

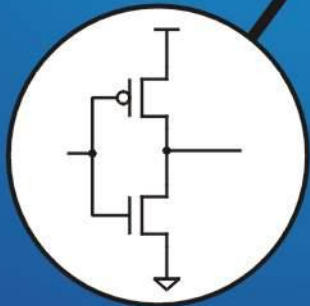
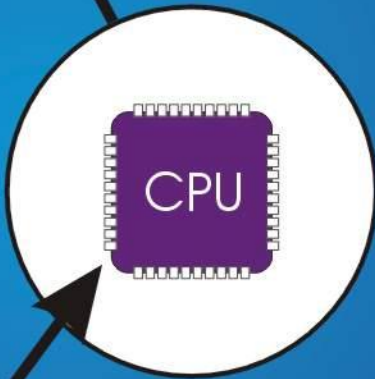
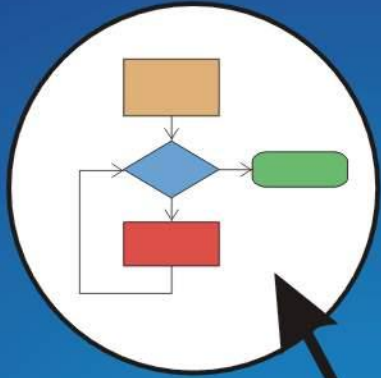


# **Introduction to Computer Engineering**

**CS/ECE 252, Spring 2013**

**Prof. Guri Sohi**

**Computer Sciences Department  
University of Wisconsin – Madison**



# Chapter 5

## The LC-3

# Instruction Set Architecture

**ISA** = All of the *programmer-visible* components and operations of the computer

- **memory organization**
  - address space -- how many locations can be addressed?
  - addressability -- how many bits per location?
- **register set**
  - how many? what size? how are they used?
- **instruction set**
  - opcodes
  - data types
  - addressing modes

ISA provides all information needed for someone that wants to write a program in **machine language** (or translate from a high-level language to machine language).

# LC-3 Overview: Memory and Registers

## Memory

- address space:  **$2^{16}$**  locations (16-bit addresses)
- addressability: **16 bits**

## Registers

- temporary storage, accessed in a single machine cycle
  - accessing memory generally takes longer than a single cycle
- eight general-purpose registers: **R0 - R7**
  - each **16 bits wide**
  - how many bits to uniquely identify a register?
- other registers
  - not directly addressable, but used by (and affected by) instructions
  - **PC** (program counter), **condition codes**

# LC-3 Overview: Instruction Set

## Opcodes

- 15 opcodes
- *Operate* instructions: ADD, AND, NOT
- *Data movement* instructions: LD, LDI, LDR, LEA, ST, STR, STI
- *Control* instructions: BR, JSR/JSRR, JMP, RTI, TRAP
- some opcodes set/clear *condition codes*, based on result:
  - N = negative, Z = zero, P = positive (> 0)

## Data Types

- 16-bit 2's complement integer

## Addressing Modes

- How is the location of an operand specified?
- non-memory addresses: *immediate, register*
- memory addresses: *PC-relative, indirect, base+offset*

# Operate Instructions

Only three operations: **ADD, AND, NOT**

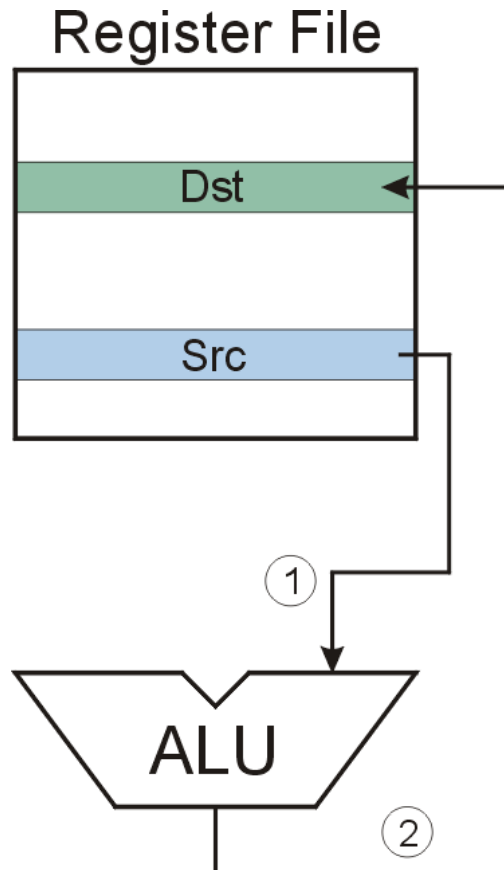
Source and destination operands are **registers**

- These instructions do not reference memory.
- ADD and AND can use “immediate” mode, where one operand is hard-wired into the instruction.

Will show **dataflow diagram** with each instruction.

- illustrates when and where data moves to accomplish the desired operation

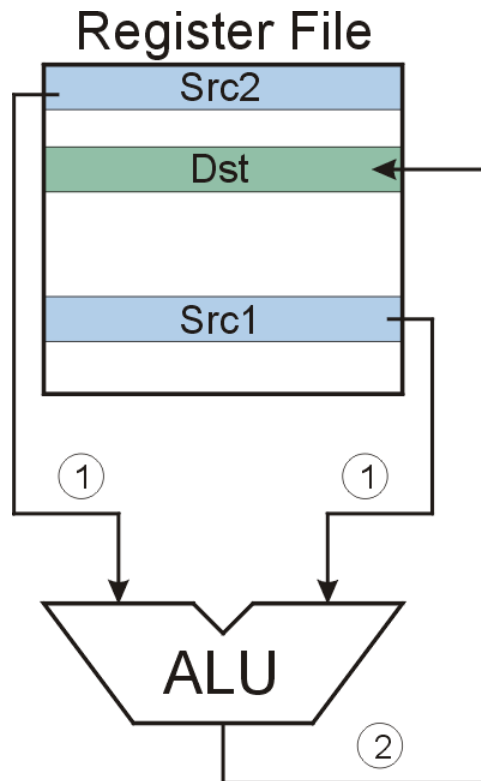
# NOT (Register)



*Note: Src and Dst could be the same register.*

# ADD/AND (Register)

*this zero means "register mode"*



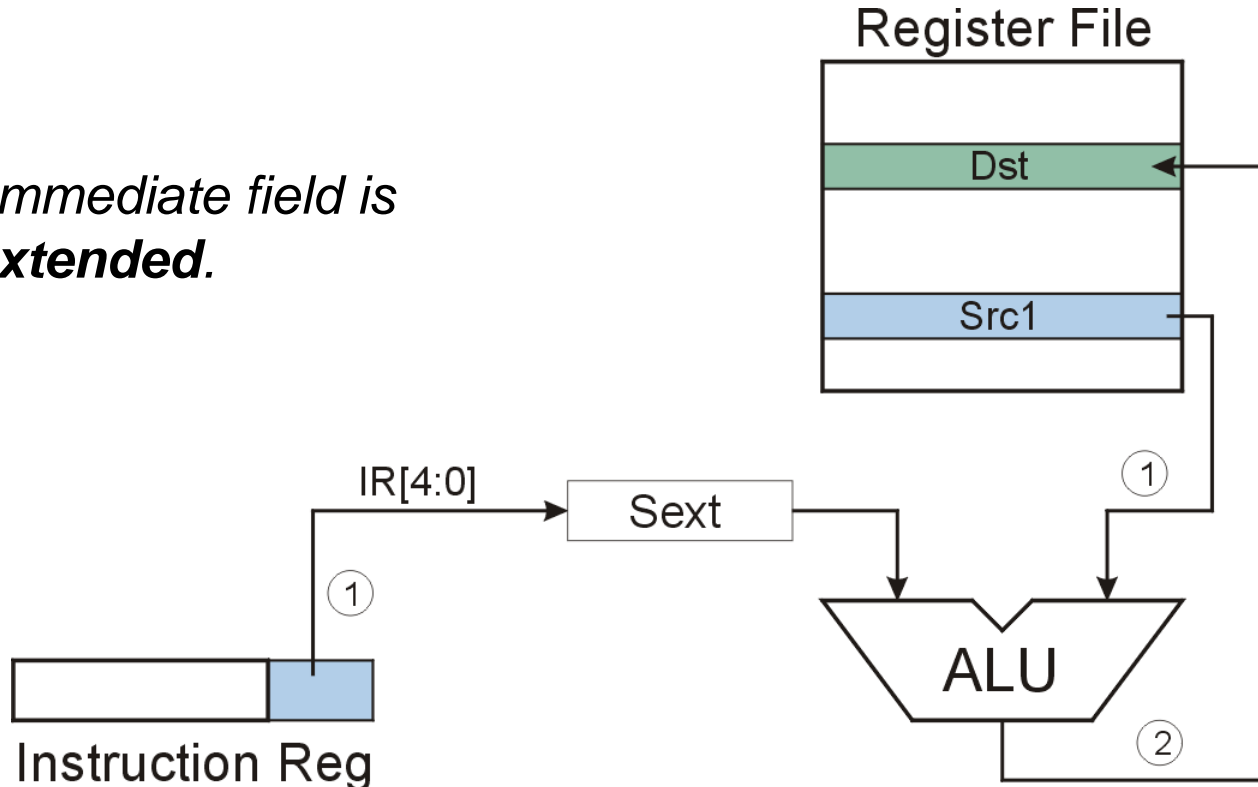


# ADD/AND (Immediate)

*this one means "immediate mode"*



*Note: Immediate field is **sign-extended**.*



# Using Operate Instructions

## With only ADD, AND, NOT...

- How do we subtract?
- How do we OR?
- How do we copy from one register to another?
- How do we initialize a register to zero?

# Data Movement Instructions

**Load -- read data from memory to register**

- **LD:** PC-relative mode
- **LDR:** base+offset mode
- **LDI:** indirect mode

**Store -- write data from register to memory**

- **ST:** PC-relative mode
- **STR:** base+offset mode
- **STI:** indirect mode

**Load effective address -- compute address, save in register**

- **LEA:** immediate mode
- *does not access memory*

## PC-Relative Addressing Mode

Want to specify address directly in the instruction

- But an address is 16 bits, and so is an instruction!
- After subtracting 4 bits for opcode and 3 bits for register, we have 9 bits available for address.

### Solution:

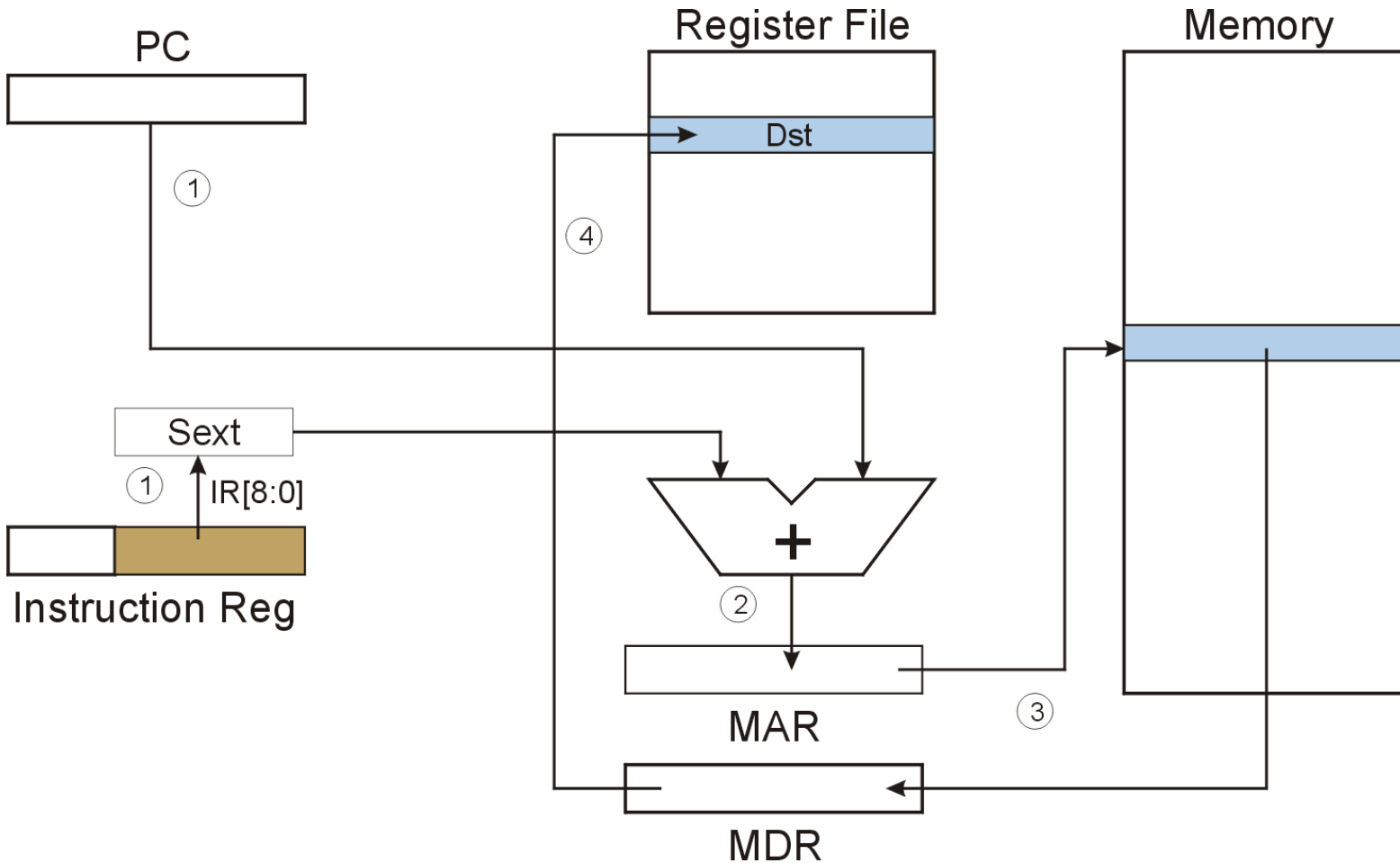
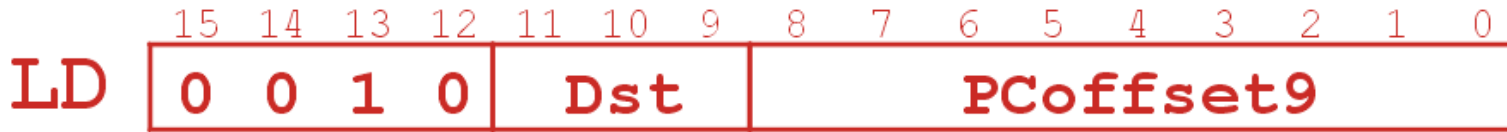
- Use the 9 bits as a signed offset from the current PC.

**9 bits:**  $-256 \leq \text{offset} \leq +255$

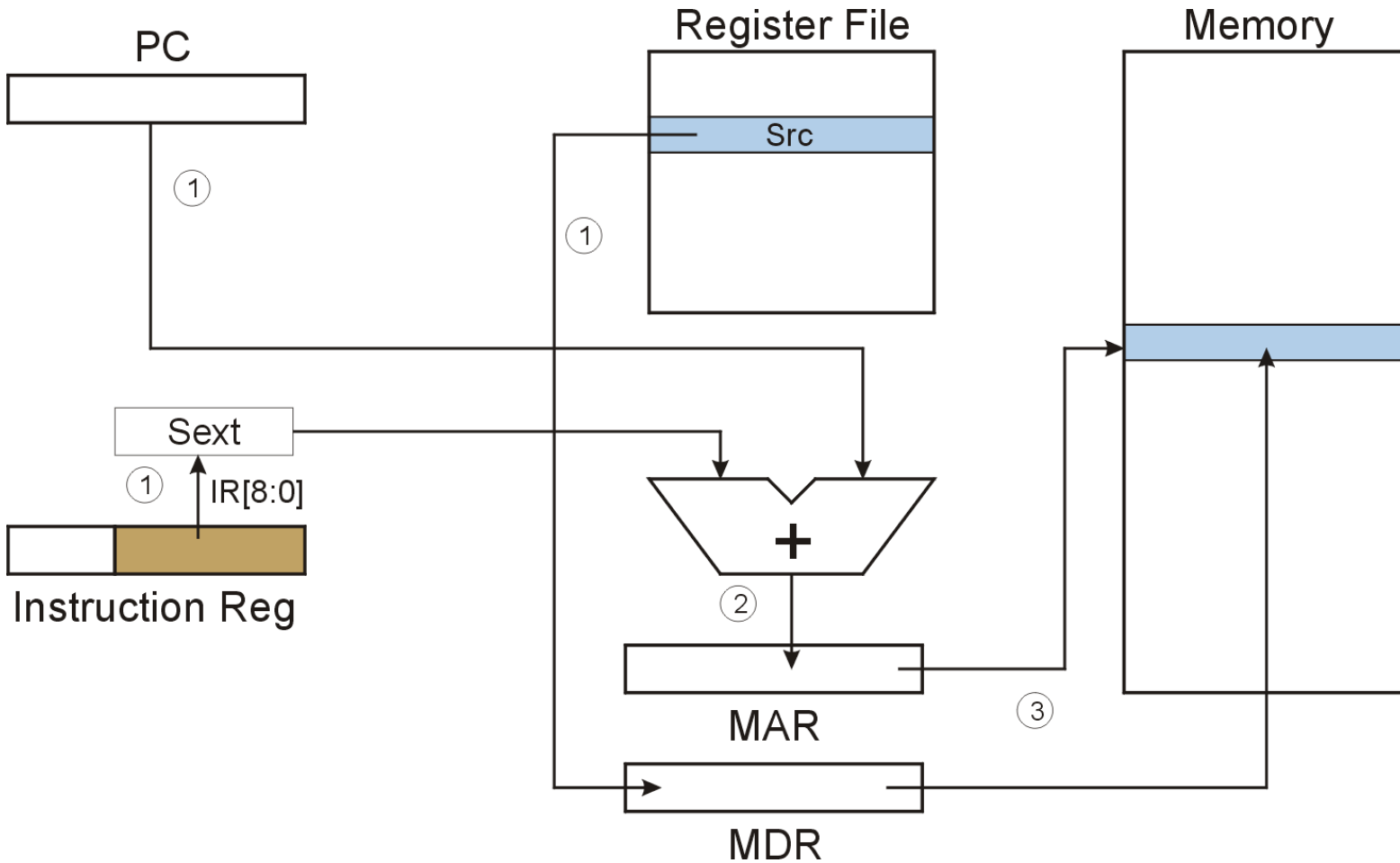
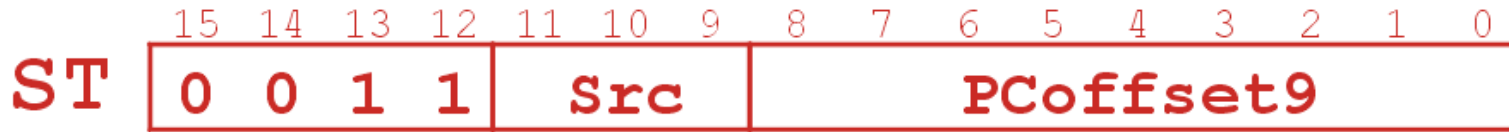
**Can form any address X, such that:**  $PC - 256 \leq X \leq PC + 255$

Remember that PC is incremented as part of the FETCH phase;  
This is done before the EVALUATE ADDRESS stage.

# LD (PC-Relative)



# ST (PC-Relative)



## Indirect Addressing Mode

**With PC-relative mode, can only address data within 256 words of the instruction.**

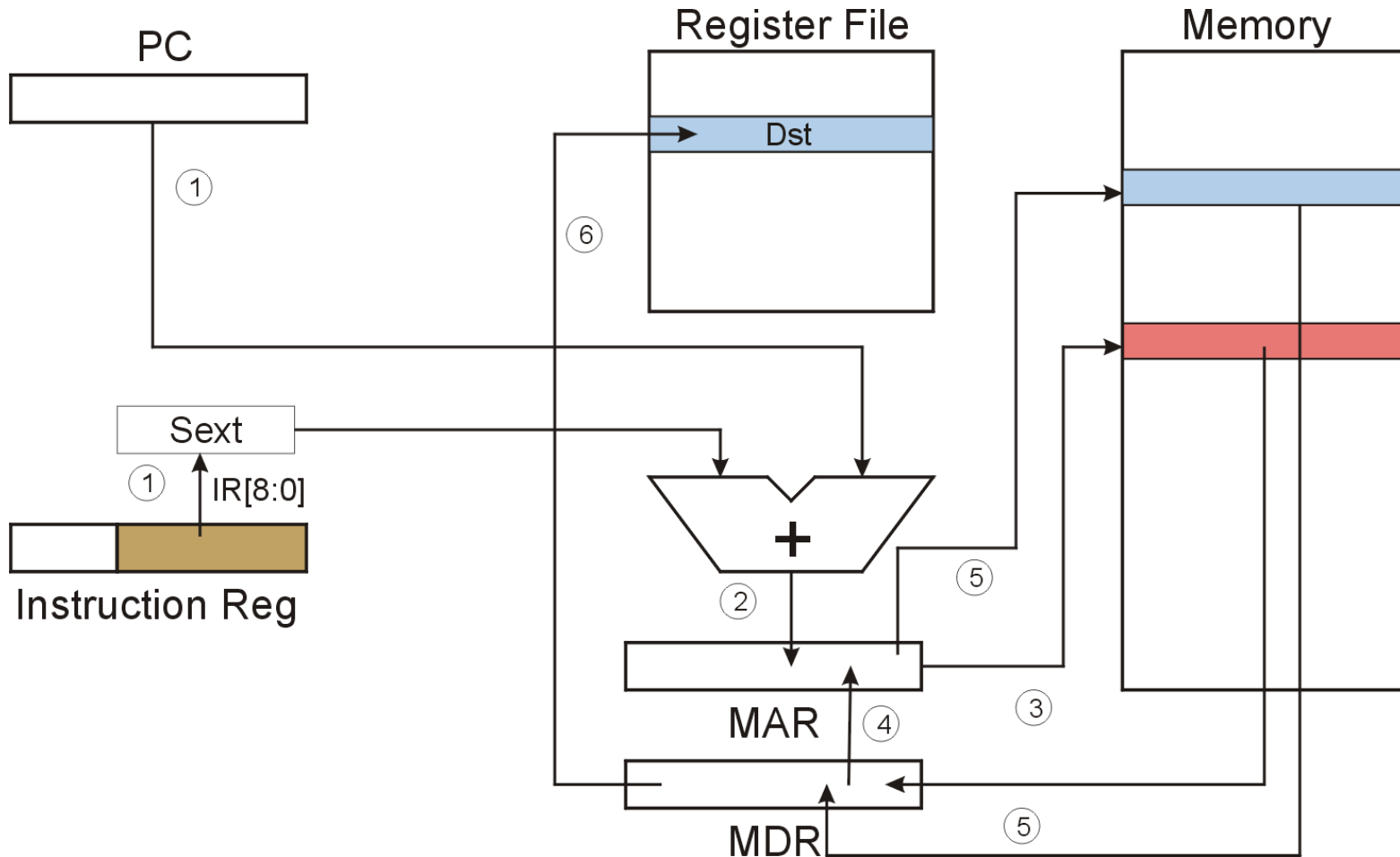
- What about the rest of memory?

### **Solution #1:**

- Read address from memory location, then load/store to that address.

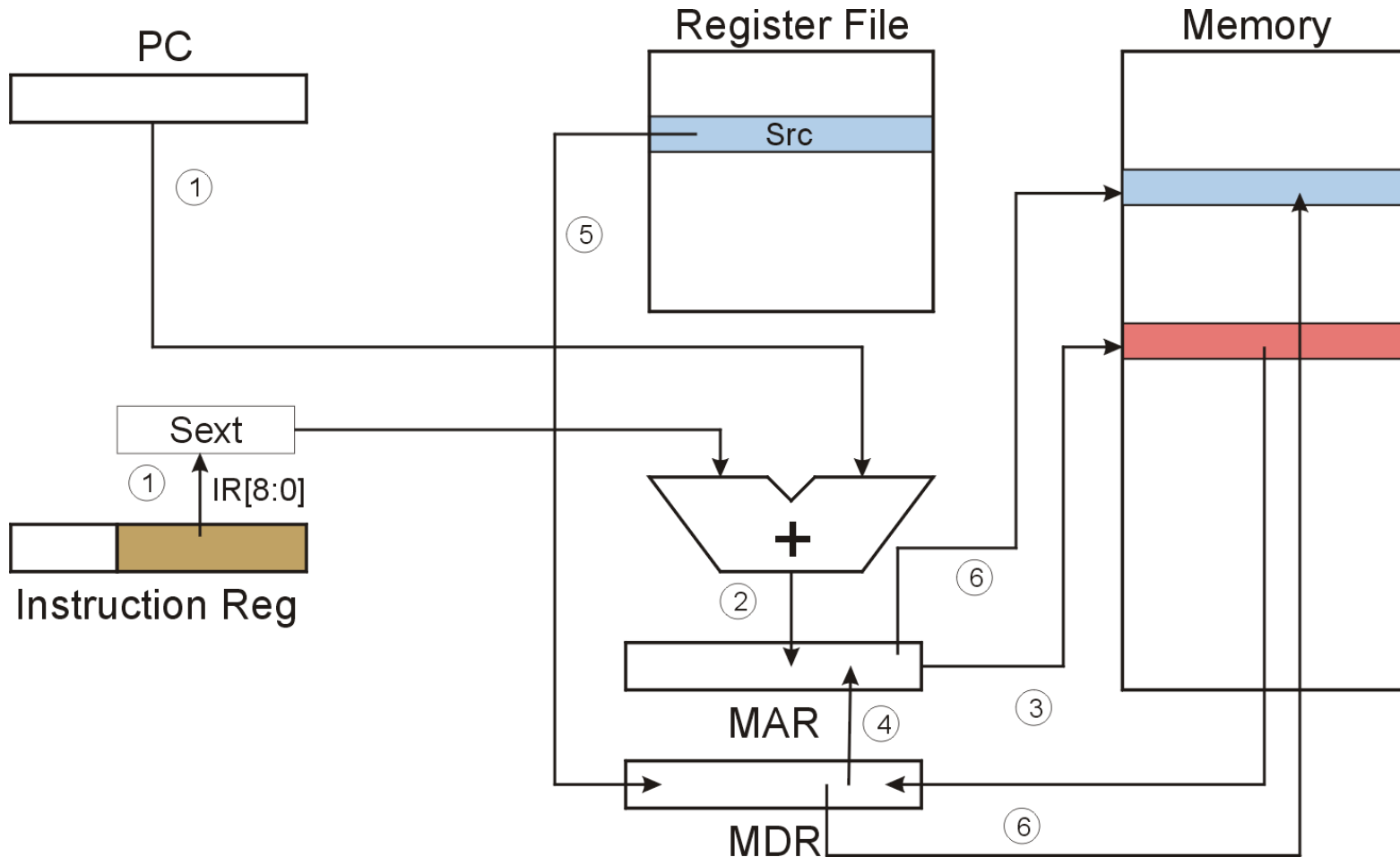
**First address is generated from PC and IR (just like PC-relative addressing), then content of that address is used as target for load/store.**

# LDI (Indirect)





# STI (Indirect)



## Base + Offset Addressing Mode

With PC-relative mode, can only address data within 256 words of the instruction.

- What about the rest of memory?

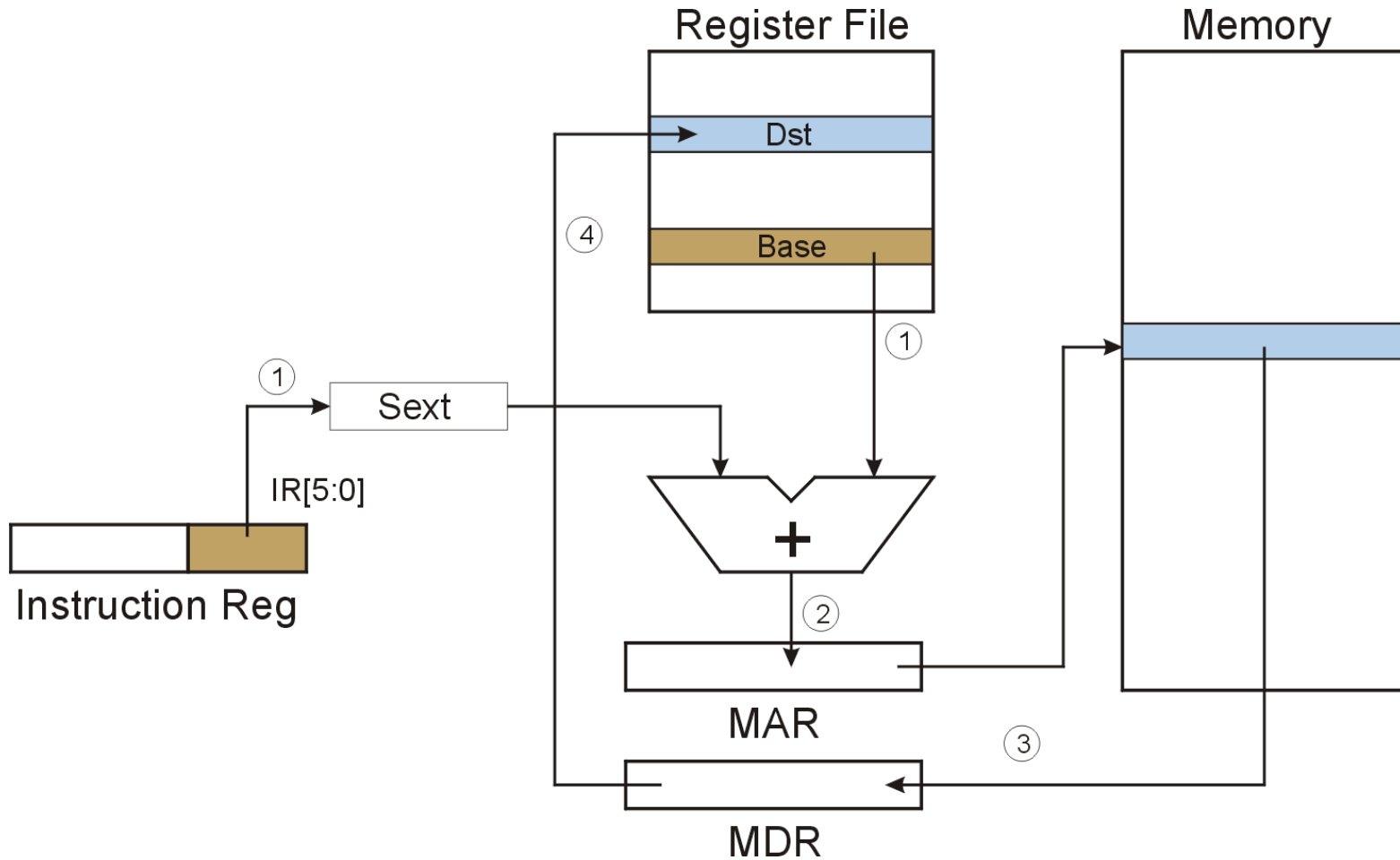
### Solution #2:

- Use a register to generate a full 16-bit address.

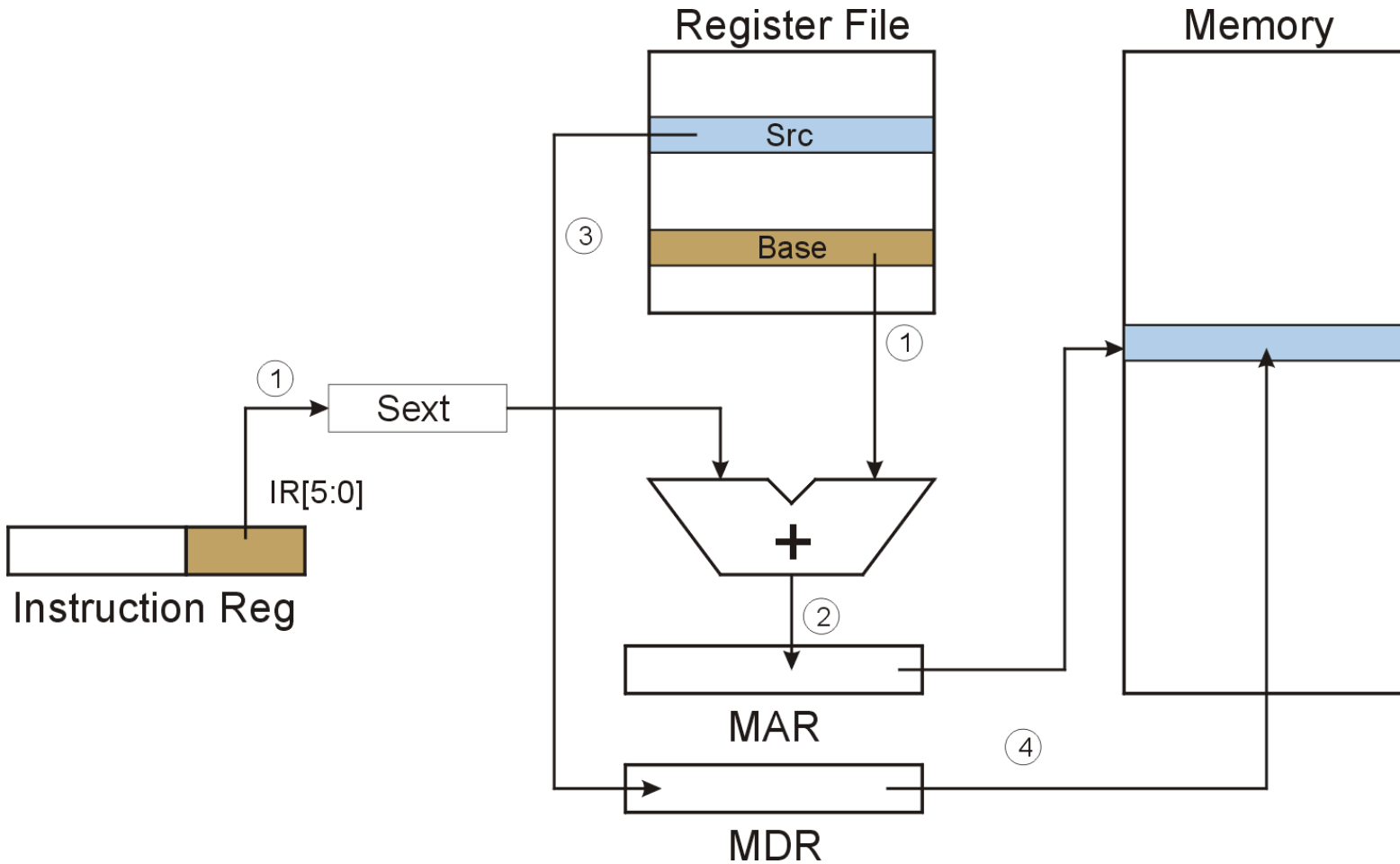
4 bits for opcode, 3 for src/dest register,  
3 bits for **base** register -- remaining 6 bits are used  
as a **signed offset**.

- Offset is *sign-extended* before adding to base register.

# LDR (Base+Offset)



# STR (Base+Offset)

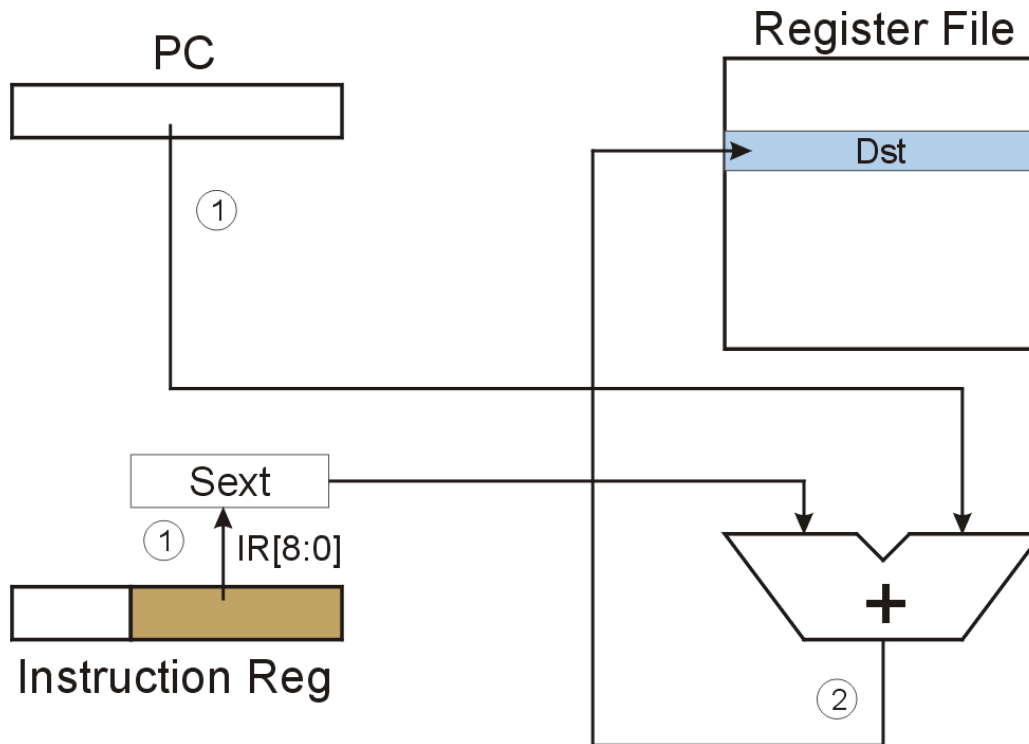


## Load Effective Address

Computes address like PC-relative (PC plus signed offset) and **stores the result into a register.**

**Note:** The address is stored in the register, not the contents of the memory location.

# LEA (Immediate)



# Example

| Address | Instruction                    | Comments   |
|---------|--------------------------------|--|
| x30F6   | <u>0 0 1 1 1 1 1 1 1 1 0 1</u> | $R1 \leftarrow PC - 3 = x30F4$   |
| x30F7   | <u>0 1 0 0 0 1 1 0 1 1 1 0</u> | $R2 \leftarrow R1 + 14 = x3102$  |
| x30F8   | <u>0 1 0 1 1 1 1 1 0 1 1</u>   | $M[PC - 5] \leftarrow R2$<br>$M[x30F4] \leftarrow x3102$                     |
| x30F9   | <u>0 1 0 0 1 0 1 0 0 0 0 0</u> | $R2 \leftarrow 0$  |
| x30FA   | <u>0 1 0 0 1 0 1 0 0 1 0 1</u> | $R2 \leftarrow R2 + 5 = 5$   |
| x30FB   | <u>0 1 0 0 0 1 0 0 1 1 1 0</u> | $M[R1+14] \leftarrow R2$<br>$M[x3102] \leftarrow 5$                          |
| x30FC   | <u>0 1 1 1 1 1 1 1 0 1 1 1</u> | $R3 \leftarrow M[M[x30F4]]$<br>$R3 \leftarrow M[x3102]$<br>$R3 \leftarrow 5$ |

opcode

# Control Instructions

Used to alter the sequence of instructions  
(by changing the Program Counter)

## Conditional Branch

- branch is *taken* if a specified condition is true
  - signed offset is added to PC to yield new PC
- else, the branch is *not taken*
  - PC is not changed, points to the next sequential instruction

## Unconditional Branch (or Jump)

- always changes the PC

## TRAP

- changes PC to the address of an OS “service routine”
- routine will return control to the next instruction (after TRAP)



## Condition Codes

LC-3 has three **condition code** registers:

**N** -- negative

**Z** -- zero

**P** -- positive (greater than zero)

Set by any instruction that writes a value to a register  
(ADD, AND, NOT, LD, LDR, LDI, LEA)

Exactly one will be set at all times

- Based on the last instruction that altered a register

## Branch Instruction

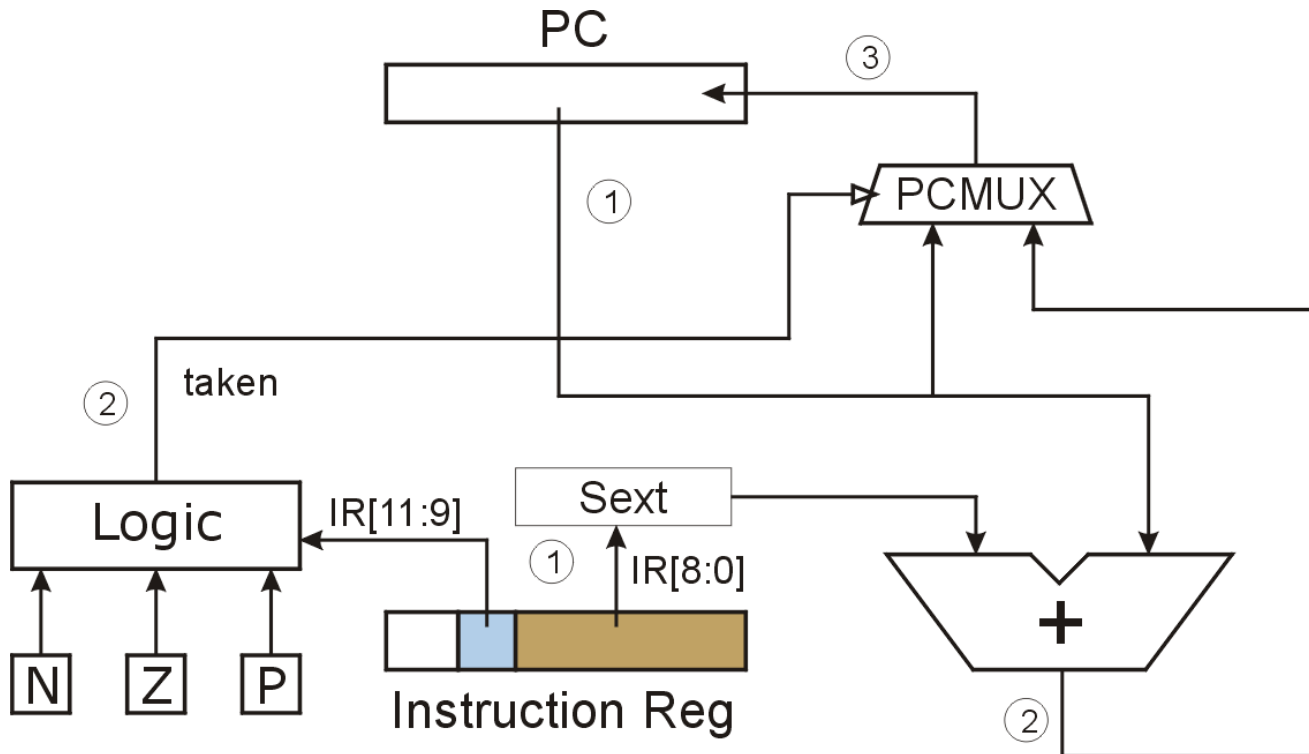
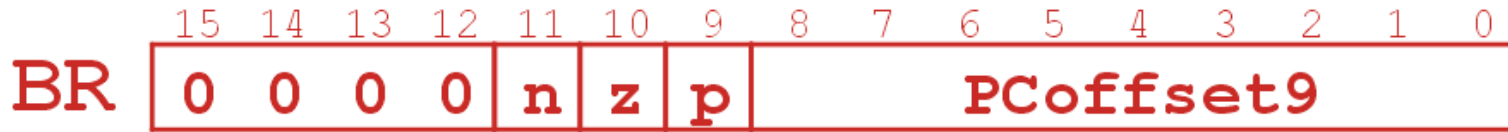
**Branch specifies one or more condition codes.**

**If the set bit is specified, the branch is taken.**

- **PC-relative addressing:**  
**target address** is made by adding signed offset (IR[8:0]) to current PC.
- **Note: PC has already been incremented by FETCH stage.**
- **Note: Target must be within 256 words of BR instruction.**

**If the branch is not taken,  
the next sequential instruction is executed.**

# BR (PC-Relative)

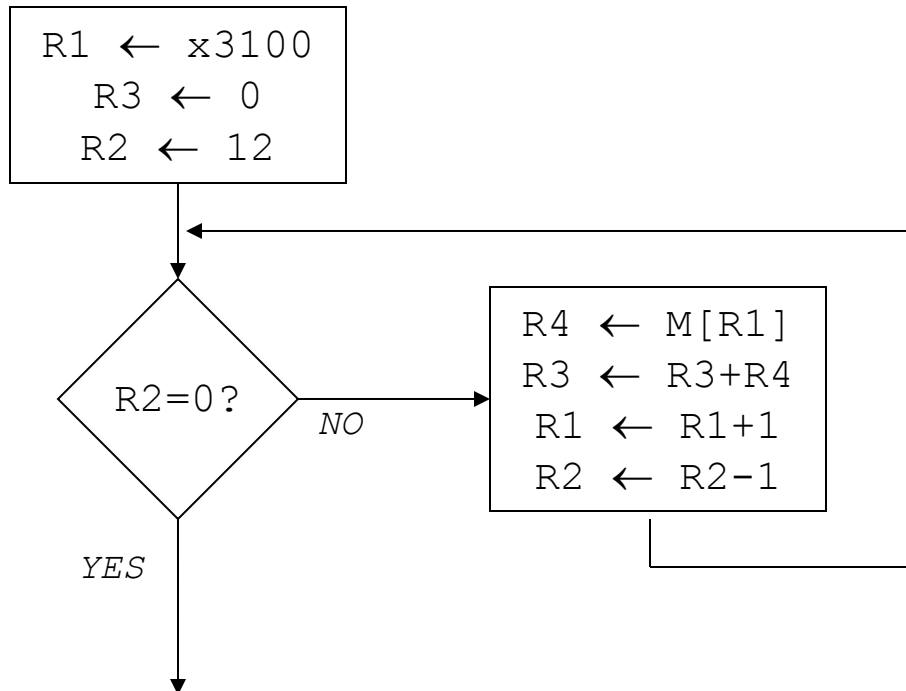


What happens if bits [11:9] are all zero? All one?

# Using Branch Instructions

**Compute sum of 12 integers.**

**Numbers start at location x3100. Program starts at location x3000.**



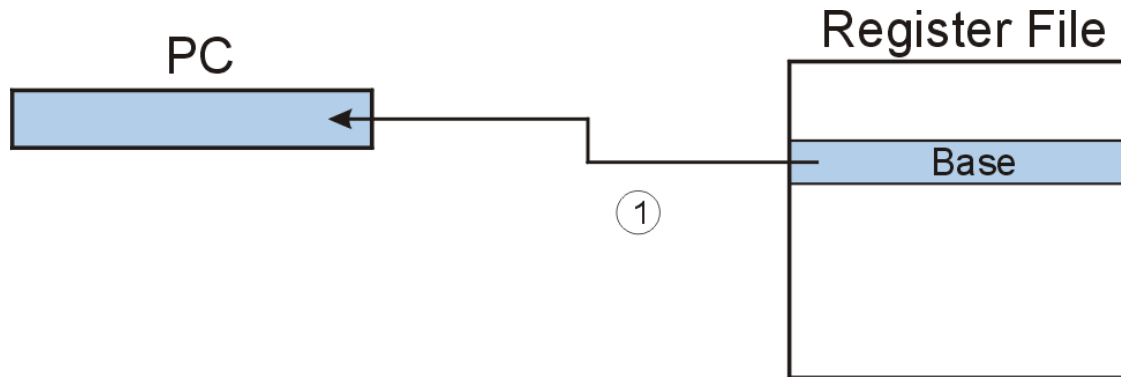
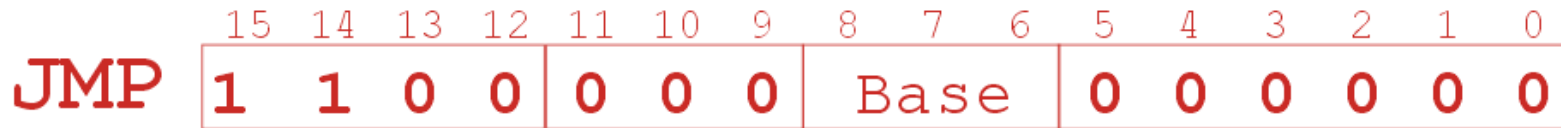
# Sample Program

| Address | Instruction |   |   |   |          |          |          |          |          |          |          |          | Comments |          |          |          |                                |                          |
|---------|-------------|---|---|---|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|--------------------------------|--------------------------|
| x3000   | 1           | 1 | 1 | 0 | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <i>R1 ← x3100 (PC+0xFF)</i>    |                          |
| x3001   | 0           | 1 | 0 | 1 | <u>0</u> | <u>1</u> | <u>1</u> | <u>0</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <i>R3 ← 0</i>                  |                          |
| x3002   | 0           | 1 | 0 | 1 | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <i>R2 ← 0</i>                  |                          |
| x3003   | 0           | 0 | 0 | 1 | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>1</u> | <u>1</u> | <u>0</u> | <u>0</u> | <i>R2 ← 12</i>                 |                          |
| x3004   | 0           | 0 | 0 | 0 | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>1</u> | <i>If Z, goto x300A (PC+5)</i> |                          |
| x3005   | 0           | 1 | 1 | 0 | <u>1</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <i>Load next value to R4</i>   |                          |
| x3006   | 0           | 0 | 0 | 1 | <u>0</u> | <u>1</u> | <u>1</u> | <u>0</u> | <u>1</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <i>Add to R3</i>               |                          |
| x3007   | 0           | 0 | 0 | 1 | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>1</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>1</u> | <i>Increment R1 (pointer)</i>  |                          |
| x3008   | 0           | 0 | 0 | 1 | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <i>Decrement R2 (counter)</i>  |                          |
| x3009   | 0           | 0 | 0 | 0 | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>0</u> | <u>1</u> | <u>0</u>                       | <i>Goto x3004 (PC-6)</i> |

# JMP (Register)

Jump is an unconditional branch -- always taken.

- Target address is the contents of a register.
- Allows any target address.



# TRAP



Calls a **service routine**, identified by 8-bit “trap vector.”

| <i>vector</i> | <i>routine</i>                      |
|---------------|-------------------------------------|
| x23           | input a character from the keyboard |
| x21           | output a character to the monitor   |
| x25           | halt the program                    |

When routine is done,  
**PC is set to the instruction following TRAP.**  
 (We’ll talk about how this works later.)

## Another Example

### Count the occurrences of a character in a file

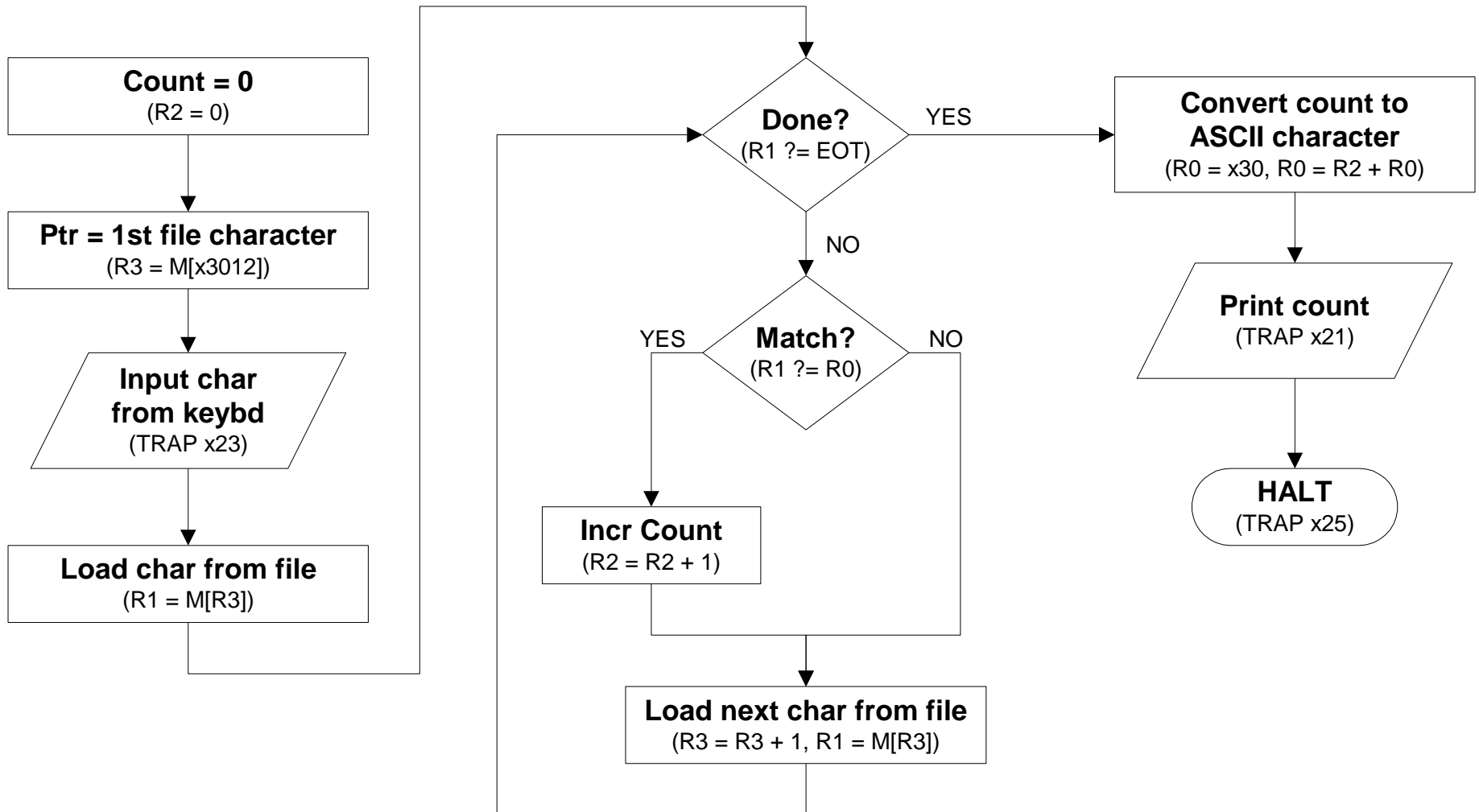
- **Program begins at location x3000**
- **Read character from keyboard**
- **Load each character from a “file”**
  - File is a sequence of memory locations
  - Starting address of file is stored in the memory location immediately after the program
- **If file character equals input character, increment counter**
- **End of file is indicated by a special ASCII value: EOT (x04)**
- **At the end, print the number of characters and halt**  
(assume there will be less than 10 occurrences of the character)

**A special character used to indicate the end of a sequence is often called a **sentinel**.**

- Useful when you don't know ahead of time how many times to execute a loop.



# Flow Chart



# Program (1 of 2)

| <b>Address</b> | <b>Instruction</b> |   |   |   |          |          |          |          |          |          |          |          | <b>Comments</b> |          |          |          |                               |
|----------------|--------------------|---|---|---|----------|----------|----------|----------|----------|----------|----------|----------|-----------------|----------|----------|----------|-------------------------------|
| <b>x3000</b>   | 0                  | 1 | 0 | 1 | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u>        | <u>0</u> | <u>0</u> | <u>0</u> | <i>R2 ← 0 (counter)</i>       |
| <b>x3001</b>   | 0                  | 0 | 1 | 0 | <u>0</u> | <u>1</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>1</u>        | <u>0</u> | <u>0</u> | <u>0</u> | <i>R3 ← M[x3012] (ptr)</i>    |
| <b>x3002</b>   | 1                  | 1 | 1 | 1 | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u>        | <u>0</u> | <u>0</u> | <u>1</u> | <i>Input to R0 (TRAP x23)</i> |
| <b>x3003</b>   | 0                  | 1 | 1 | 0 | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>1</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>0</u>        | <u>0</u> | <u>0</u> | <u>0</u> | <i>R1 ← M[R3]</i>             |
| <b>x3004</b>   | 0                  | 0 | 0 | 1 | <u>1</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u>        | <u>1</u> | <u>0</u> | <u>0</u> | <i>R4 ← R1 - 4 (EOT)</i>      |
| <b>x3005</b>   | 0                  | 0 | 0 | 0 | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u>        | <u>1</u> | <u>0</u> | <u>0</u> | <i>If Z, goto x300E</i>       |
| <b>x3006</b>   | 1                  | 0 | 0 | 1 | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>1</u> | <u>1</u> | <u>1</u> | <u>1</u>        | <u>1</u> | <u>1</u> | <u>1</u> | <i>R1 ← NOT R1</i>            |
| <b>x3007</b>   | 0                  | 0 | 0 | 1 | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>1</u> | <u>1</u> | <u>0</u> | <u>0</u>        | <u>0</u> | <u>0</u> | <u>1</u> | <i>R1 ← R1 + 1</i>            |
| <b>x3008</b>   | 0                  | 0 | 0 | 1 | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>0</u>        | <u>0</u> | <u>0</u> | <u>0</u> | <i>R1 ← R1 + R0</i>           |
| <b>x3009</b>   | 0                  | 0 | 0 | 0 | <u>1</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u> | <u>0</u>        | <u>0</u> | <u>0</u> | <u>1</u> | <i>If N or P, goto x300B</i>  |

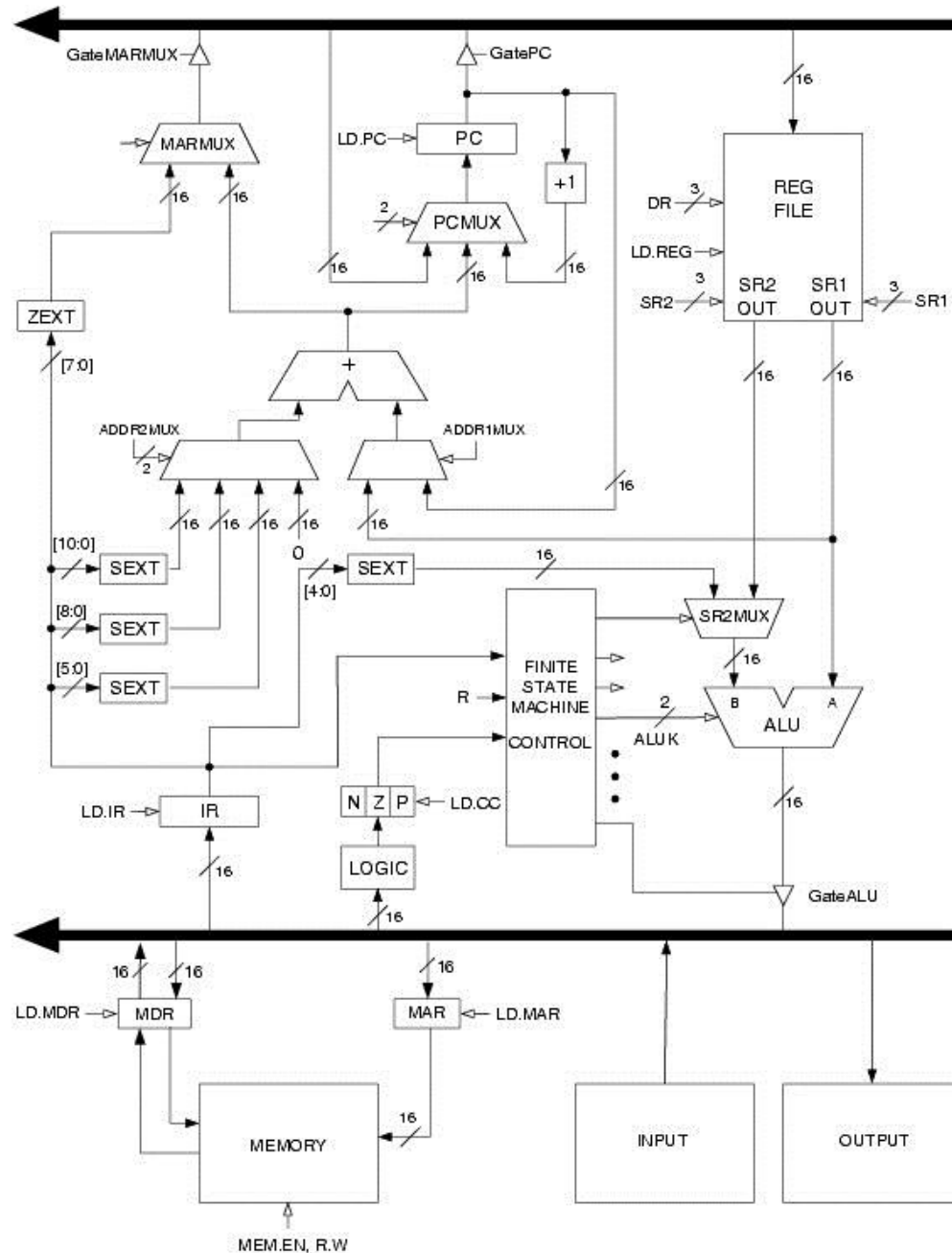
## Program (2 of 2)

| <i>Address</i> | <i>Instruction</i>                              | <i>Comments</i>            |
|----------------|---|----------------------------|
| x300A          | 0 0 0 1 <u>0 1 0 0</u> <u>1 0 1 0</u> 0 0 0 0 1 | $R2 \leftarrow R2 + 1$     |
| x300B          | 0 0 0 1 <u>0 1 1 0</u> <u>1 1 1 0</u> 0 0 0 0 1 | $R3 \leftarrow R3 + 1$     |
| x300C          | 0 1 1 0 <u>0 0 1 0</u> <u>1 1 0 0</u> 0 0 0 0 0 | $R1 \leftarrow M[R3]$      |
| x300D          | 0 0 0 0 <u>1 1 1 1</u> <u>1 1 1 1</u> 0 1 1 0   | <i>Goto x3004</i>          |
| x300E          | 0 0 1 0 <u>0 0 0 0</u> <u>0 0 0 0</u> 0 0 1 0 0 | $R0 \leftarrow M[x3013]$   |
| x300F          | 0 0 0 1 <u>0 0 0 0</u> <u>0 0 0 0</u> 0 0 0 1 0 | $R0 \leftarrow R0 + R2$    |
| x3010          | 1 1 1 1 0 0 0 0 <u>0 0 1 0</u> 0 0 0 1          | <i>Print R0 (TRAP x21)</i> |
| x3011          | 1 1 1 1 0 0 0 0 <u>0 0 1 0</u> 0 1 0 1          | <i>HALT (TRAP x25)</i>     |
| X3012          | Starting Address of File                        |                            |
| x3013          | 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0                 | <i>ASCII x30 ('0')</i>     |

# LC-3 Data Path Revisited

**Filled arrow**  
= info to be processed.

**Unfilled arrow**  
= control signal.



# Data Path Components

## Global bus

- **special set of wires that carry a 16-bit signal to many components**
- **inputs to the bus are “tri-state devices,” that only place a signal on the bus when they are enabled**
- **only one (16-bit) signal should be enabled at any time**
  - **control unit decides which signal “drives” the bus**
- **any number of components can read the bus**
  - **register only captures bus data if it is write-enabled by the control unit**

## Memory

- **Control and data registers for memory and I/O devices**
- **memory: MAR, MDR (also control signal for read/write)**

# Data Path Components

## ALU

- **Accepts inputs from register file and from sign-extended bits from IR (immediate field).**
- **Output goes to bus.**
  - **used by condition code logic, register file, memory**

## Register File

- **Two read addresses (SR1, SR2), one write address (DR)**
- **Input from bus**
  - **result of ALU operation or memory read**
- **Two 16-bit outputs**
  - **used by ALU, PC, memory address**
  - **data for store instructions passes through ALU**

# Data Path Components

## PC and PCMUX

- Three inputs to PC, controlled by PCMUX
  1. PC+1 – FETCH stage
  2. Address adder – BR, JMP
  3. bus – TRAP (discussed later)

## MAR and MARMUX

- Two inputs to MAR, controlled by MARMUX
  1. Address adder – LD/ST, LDR/STR
  2. Zero-extended IR[7:0] -- TRAP (discussed later)

# Data Path Components

## Condition Code Logic

- Looks at value on bus and generates N, Z, P signals
- Registers set only when control unit enables them (LD.CC)
  - only certain instructions set the codes (ADD, AND, NOT, LD, LDI, LDR, LEA)

## Control Unit – Finite State Machine

- On each machine cycle, changes control signals for next phase of instruction processing
  - who drives the bus? (GatePC, GateALU, ...)
  - which registers are write enabled? (LD.IR, LD.REG, ...)
  - which operation should ALU perform? (ALUK)
  - ...
- Logic includes decoder for opcode, etc.