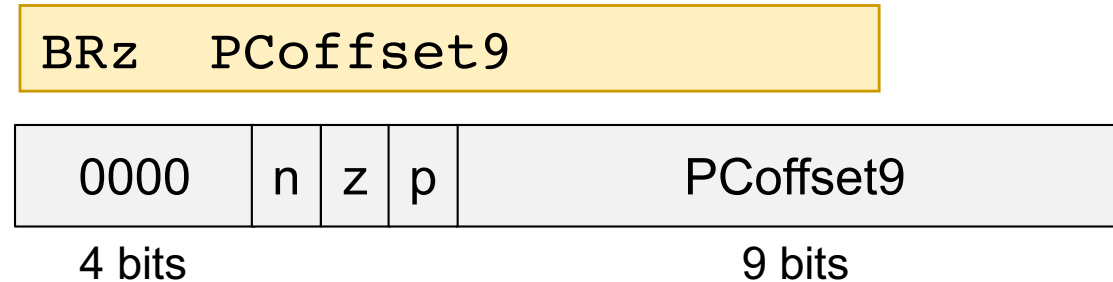
- Allow a program to execute **out of sequence**
- Conditional branches and jumps
 - **Conditional branches** are used to **make decisions**
 - E.g., if-else statement
 - In LC-3, three **condition codes** are used
 - **Jumps** are used to implement
 - **Loops**
 - **Function calls**
 - **JMP** in LC-3 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 N, Z, and/or P is tested
- PCoffset9 = immediate or constant value
- if $((n \text{ AND } N) \text{ OR } (p \text{ AND } P) \text{ OR } (z \text{ AND } Z))$
 - then $PC \leftarrow PC^{\dagger} + \text{sign-extend}(\text{PCoffset9})$
- Variations: BRn, BRz, BRp, BRzp, BRnp, BRnz, BRnzp

[†] This is the incremented PC

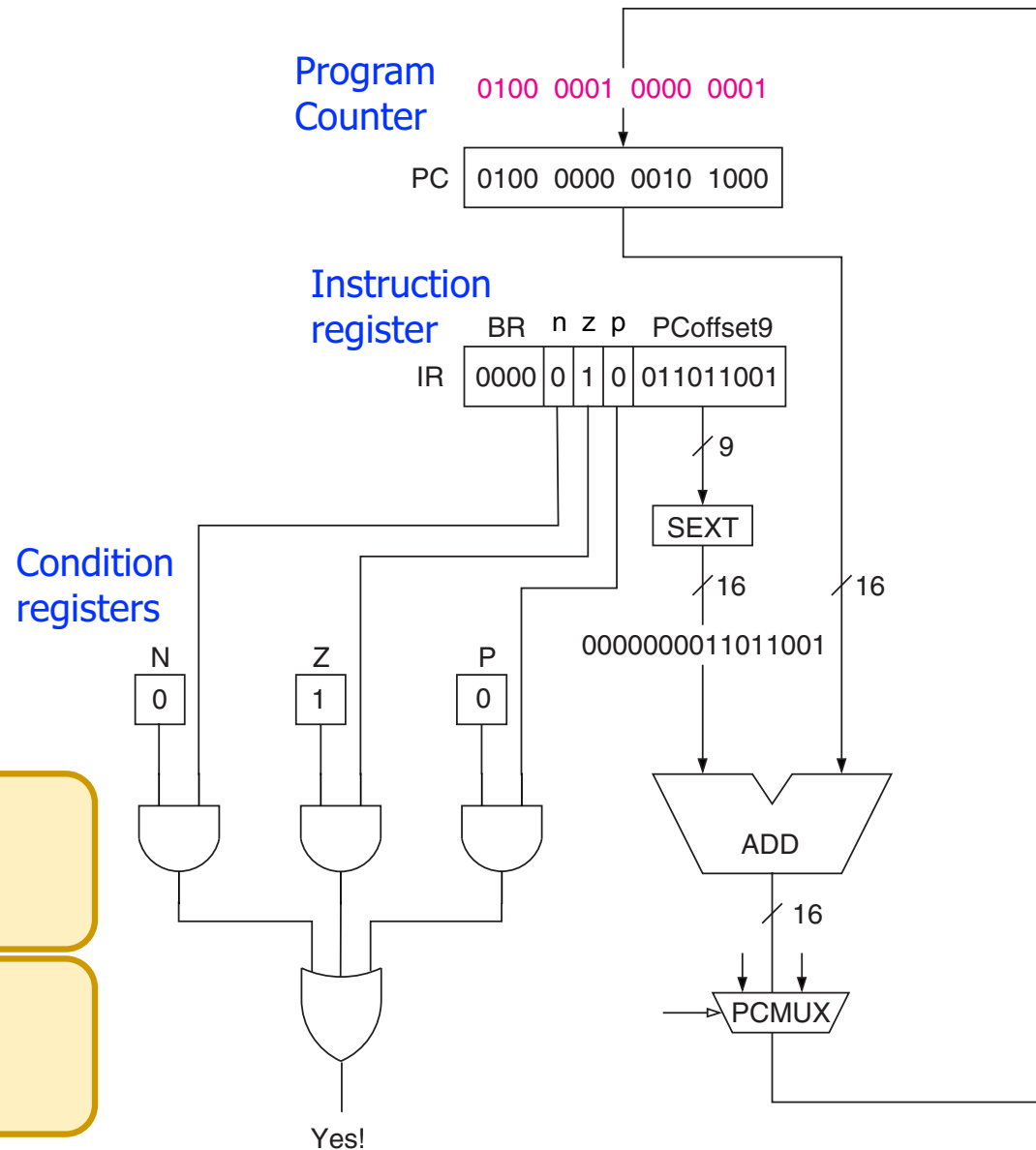
Conditional Branches in LC-3

■ BRz

BRz 0x0D9

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

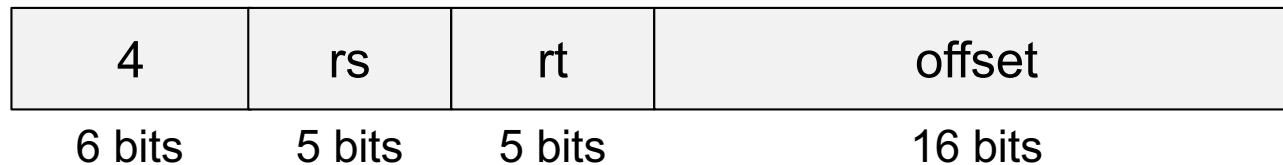
And what if $n = z = p = 0$?



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
```

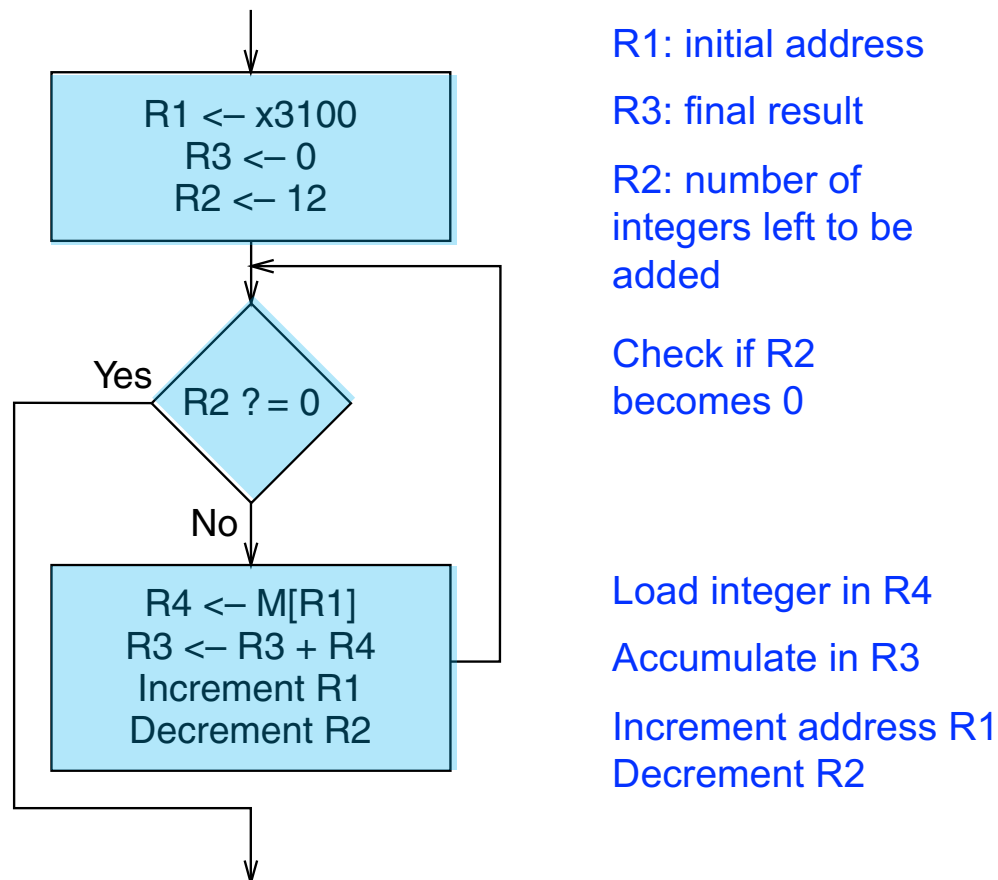
**Subtract
(R0 - R1)**

- 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**

Use of Conditional Branches for Looping

An Algorithm for Adding Integers

- We want to write a program that adds 12 integers
 - They are stored in addresses 0x3100 to 0x310B
 - Let us take a look at the flowchart of the algorithm



A Program for Adding Integers in LC-3

- We use **conditional branch instructions** to create a **loop**

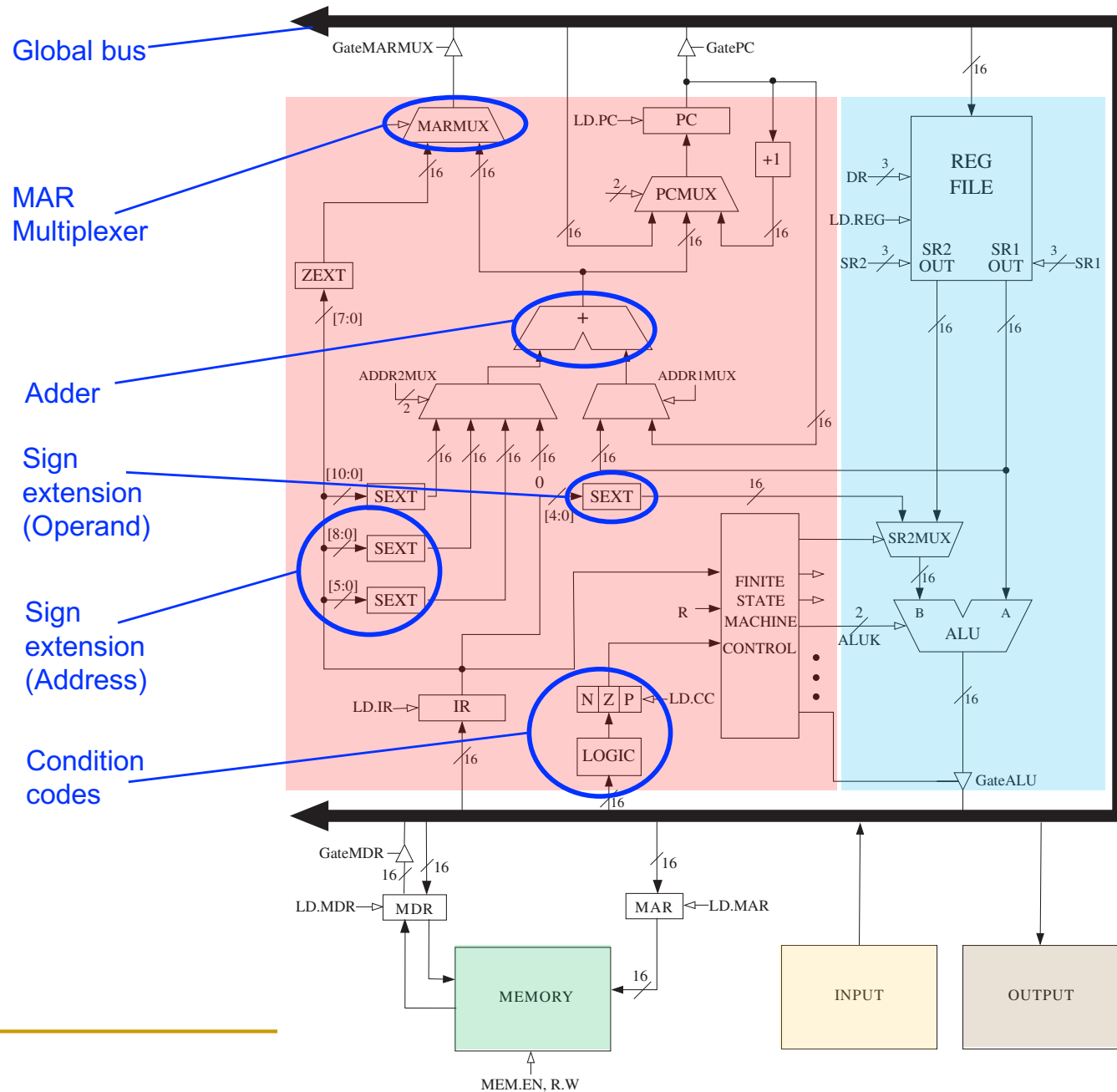
Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	LEA	1	1	0	0	0	1	0x00FF	1	1	1	1	1	1	1	1	R1 = PC [†] + 0x00FF = 3100
x3001	AND	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0	R3 = 0
x3002	AND	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 = 0
x3003	ADD	0	0	1	0	1	0	0	1	0	1	0	1	1	0	0	R2 = R2 + 12
x3004	BR	0	0	0	0	z	1	5	0	0	0	0	0	0	1	0	BRz (PC [†] + 5) = BRz 0x300A
x3005	LDR	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	R4 = M[R1 + 0]
x3006	ADD	0	0	1	0	1	1	0	1	1	0	0	0	1	0	0	R3 = R3 + R4
x3007	ADD	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	R1 = R1 + 1
x3008	ADD	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	R2 = R2 - 1
x3009	BR	0	0	0	n	z	1	p	1	6	1	1	1	1	0	1	BRnzp (PC [†] - 6) = BRnzp 0x3004

[†] This is the incremented PC

The LC-3 Data Path Revisited

The LC-3 Data Path

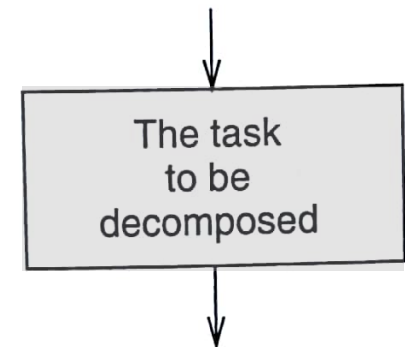
We highlight some data path components used in the execution of the instructions in the previous slides (not shown in the simplified data path)



(Assembly) Programming

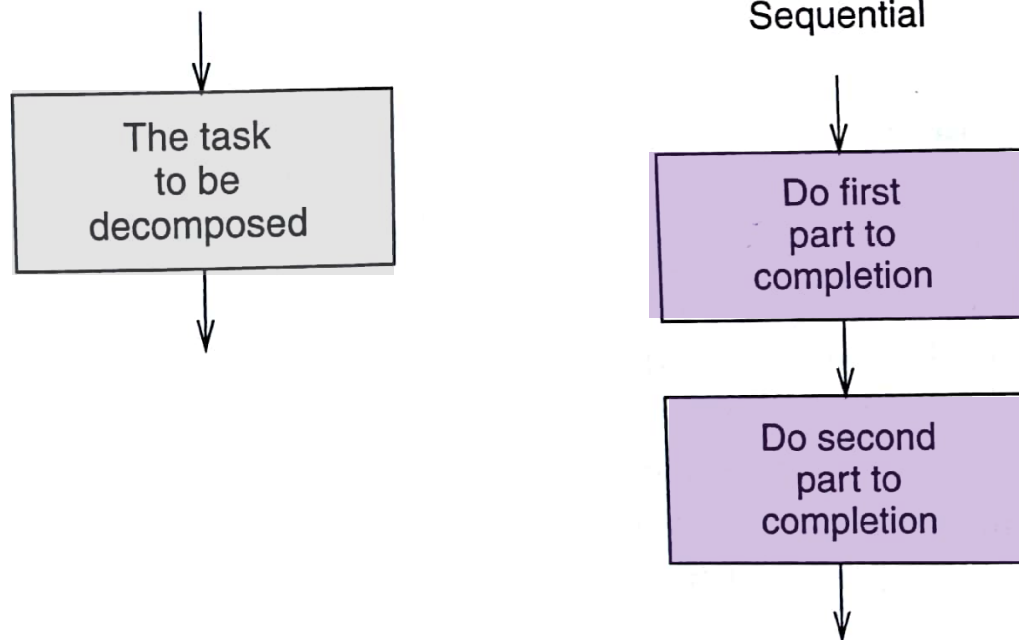
Programming Constructs

- Programming requires **dividing a task**, i.e., a unit of work into **smaller units of work**
- The goal is to replace the units of work with **programming constructs** that represent that part of the task
- There are **three basic programming constructs**
 - Sequential construct
 - Conditional construct
 - Iterative construct



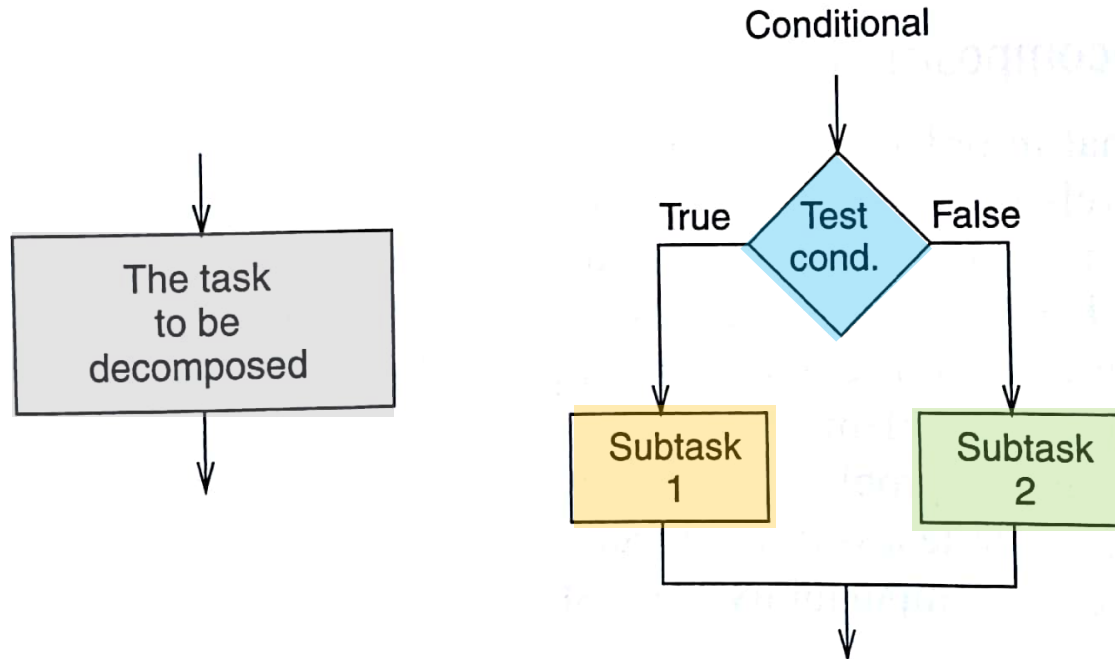
Sequential Construct

- The sequential construct is used if the designated task can be broken down into two subtasks, one following the other



Conditional Construct

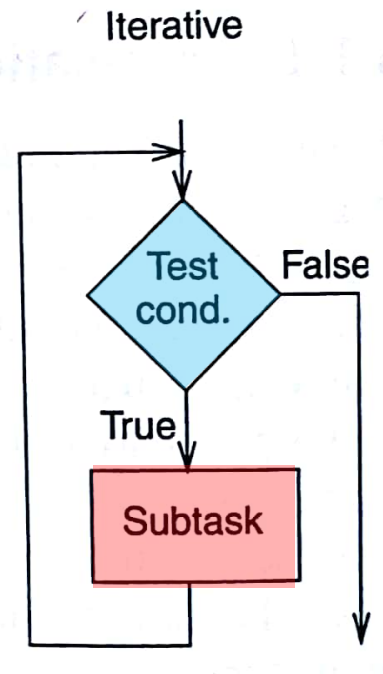
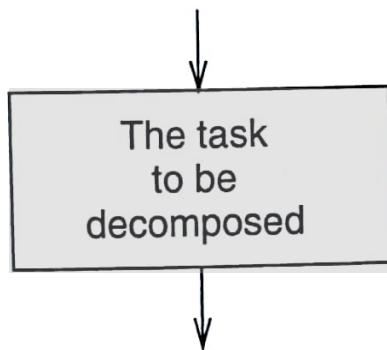
- The conditional construct is used if the designated task consists of **doing one of two subtasks**, **but not both**



- Either subtask may be "do nothing"
- **After** the correct subtask is **completed**, the program **moves onward**
- E.g., if-else statement, switch-case statement

Iterative Construct

- The iterative construct is used if the designated task consists of **doing a subtask a number of times**, but only **as long as some condition is true**



- E.g., for loop, while loop, do-while loop

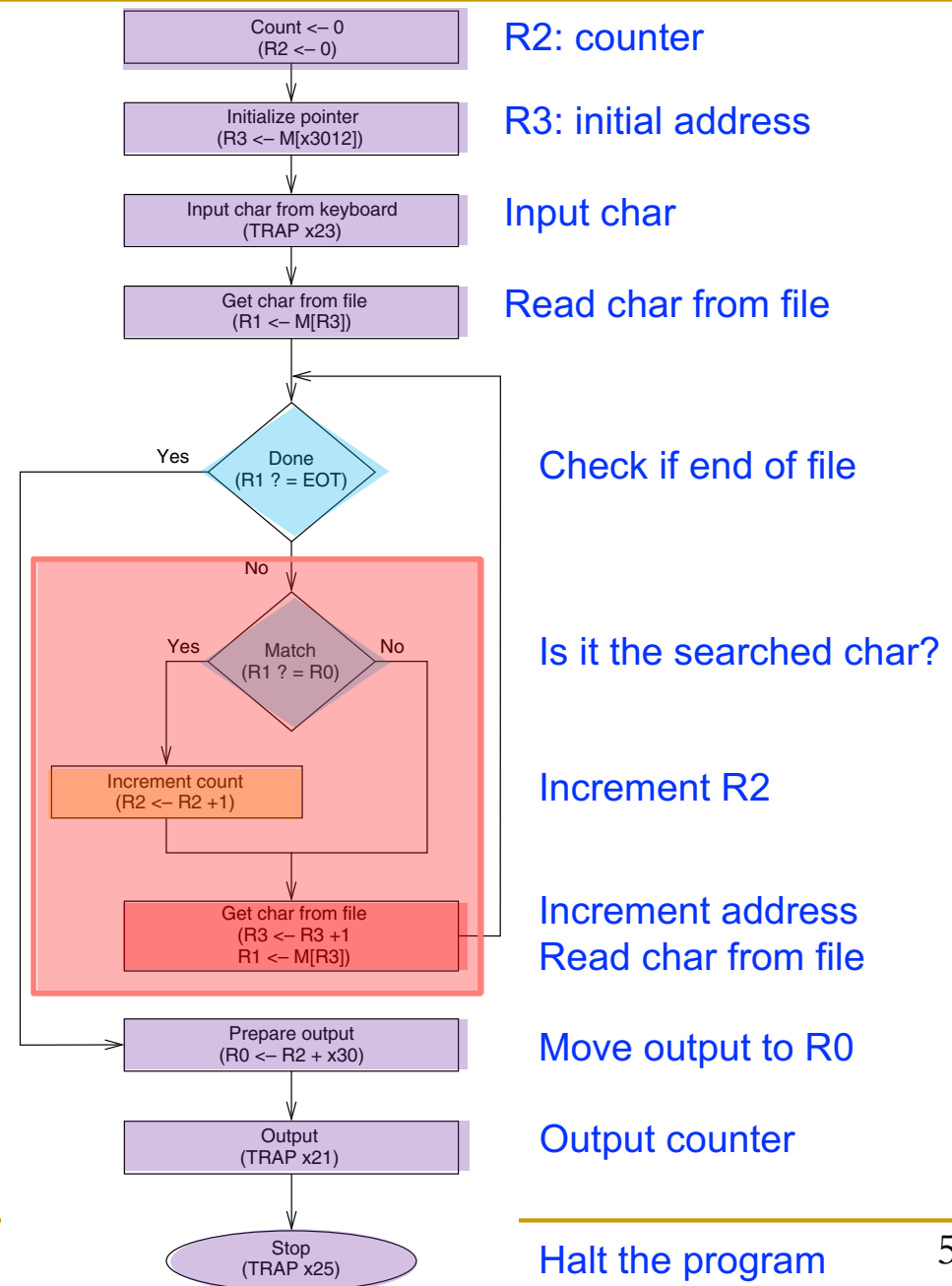
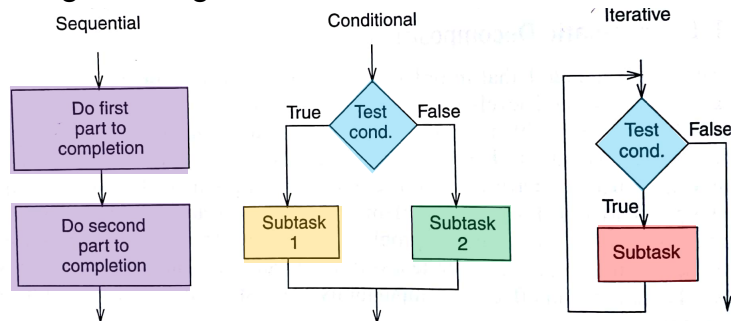
Constructs in an Example Program

- Let us see how to use the programming constructs in an example program
- The example program counts the number of occurrences of a character in a text file
- It uses sequential, conditional, and iterative constructs
- We see how to write conditional and iterative constructs with conditional branches

Counting Occurrences of a Character

- We want to write a program that counts the occurrences of a character in a file
 - Character from the keyboard (TRAP instr.)
 - The file finishes with the character EOT (End Of Text)
 - That is called a sentinel
 - In this example, EOT = 4
 - Result to the monitor (TRAP instr.)

Programming constructs



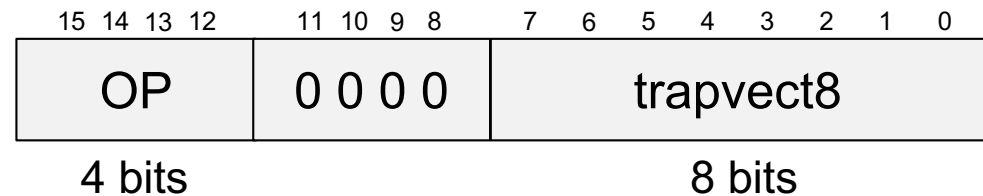
TRAP Instruction

- TRAP invokes an OS service call

LC-3 assembly

```
TRAP 0x23;
```

Machine Code



- OP = 1111
- **trapvect8** = service call
 - 0x23 = **Input a character** from the keyboard
 - 0x21 = **Output a character** to the monitor
 - 0x25 = **Halt** the program

Counting Occurrences of a Char in LC-3

- We use **conditional branch instructions** to create a **loops** and **if statements**

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 = 0 // initialize counter
x3001	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	0	R3 = M[0x3012] // initial address
x3002	1	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	TRAP 0x23 // input char to R0
x3003	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1 = M[R3] // char from file
x3004	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	0	R4 = R1 - 4 // char - EOT
x3005	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	BRz 0x300E // check if end of file
x3006	1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	R1 = NOT(R1) // subtract char from
x3007	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1	R1 = R1 + 1 // file from input char
x3008	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	R1 = R1 + R0 // for comparison
x3009	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	BRnp 0x300B
x300A	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	R2 = R2 + 1 // increment the counter
x300B	0	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1	R3 = R3 + 1 // increment address
x300C	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0	R1 = M[R3] // char from file
x300D	0	0	0	0	1	1	1	1	1	1	1	1	0	1	1	0	BRnzp 0x3004
x300E	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	R0 = M[0x3013]
x300F	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	R0 = R0 + R2 // output counter
x3010	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	TRAP 0x21
x3011	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	TRAP 0x25
x3012	Starting address of file																
x3013	ASCII TEMPLATE 0 0 0 0 0 1 1 0 0 0 0																

Programming Constructs in LC-3

■ Conditional constructs and iterative constructs

□ Let us do some reverse engineering

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x3000	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0
x3001	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0	0
x3002	1	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1
x3003	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0
x3004	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	0
x3005	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
x3006	1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1
x3007	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	1
x3008	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0
x3009	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1
x300A	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1
x300B	0	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1
x300C	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0
x300D	0	0	0	0	1	1	1	1	1	1	1	1	0	1	1	0
x300E	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
x300F	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
x3010	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1
x3011	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1
x3012	Starting address of file															
x3013	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0

```
do
{
...
}while(R1 != EOT);
```

R4 = R1 - 4 // char - EOT

BRz 0x300E // check if end of file

R1 = NOT(R1) // subtract char from
R1 = R1 + 1 file from input char
R1 = R1 + R0 for comparison

BRnp 0x300B

R2 = R2 + 1 // increment the counter

BRnzp 0x3004

```
if (R1 == R0)
{
... // increment the counter
}
```

Debugging

Debugging

- Debugging is the process of **removing errors in programs**
- It consists of **tracing the program**, i.e., keeping track of the **sequence of instructions** that have been executed and the **results** produced by each instruction
- A useful technique is to partition the program into parts, often referred to as **modules**, and examine the results computed in each module
- High-level language (e.g., C programming language) debuggers: dbx, gdb, visual studio debugger
- Machine code debugging: **Elementary interactive debugging operations**

Interactive Debugging

- When debugging interactively, it is important to be able to
 - 1. Deposit values in memory and in registers, in order to test the execution of a part of a program in isolation
 - 2. Execute instruction sequences in a program by using
 - RUN command: execute until HALT instruction or a breakpoint
 - STEP command: execute a fixed number of instructions
 - 3. Stop execution when desired
 - SET BREAKPOINT command: stop execution at a specific instruction in a program
 - 4. Examine what is in memory and registers at any point in the program

Example: Multiplying in LC-3

- A program is necessary to multiply, since LC-3 does not have multiply instruction
 - The following program multiplies R4 and R5
 - Initially, R4 = 10 and R5 = 3
 - The program produces 40. What went wrong?
 - It is useful to annotate each instruction

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3200	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 = 0 // initialize register
x3201	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	R2 = R2 + R4
x3202	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	R5 = R5 - 1
x3203	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	BRzp 0x3201
x3204	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT // end program

Debugging the Multiply Program

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3200	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 = 0 // initialize register
x3201	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	R2 = R2 + R4
x3202	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	R5 = R5 - 1
x3203	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	BRzp 0x3201
x3204	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT // end program

- We examine the contents of all registers after the execution of each instruction

PC	R2	R4	R5	
x3201	0	10	3	
x3202	10	10	3	
x3203	10	10	2	
x3201	10	10	2	
x3202	20	10	2	
x3203	20	10	1	
x3201	20	10	1	
x3202	30	10	1	← Correct result
x3203	30	10	0	← BR should not be taken if R5 = 0
x3201	30	10	0	
x3202	40	10	0	
x3203	40	10	-1	
x3204	40	10	-1	
	40	10	-1	

The branch condition codes were set wrong. The conditional branch should only be taken if R5 is positive

Correct instruction:

BRp #-3 // BRp 0x3201

Easier Debugging with Breakpoints

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3200	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	R2 = 0 // initialize register
x3201	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	R2 = R2 + R4
x3202	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	R5 = R5 - 1
x3203	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	BRzp 0x3201
x3204	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	HALT // end program

- We could use a **breakpoint** to save some work
- Setting a breakpoint in 0x3203 (BR) allows us to examine the **results of each iteration of the loop**

PC	R2	R4	R5	
x3203	10	10	2	
x3203	20	10	1	
x3203	30	10	0	← BR should not be taken if R5 = 0
x3203	40	10	-1	

One last question:
Does this program work if
the initial value of R5 is 0?

A good test should also consider the **corner cases**,
i.e., unusual values that the programmer might fail to consider

Conditional Statements and Loops in MIPS Assembly

If Statement

- In MIPS, we create **conditional constructs** with **conditional branches** (e.g., beq, bne...)

High-level code

```
if (i == j)
    f = g + h;

f = f - i;
```

MIPS assembly

```
# $s0 = f, $s1 = g
# $s2 = h
# $s3 = i, $s4 = j

    bne $s3, $s4, L1
    add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

If-Else Statement

- We use the **unconditional branch** (i.e., `j`) to skip the **"else"** subtask if the **"if"** subtask is the correct one

High-level code

```
if (i == j)
    f = g + h;
else
    f = f - i;
```

MIPS assembly

```
# $s0 = f, $s1 = g,
# $s2 = h
# $s3 = i, $s4 = j

        bne $s3, $s4, L1
        add $s0, $s1, $s2
        j   done
L1:     sub $s0, $s0, $s3
done:
```


While Loop

- As in LC-3, the **conditional branch** (i.e., beq) checks the condition and the **unconditional branch** (i.e., j) jumps to the beginning of the loop

High-level code

```
// determines the power
// of 2 equal to 128
int pow = 1;
int x    = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

MIPS assembly

```
# $s0 = pow, $s1 = x

        addi $s0, $0, 1
        add  $s1, $0, $0
        addi $t0, $0, 128
while:   beq  $s0, $t0, done
        sll  $s0, $s0, 1
        addi $s1, $s1, 1
        j   while
done:
```

For Loop

- The implementation of the "for" loop is similar to the "while" loop

High-level code

```
// add the numbers from 0 to 9

int sum = 0;
int i;
for (i = 0; i != 10; i = i+1)
{
    sum = sum + i;
}
```

MIPS assembly

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        add  $s0, $0, $0
        addi $t0, $0, 10
for:     beq  $s0, $t0, done
        add  $s1, $s1, $s0
        addi $s0, $s0, 1
        j    for
done:
```

For Loop Using SLT

- We use **slt** (i.e., set less than) for the “less than” comparison

High-level code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i = 1; i < 101; i = i*2)
{
    sum = sum + i;
}
```

MIPS assembly

```
# $s0 = i, $s1 = sum

        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

Set less than

$\$t1 = \$s0 < \$t0 ? 1:0$

Shift left logical

Arrays in MIPS

Arrays

- Accessing an array requires loading the base address into a register

0x12340010	array[4]
0x1234800C	array[3]
0x12348008	array[2]
0x12348004	array[1]
0x12348000	array[0]

- In MIPS, this is something we cannot do with one single immediate operation
- Load upper immediate + OR immediate

```
lui    $s0, 0x1234
ori    $s0, $s0, 0x8000
```

Arrays: Code Example

- We first load the **base address of the array** into a register (e.g., \$s0) using **lui** and **ori**

High-level code

```
int array[5];

array[0] = array[0] * 2;

array[1] = array[1] * 2;
```

MIPS assembly

```
# array base address = $s0
# Initialize $s0 to 0x12348000
lui    $s0, 0x1234
ori     $s0, $s0, 0x8000

lw      $t1, 0($s0)
sll     $t1, $t1, 1
sw      $t1, 0($s0)
lw      $t1, 4($s0)
sll     $t1, $t1, 1
sw      $t1, 4($s0)
```

Function Calls

Function Calls

- Why functions (i.e., procedures)?
 - Frequently accessed code
 - Make a program more modular and readable
- Functions have arguments and return value
- Caller: calling function
 - main()
- Callee: called function
 - sum()

```
void main()  
{  
    int y;  
    y = sum(42, 7);  
    ...  
}  
  
int sum(int a, int b)  
{  
    return (a + b);  
}
```


Function Calls: Conventions

- Conventions

- Caller

- passes arguments
 - jumps to callee

- Callee

- performs the procedure
 - returns the result to caller
 - returns to the point of call
 - must not overwrite registers or memory needed by the caller

Function Calls in MIPS and LC-3

- Conventions in MIPS and LC-3
 - Call procedure
 - MIPS: Jump and link (jal)
 - LC-3: Jump to Subroutine (JSR, JSRR)
 - Return from procedure
 - MIPS: Jump register (jr)
 - LC-3: Return from Subroutine (RET)
 - Argument values
 - MIPS: \$a0 - \$a3
 - Return value
 - MIPS: \$v0

Function Calls: Simple Example

High-level code

```
int main() {  
    simple();  
    a = b + c;  
}  
  
void simple() {  
    return;  
}
```

MIPS assembly

```
0x00400200 main: jal simple  
0x00400204          add $s0,$s1,$s2  
  
...  
0x00401020 simple: jr $ra
```

- **jal** jumps to **simple()** and saves PC+4 in the **return address register** (\$ra)
 - \$ra = 0x00400204
 - In LC-3, **JSR(R)** put the return address in **R7**
- **jr \$ra** jumps to address in \$ra (LC-3 uses **RET** instruction)

Function Calls: Code Example

High-level code

```
int main()
{
    int y;
    ...
    // 4 arguments
    y = diffofsums(2, 3, 4, 5);
    ...
}

int diffofsums(int f, int g,
               int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    // return value
    return result;
}
```

MIPS assembly

```
# $s0 = y
main:
    ...
    addi $a0, $0, 2    # argument 0 = 2
    addi $a1, $0, 3    # argument 1 = 3
    addi $a2, $0, 4    # argument 2 = 4
    addi $a3, $0, 5    # argument 3 = 5
    jal  diffofsums    # call procedure
    add  $s0, $v0, $0   # y = returned value
    ...

# $s0 = result
diffofsums:
    add $t0, $a0, $a1   # $t0 = f + g
    add $t1, $a2, $a3   # $t1 = h + i
    sub $s0, $t0, $t1   # result=(f + g) - (h + i)
    add $v0, $s0, $0    # put return value in $v0
    jr   $ra           # return to caller
```

Argument values

Return value

Return address

Function Calls: Need for the Stack

MIPS assembly

```
diffofsums:
    add $t0, $a0, $a1    # $t0 = f + g
    add $t1, $a2, $a3    # $t1 = h + i
    sub $s0, $t0, $t1    # result=(f + g) - (h + i)
    add $v0, $s0, $0      # put return value in $v0
    jr  $ra              # return to caller
```

- What if the main function was using some of those registers?
 - \$t0, \$t1, \$s0
- They could be **overwritten** by the function
- We can use the **stack** to temporarily store registers

The Stack

- The stack is a memory area used to **save local variables**
- It is a **Last-In-First-Out (LIFO)** queue
- The **stack pointer** (\$sp) points to the top of the stack
 - It grows down in MIPS

Address	Data		Address	Data	
7FFFFFFC	12345678	←\$sp	7FFFFFFC	12345678	Two words pushed on the stack
7FFFFFF8			7FFFFFF8	AABBCCDD	
7FFFFFF4			7FFFFFF4	11223344	←\$sp
7FFFFFF0			7FFFFFF0		
⋮	⋮		⋮	⋮	
⋮	⋮		⋮	⋮	
⋮	⋮		⋮	⋮	

The Stack: Code Example

MIPS assembly

```
diffofsums:
    addi $sp, $sp, -12    # allocate space on stack to store 3 registers
    sw   $s0, 8($sp)      # save $s0 on stack
    sw   $t0, 4($sp)      # save $t0 on stack
    sw   $t1, 0($sp)      # save $t1 on stack
    add  $t0, $a0, $a1     # $t0 = f + g
    add  $t1, $a2, $a3     # $t1 = h + i
    sub  $s0, $t0, $t1     # result=(f + g) - (h + i)
    add  $v0, $s0, $0      # put return value in $v0
    lw   $t1, 0($sp)      # restore $t1 from stack
    lw   $t0, 4($sp)      # restore $t0 from stack
    lw   $s0, 8($sp)      # restore $s0 from stack
    addi $sp, $sp, 12     # deallocate stack space
    jr   $ra              # return to caller
```

- Saving and restoring **all** registers requires a lot of effort
- In MIPS, there is a convention about **temporary registers** (i.e., \$t0-\$t9): There is **no need to save them**

The Stack: Nonpreserved Registers

MIPS assembly

```
diffofsums:
    addi $sp, $sp, -4    # allocate space on stack to store 1 register
    sw   $s0, 0($sp)     # save $s0 on stack

    add  $t0, $a0, $a1    # $t0 = f + g
    add  $t1, $a2, $a3    # $t1 = h + i
    sub  $s0, $t0, $t1    # result=(f + g) - (h + i)
    add  $v0, $s0, $0     # put return value in $v0

    lw   $s0, 0($sp)     # restore $s0 from stack
    addi $sp, $sp, 4     # deallocate stack space
    jr   $ra             # return to caller
```

- Temporary registers \$t0-\$t9 are **nonpreserved** registers
- Registers \$s0-\$s7 are **preserved** (saved) registers

Lecture Summary

- Instruction Set Architectures: LC-3 and MIPS
 - Operate instructions
 - Data movement instructions
 - Control instructions
- Instruction formats
- Addressing modes

- Assembly Programming
 - Programming constructs
 - Debugging
 - Conditional statements and loops in MIPS assembly
 - Arrays in MIPS assembly
 - Function calls
 - The stack

Design of Digital Circuits

Lecture 10: ISA (II)

and Assembly Programming

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zurich

Spring 2018

23 March 2018