McGraw Hill Education

Because learning changes everything.®

# From Bits and Gates to C and Beyond

## The LC-3

Chapter 5

# LC-3 Instruction Set Architecture

The Instruction Set Architecture (ISA) specifies everything about the computer that is visible to the programmer.

**Memory Organization**

- $2^{16}$ memory locations, with 16 bits in each location.

**Registers**

- 8 general purpose registers, each 16 bits, called R0 - R7.
- Used as source and destination locations for instruction operands.

**Instructions**

- 15 opcodes, each specified by a 4-bit code.
- Only one data type: 16-bit 2's complement integer.
- Addressing modes: various encodings to specify location of operands.

**Condition Codes**

- Three 1-bit registers (N, Z, P) that are set whenever a value is written to a register. These are used for conditional branch instructions. N = negative, Z = zero, P = positive.

# LC-3 Instruction Set

Bits: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| Instruction | | | | | | |
|---|---|---|---|---|---|---|
| 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 | | | | | |

Access the text alternative for slide images.

# LC-3 Instruction Set: Notation

Each opcode has a 4-bit encoding and one or more mnemonic symbols.
For example, the ADD instruction has opcode 0001, but we typically just say "ADD". The mnemonic symbols are used also used when we program in assembly language (Chapter 7).

Some opcodes appear twice, because they have different addressing modes -- different ways to specify the operands. (Sometimes, the different addressing modes use different mnemonic symbols.)

Some abbreviations are defined below, and will be explained when we discuss the relevant instructions:

- SR = source register
- DR = destination register
- BaseR = base register
- immN = immediate value, N bits
- offsetN, PCoffsetN = offset to be added to the BaseR or PC, N bits
- trapvector8 = index into the TRAP table, 8 bits

# Operate Instructions

ADD

Add two integers and write the result to a register.

Set condition codes.

AND

Perform a bitwise AND operation on two integers, and write the result to a register.

Set condition codes.

NOT

Perform a bitwise NOT on an integer, and write the result to a register.

Set condition codes.

LEA

Add an offset to the PC and write the result to a register.

# Addressing Modes

Register

- First operand is always specified by a register.
- Second operand is a register if IR[5] = 0.

Immediate

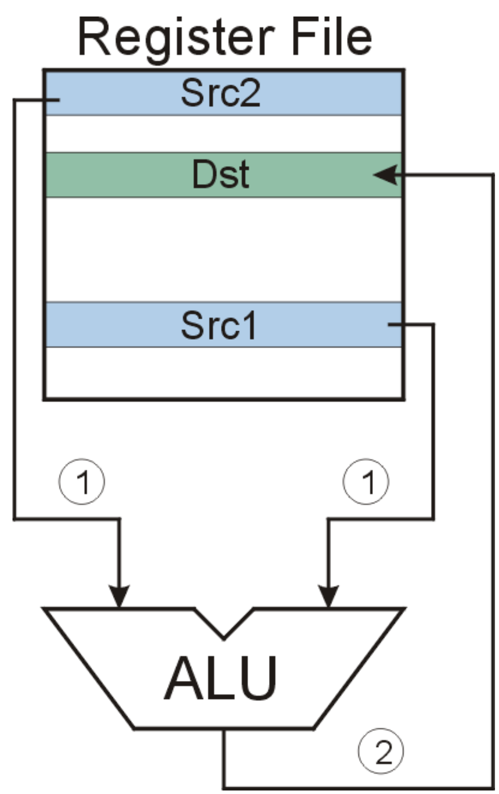
- If IR[5] = 1, the second operand is specified in the instruction. IR[4:0] is a 5-bit 2's complement integer, sign-extended to 16 bits.

# ADD/AND (Register mode)

*this zero means "register mode"* (bit 5)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD | 0 | 0 | 0 | 1 | Dst | | | Src1 | | | 0 | 0 | 0 | Src2 | | |
| AND | 0 | 1 | 0 | 1 | Dst | | | Src1 | | | 0 | 0 | 0 | Src2 | | |

Sets condition codes.




Access the text alternative for slide images.

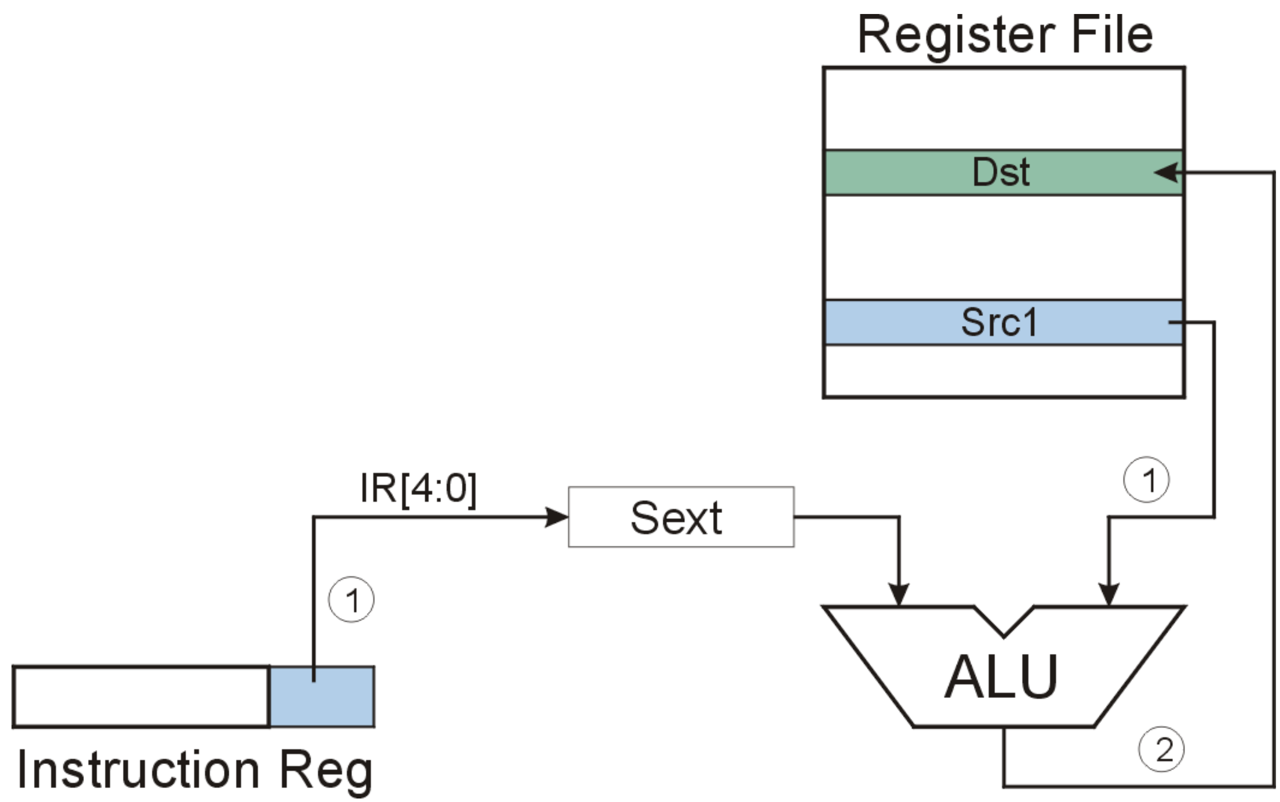
# ADD/AND (Immediate mode)

this one means "immediate mode"

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD | 0 | 0 | 0 | 1 | Dst | | | Src1 | | | 1 | Imm5 | | | | |

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AND | 0 | 1 | 0 | 1 | Dst | | | Src1 | | | 1 | Imm5 | | | | |

Sets condition codes.

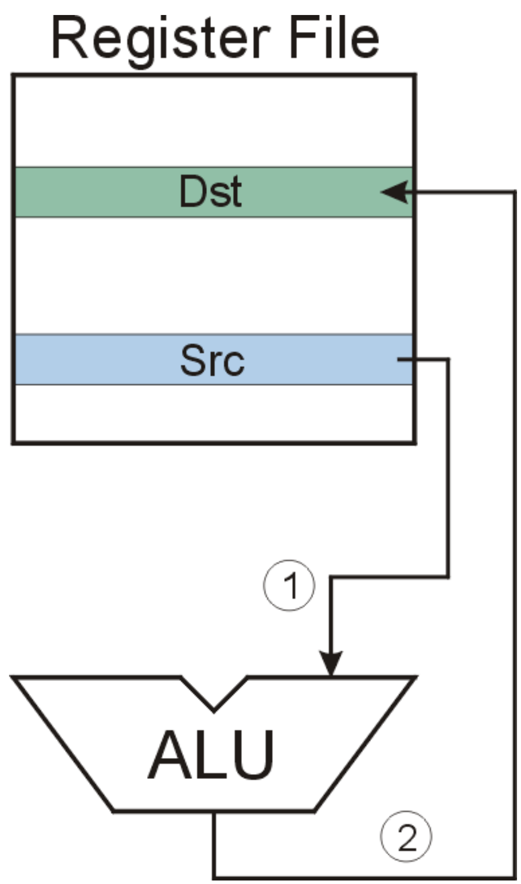



Access the text alternative for slide images.

# NOT (Register mode)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NOT | 1 | 0 | 0 | 1 | Dst | | | Src | | | 1 | 1 | 1 | 1 | 1 | 1 |

Sets condition codes.



# Only ADD, AND, and NOT?

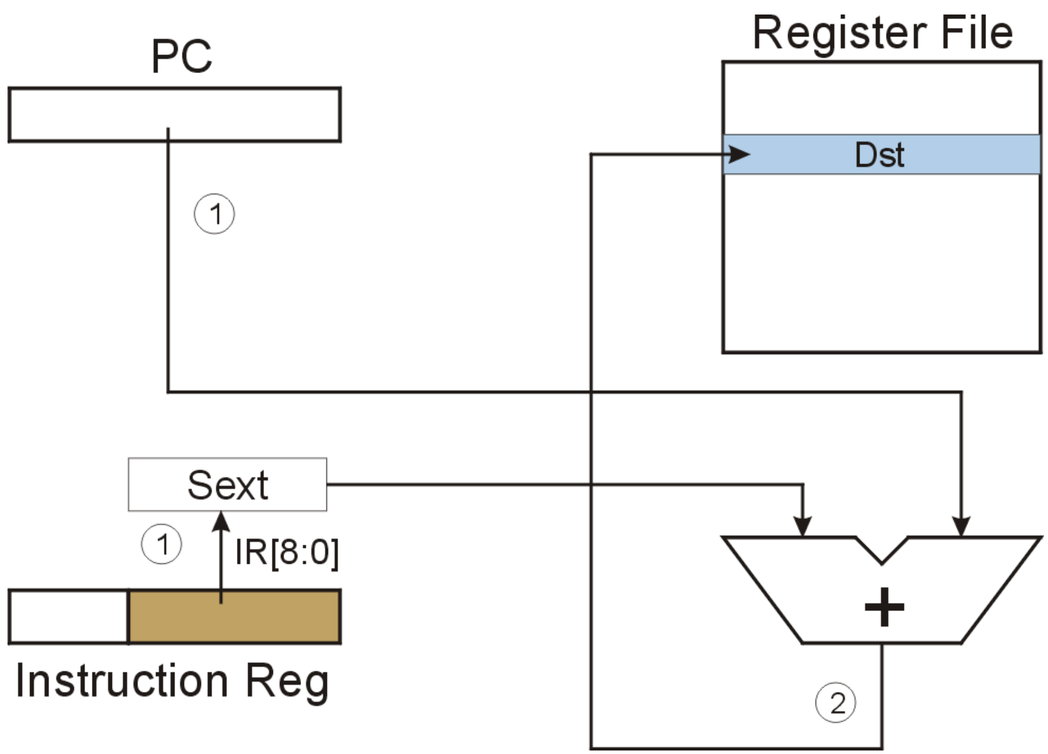
It might seem very limiting to only provide these three operations, but anything else can be built from these.

- Negate -- Flip bits with NOT, then ADD +1.
- Subtract -- Negate and ADD.
- Multiply -- Repeated additions.
- OR -- Use DeMorgan's Law (see Chapter 2).
- Clear -- AND with 0 to set register to zero.
- Copy -- ADD 0 to copy from one register (src) to another (dst).
- Others:  XOR, divide, shift left, ...

# LEA

Computes an address by adding sign-extended IR[8:0] to PC, and write the result to a register. (Does not set condition codes.)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **LEA** | 1 | 1 | 1 | 0 | Dst | | | PCoffset9 | | | | | | | | |




Access the text alternative for slide images.

# Data Movement Instructions: Load and Store

Used to move data between memory and registers.

LOAD = from memory to register (and set condition codes)

STORE = from register to memory

Three variants of each, with different addressing modes:

| | |
|---|---|
| PC-Relative | Compute address by adding an offset IR[8:0] to the PC. |
| Indirect | Load an address from a memory location. |
| Base+Offset | Compute address by adding an offset IR[5:0] to a register. |

# 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.

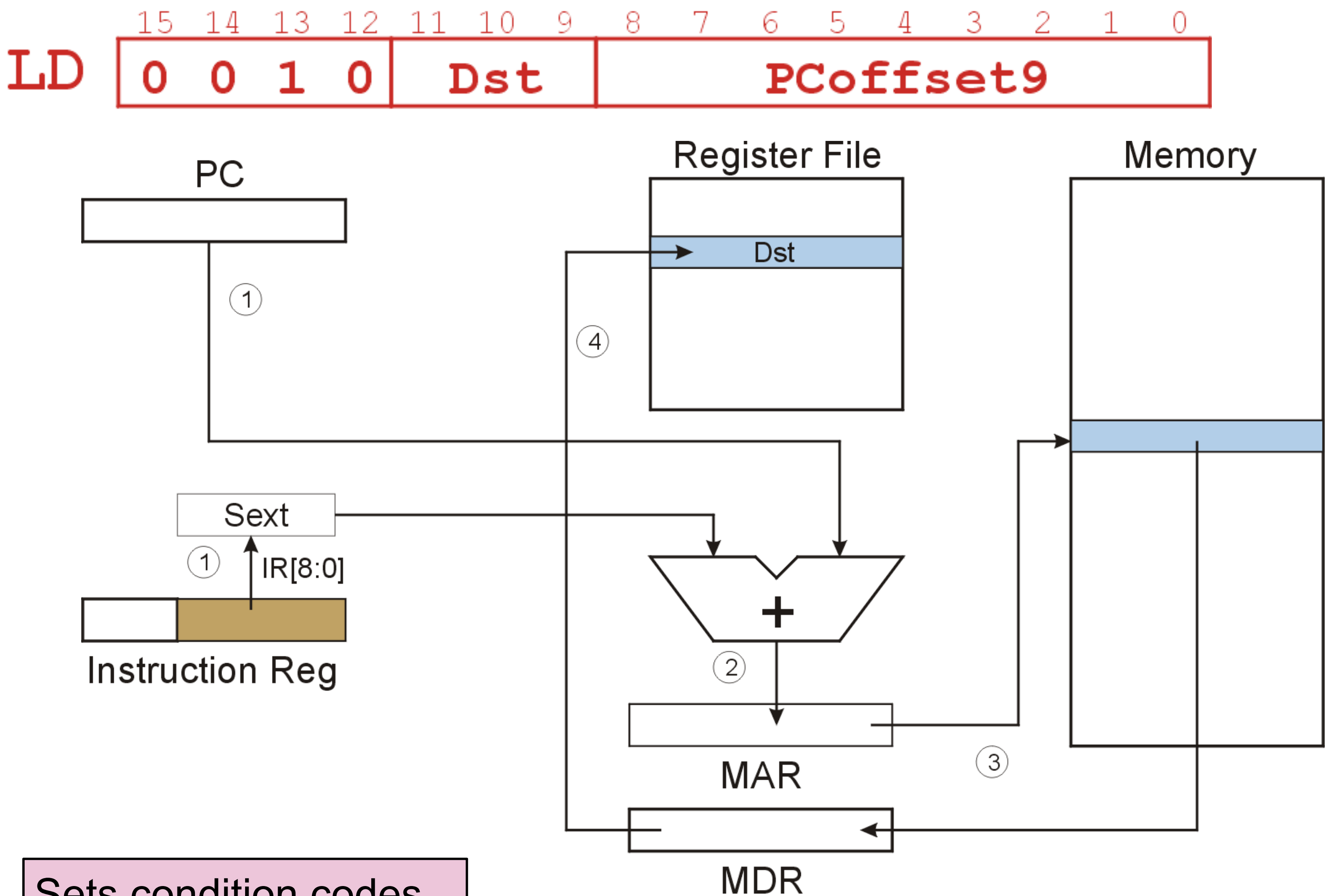
Solution: Use the 9 bits as a *signed offset* from the current PC.

Can form any address X, such that: PC − 256 ≤ $X$ ≤ PC + 255

Remember that PC is incremented as part of the FETCH phase;

This is done before the EVALUATE ADDRESS stage.

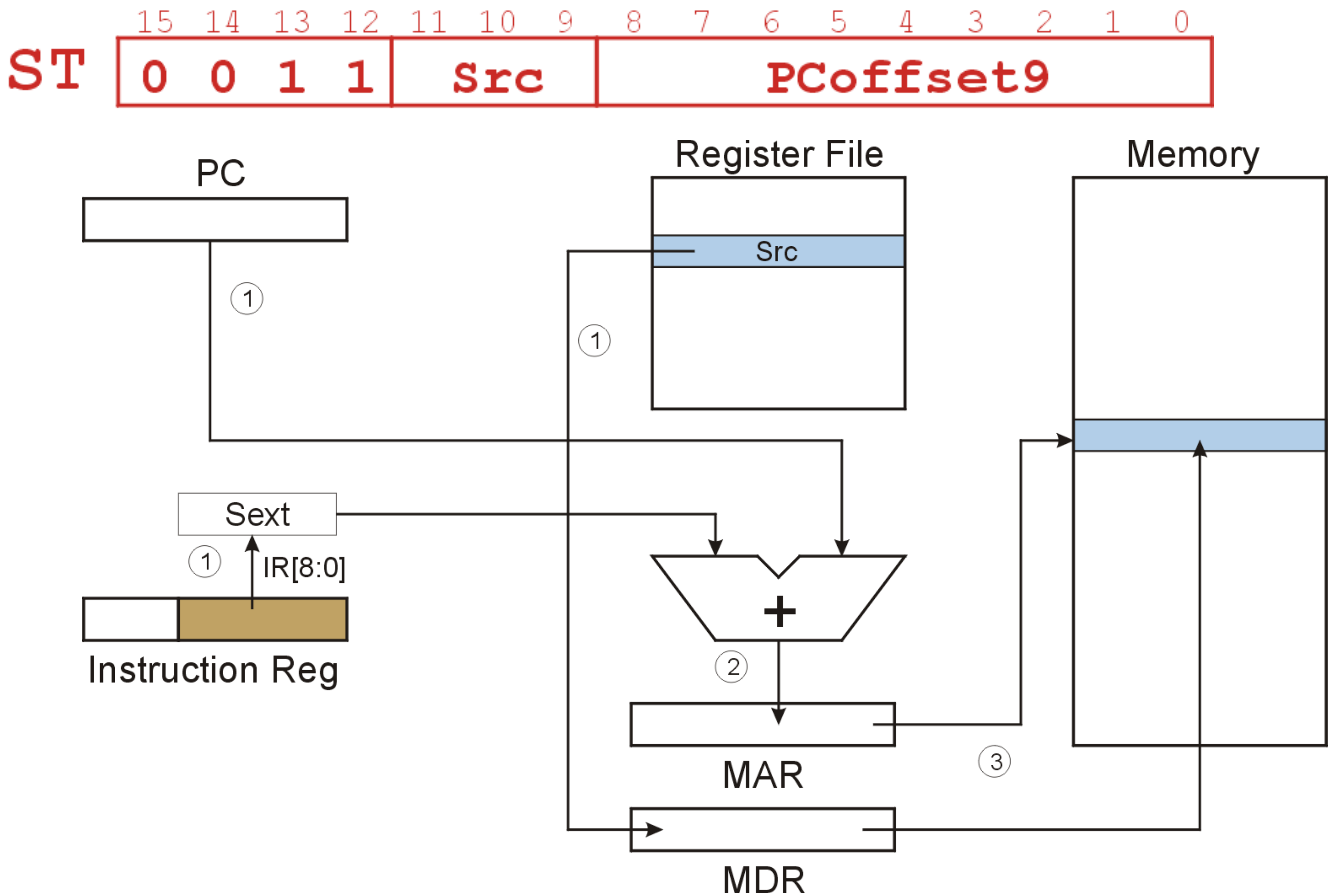
# LD (PC-Relative mode)




Sets condition codes.

Access the text alternative for slide images.

# ST (PC-Relative mode)




Access the text alternative for slide images.

# 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.

(1) Compute PC + IR[8:0] (just like PC-Relative), put in MAR.

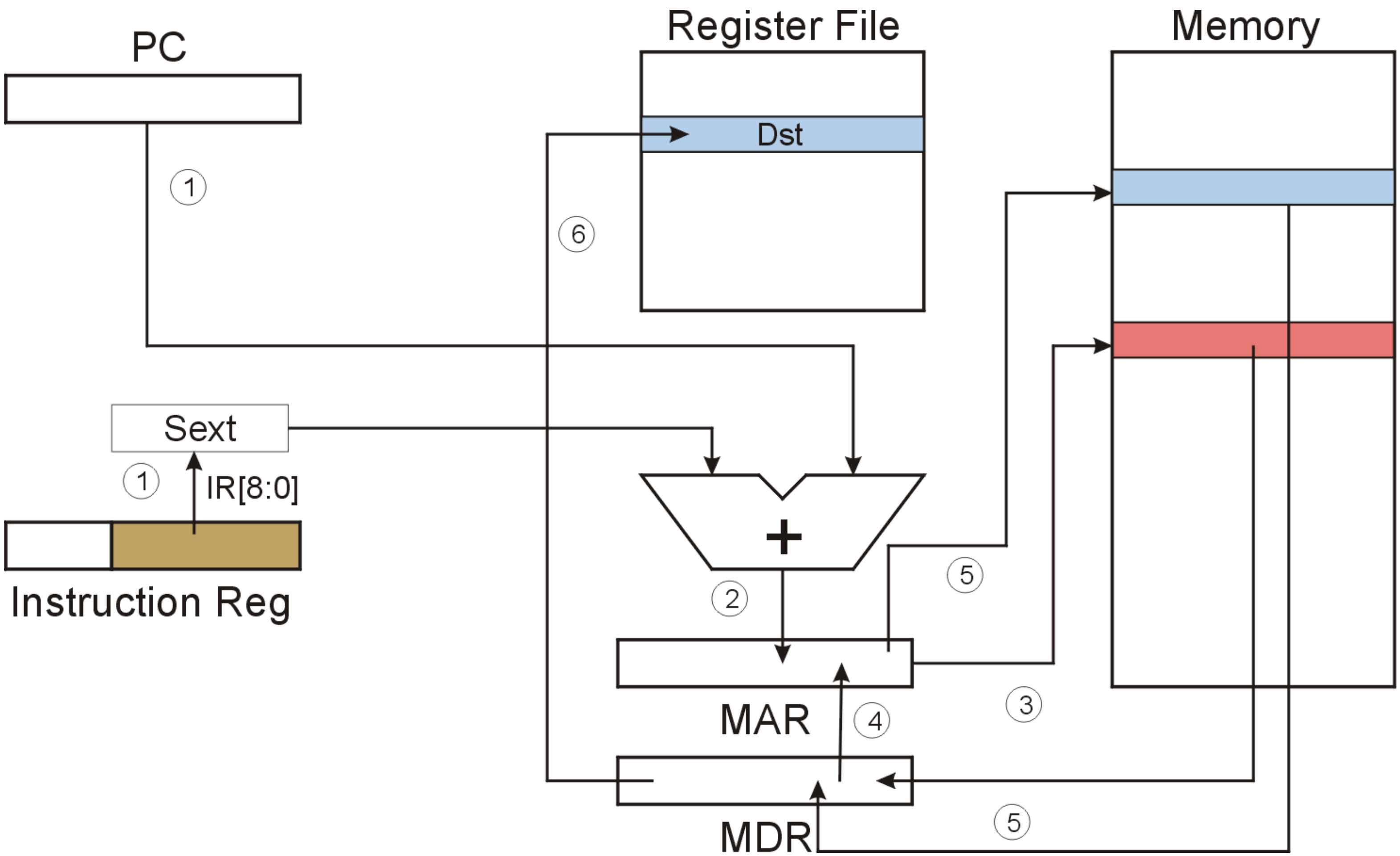
(2) Load from that address into MDR.

(3) Move data from MDR into MAR.

(4) Perform load/store.

# LDI (Indirect mode)

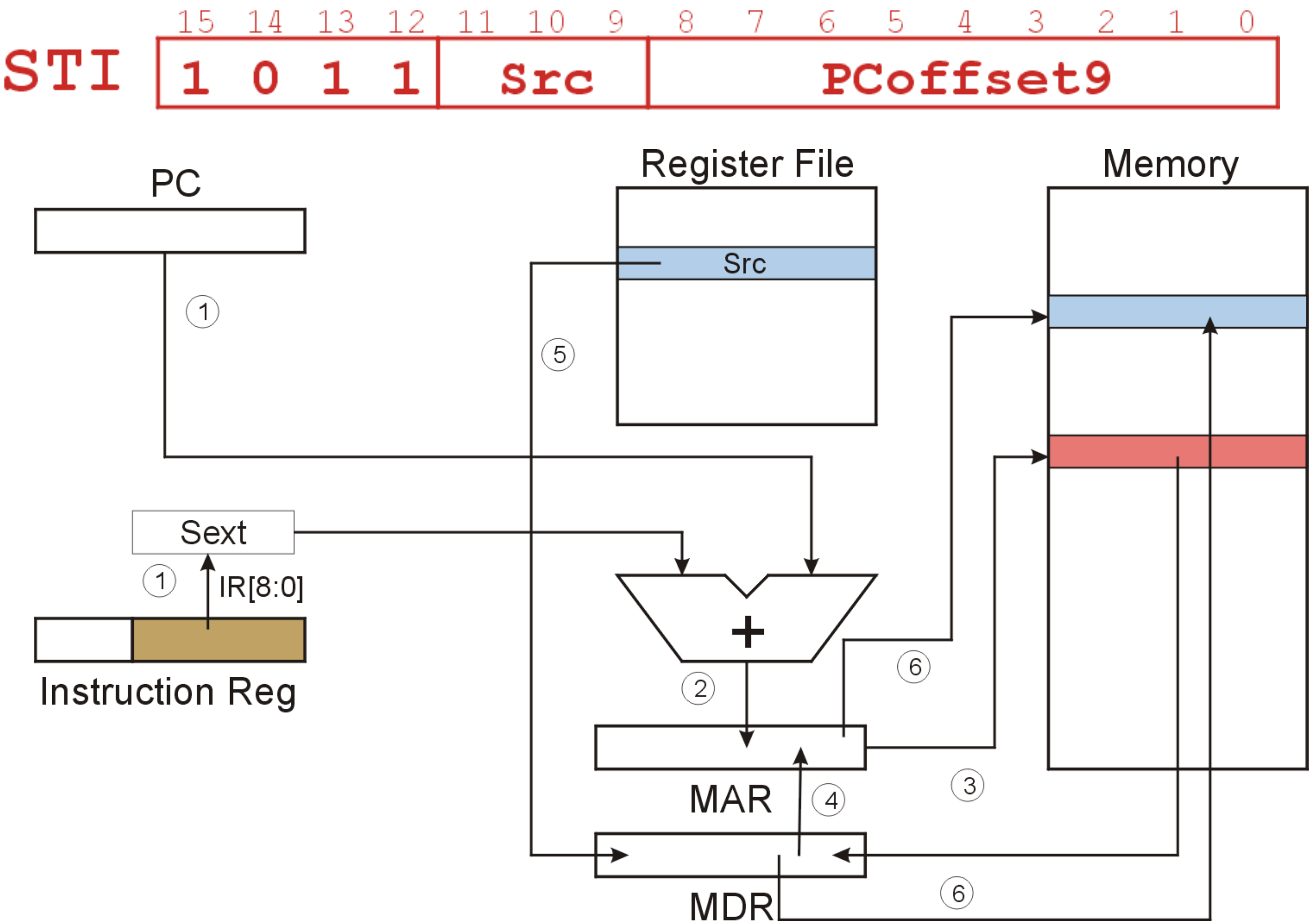
| | 15 | 14 | 13 | 12 | 11 10 9 | 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| LDI | 1 | 0 | 1 | 0 | Dst | PCoffset9 |




Sets condition codes.

Access the text alternative for slide images.

# STI (Indirect mode)




Access the text alternative for slide images.

# Base + Offset Addressing Mode

With PC-relative mode, can only address data within 256 words of the instruction. What about the rest of memory?

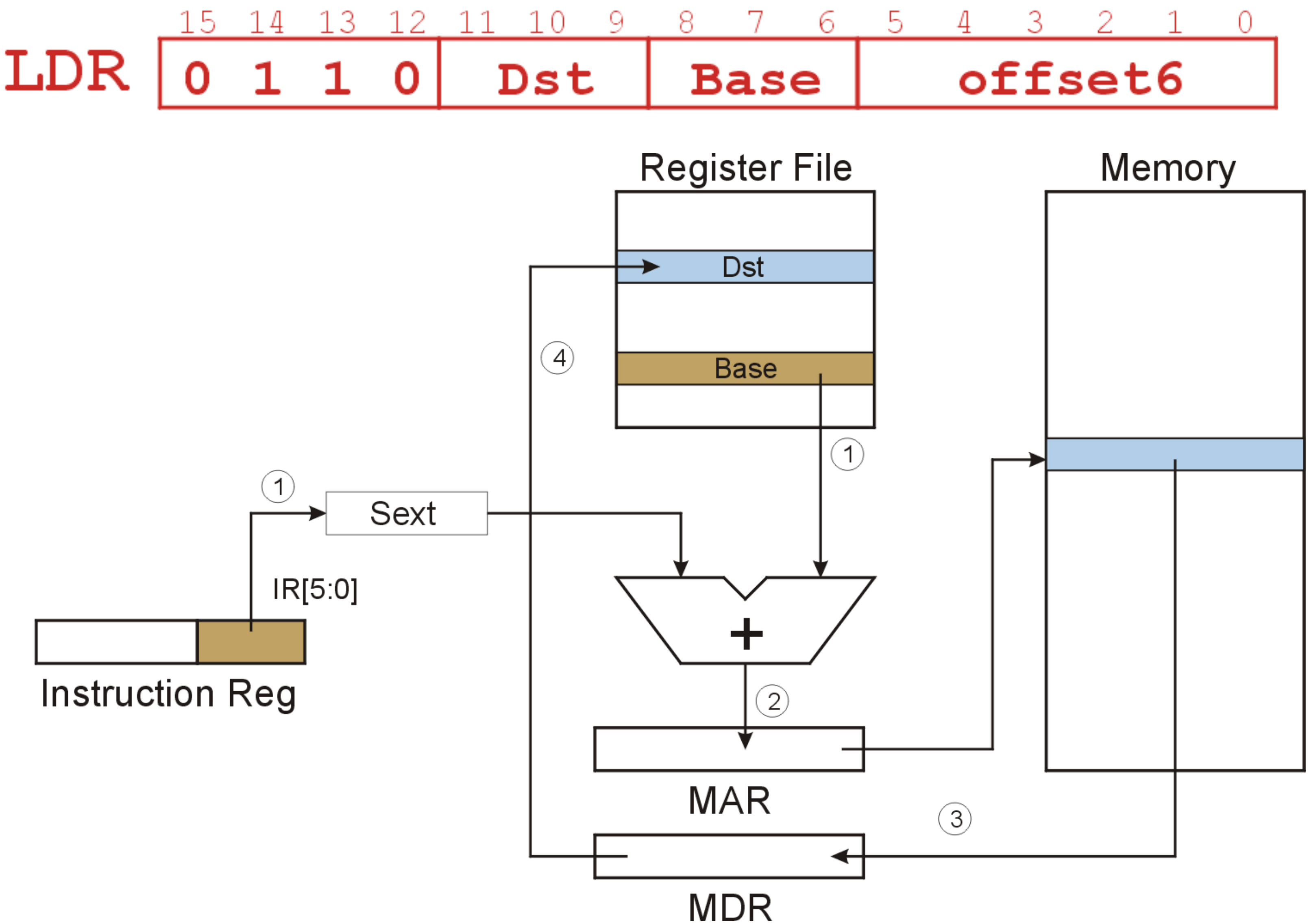
Solution #2: Use a register to generate a full 16-bit address.

4 bits for opcode, 3 for src/dest register,
3 bits for *base* register -- the one to be used as an address.

Remaining 6 bits are used as a *signed offset*.

- Offset is *sign-extended* before adding to base register.
  Offset is often zero, but a non-zero offset can be useful at times.

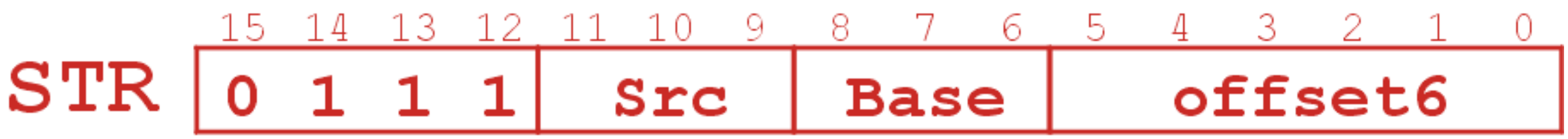
# LDR (Base + Offset mode)

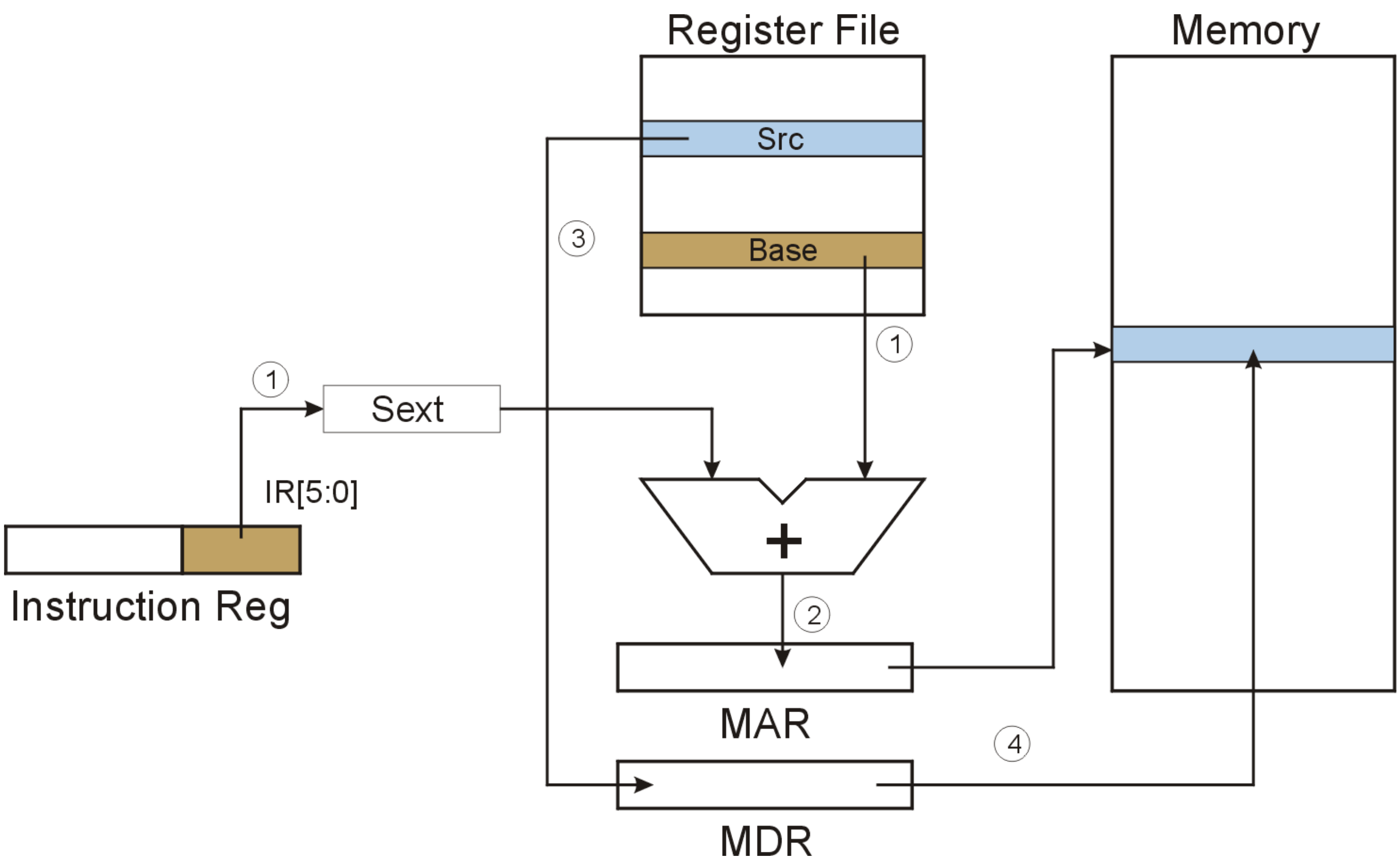



Sets condition codes.

Access the text alternative for slide images.

# STR (Base + Offset mode)




Access the text alternative for slide images.

# Example using Load/Store Addressing Modes

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

*opcode*

Access the text alternative for slide images.

# Control Instructions

Changes the PC to redirect program execution.

Conditional Branch: BR

Unconditional Branch: JMP

Subroutines and System Calls: JSR, JSRR, TRAP, RET, RTI
(Discussed in Chapter 8 and Chapter 9)

# Condition Codes

LC-3 has three condition code registers.

N = negative (less than zero)

Z = zero

P = positive (greater than zero)

Set by instructions that compute or load a value into a register.

ADD, AND, NOT, LD, LDI, LDR

Based on result of computation / load. Other instructions do not change the condition codes.

Exactly one of these will be set (1). Others will be zero.

# Conditional Branch (BR)

BR specifies two things:

- Condition --
  if true, branch is taken (PC changed to target address)
  if false, branch is not taken (PC not changed)
- Target address.

Condition specifies which condition code bits are relevant for this branch.
If any specified bit is set, branch is taken.

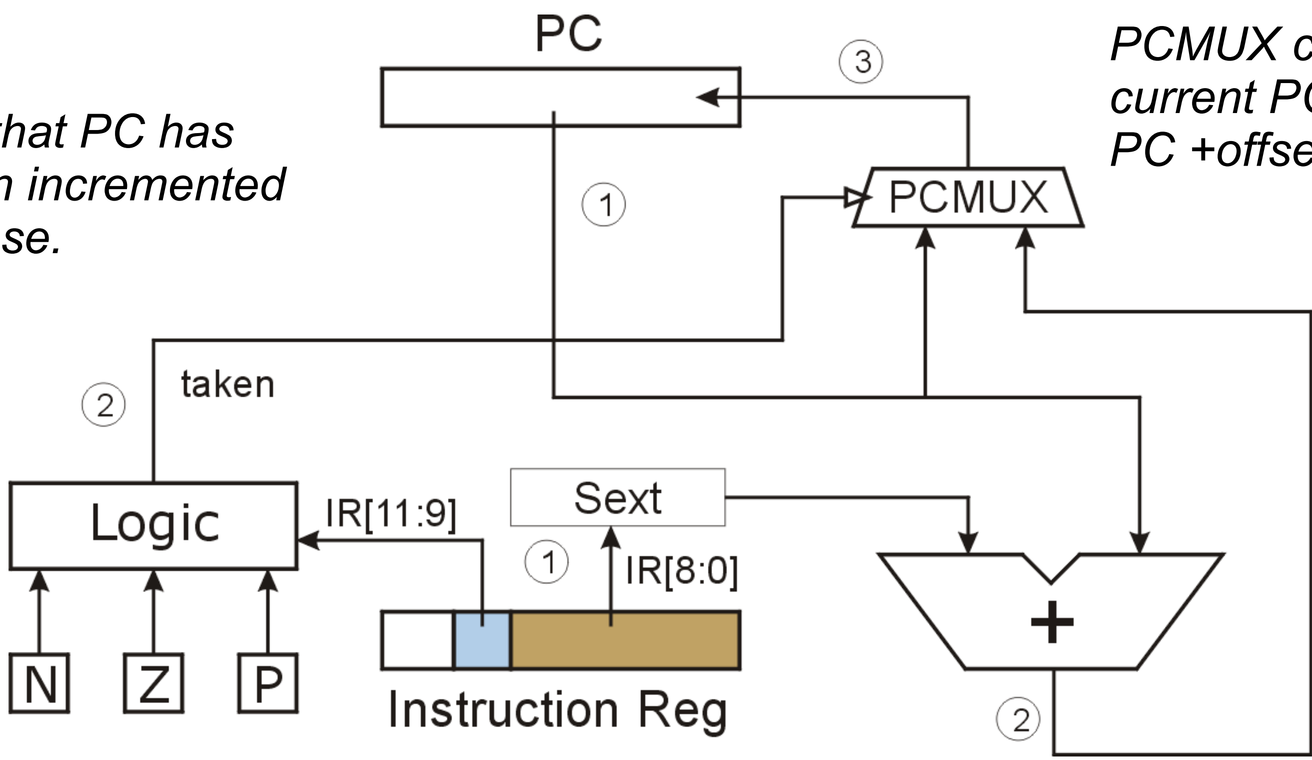
Example: BRn = branch if N bit is set
Example: BRnp = branch if N or P bit is set

Target is specified using PC-Relative mode.

# BR


Remember that PC has already been incremented in Fetch phase.

PCMUX chooses either the current PC if not taken, or PC +offset if taken.




*Logic: (IR[11] and N) or (IR[10] and Z) or (IR[9] and P)*

Access the text alternative for slide images.

# Example BR instruction

Suppose the following instruction is fetched from address x4027



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| BR | | | | n | z | p | x0D9 | | | | | | | | |

Condition: z
Branch will be taken if Z bit is set.

Target: x4028 + x00D9 = x4101

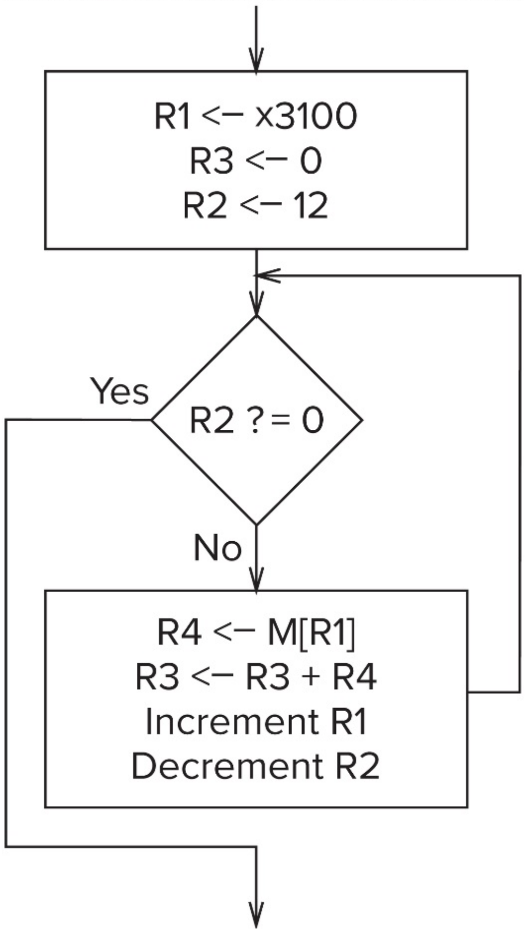
Current PC is x4028, because it was incremented when fetching.
If branch is taken PC will be x4101.
If branch is not taken PC will remain x4028.

Access the text alternative for slide images.

# Example: Loop with a Counter

Compute the sum of 12 integers stored in x3100 - x310B. Instructions begin at address x3000.








| Address | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x3000 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | R1<- 3100 |
| x3001 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | R3 <- 0 |
| x3002 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | R2 <- 0 |
| x3003 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | R2 <- 12 |
| x3004 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | BRz x300A |
| x3005 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | R4 <- M[R1] |
| x3006 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | R3 <- R3+R4 |
| x3007 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | R1 <- R1+1 |
| x3008 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | R2 <- R2-1 |
| x3009 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | BRnzp x3004 |

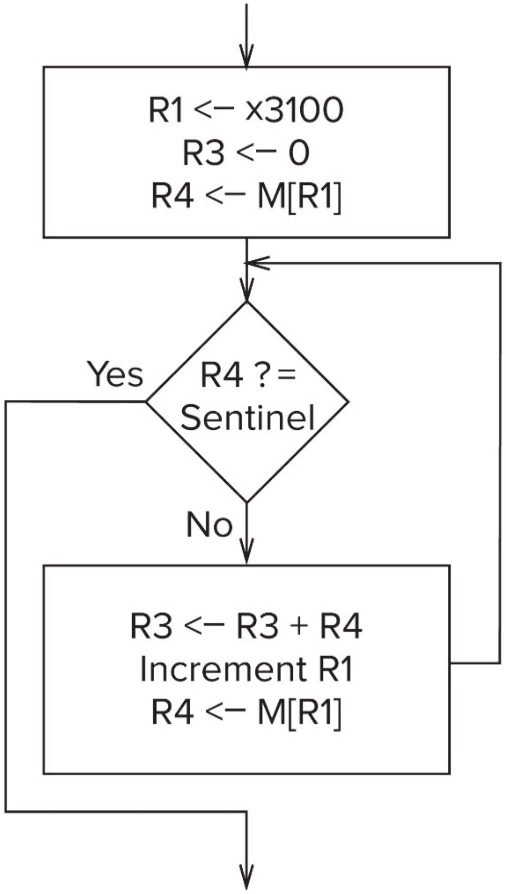
R2 counts down from 12. Branch out of the loop when R2 becomes zero.

Access the text alternative for slide images.

# Example: Loop with Sentinel

Compute the sum of integers starting at x3100, until a negative integer is found. Instructions start at x3000.






| Address | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x3000 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | R1<- x3100 |
| x3001 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | R3 <- 0 |
| x3002 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | R4 <- M[R1] |
| x3003 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | BRn x3008 |
| x3004 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | R3 <- R3+R4 |
| x3005 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | R1 <- R1+1 |
| x3006 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | R4 <- M[R1] |
| x3007 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | BRnzp x3003 |

In this case, it's the data that tells when to exit the loop. When the value loaded into R4 is negative, take branch to x3008.

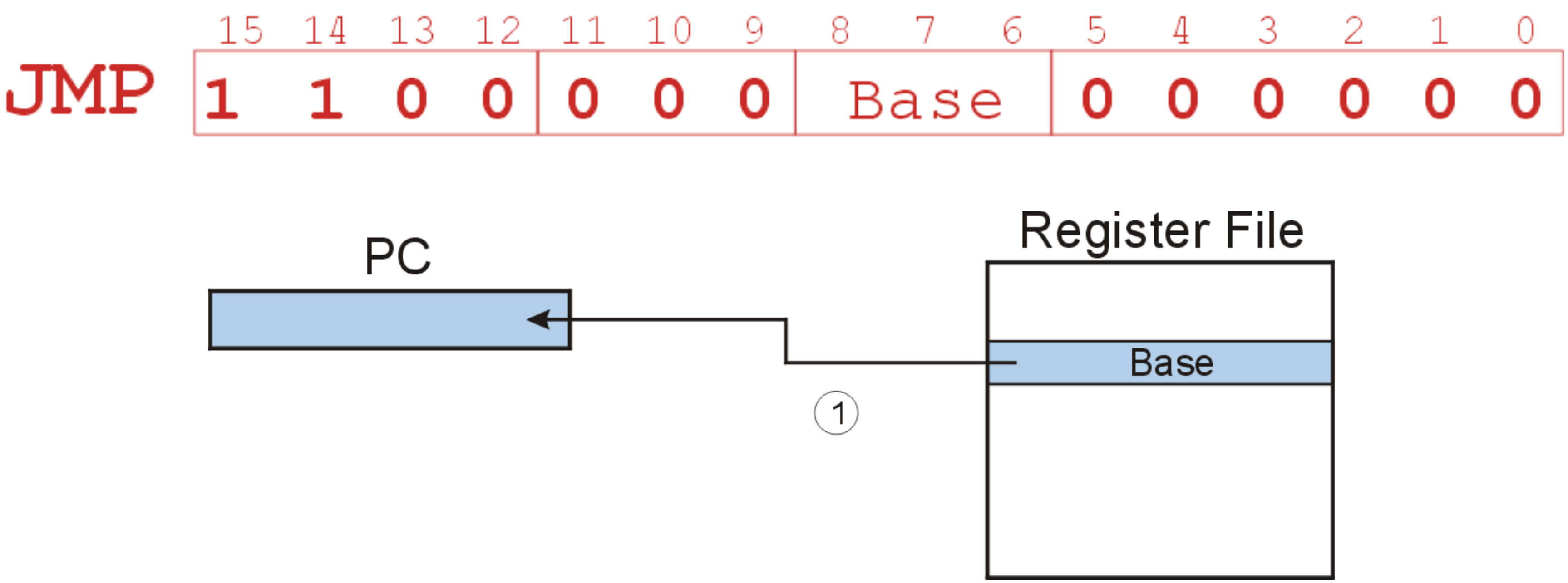
Access the text alternative for slide images.

# Unconditional Branch (JMP)

We can create a branch that is always taken by setting n, z, and p to 1.

However, the target address of BR is limited by the 9-bit PC offset.

The JMP instruction provides an unconditional branch to any target location by using the contents of a register as the target address. It simply copies the register into the PC.




Access the text alternative for slide images.

# TRAP: Invoke a System Service Routine

The TRAP instruction is used to give control to the operating system to perform a task that user code is not allowed to do. The details will be explained in Chapter 9.


For now, you just need to know that bits [7:0] hold a "trap vector" -- a unique code that specifies the service routine. The service routines used in this part of the course are:

| trapvector | service routine |
|---|---|
| x23 | Input a character from the keyboard. |
| x21 | Output a character to the monitor |
| x25 | Halt the processor. |

Access the text alternative for slide images.

# Input / Output Service Routines

Getting character input from the keyboard

- TRAP x23 is used to invoke the keyboard input service routine.
- When the OS returns control to our program, the ASCII code for the key pressed by the user will be in R0.

Sending character output to the monitor

- TRAP x21 is used to invoke the monitor output service routine.
- Before invoking the routine, put the ASCII character to be output into R0.

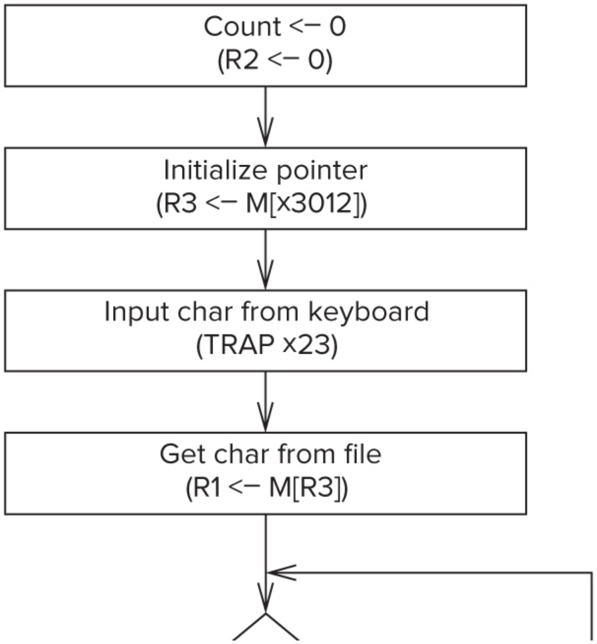
# Example Program: Count Occurrences of a Character

We want to count the number of times a user-specified character appears in a text file, and then print the count to the monitor.

- The text file is stored in memory as a sequence of ASCII characters.
- The end of the file is denoted with the EOT character (x04).
  NOTE: We do not know how many characters are in the file; EOT is the sentinel value that signals when we are done.
- A pointer to the file will be stored at the end of the program. (A "pointer" is a memory address; it will be the address of the first character in the file.)
- We will assume that the character will appear no more than 9 times.
- Program instructions will start at x3000. The file data can be anywhere in memory.

# Part 1: Initializing Registers







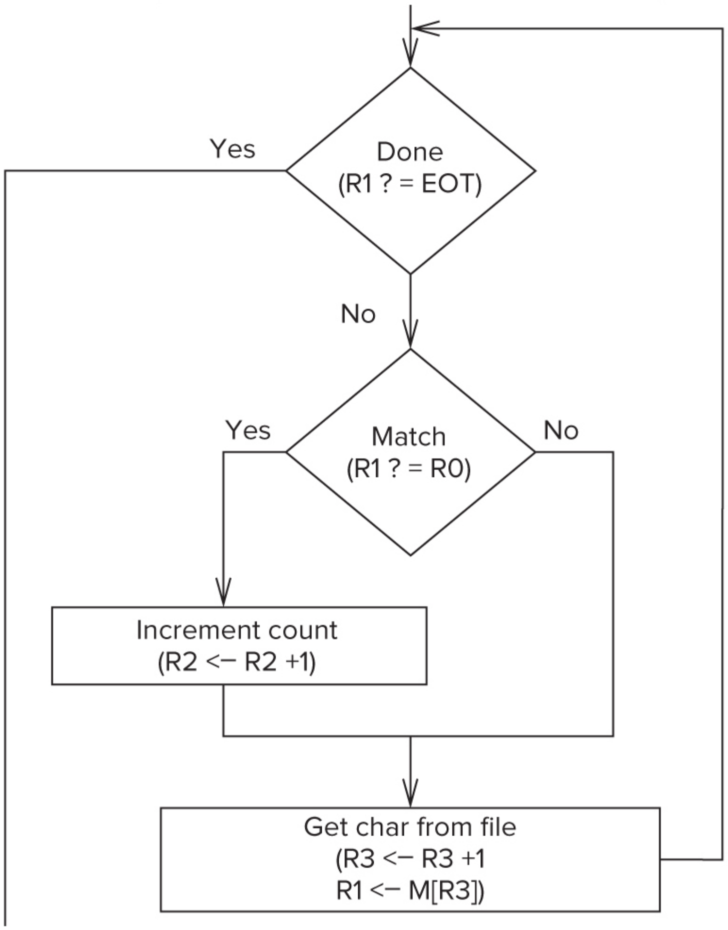

| Address | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x3000 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | R2 <- 0 |
| x3001 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | R3 <- M[x3012] |
| x3002 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | TRAP x23 |
| x3003 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | R1 <- M[R3] |
| x3004 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | R4 <- R1-4 |
| x3005 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | BRz x300E |
| x3006 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | R1 <- NOT R1 |
| x3007 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | R1 <- R1 + 1 |
| x3008 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | R1 <- R1 + R0 |
| x3009 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | BRnp x300B |
| x300A | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | R2 <- R2 + 1 |
| x300B | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | R3 <- R3 + 1 |
| x300C | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | R1 <- M[R3] |
| x300D | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | BRnzp x3004 |
| x300E | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | R0 <- M[x3013] |
| x300F | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | R0 <- R0 + R2 |
| x3010 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | TRAP x21 |
| x3011 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | TRAP x25 |
| x3012 | Starting address of file | | | | | | | | | | | | | | | | |
| x3013 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | ASCII TEMPLATE |

R2 is counter. R3 is address of first character to read from file.
R0 is character from keyboard. R1 is first character from file.

[Access the text alternative for slide images.]

# Part 2: Read Characters and Count







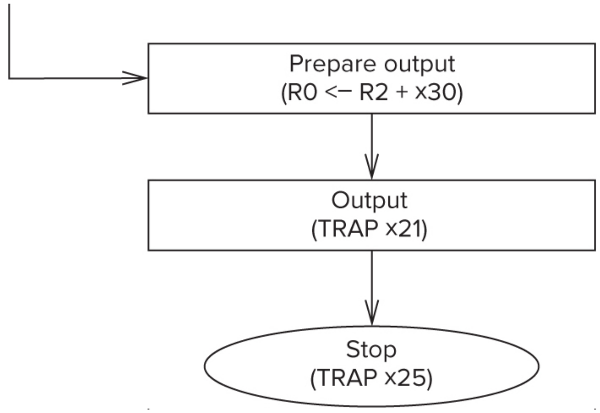

| Address | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x3000 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | R2 <- 0 |
| x3001 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | R3 <- M[x3012] |
| x3002 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | TRAP x23 |
| x3003 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | R1 <- M[R3] |
| x3004 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | R4 <- R1-4 |
| x3005 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | BRz x300E |
| x3006 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | R1 <- NOT R1 |
| x3007 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | R1 <- R1 + 1 |
| x3008 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | R1 <- R1 + R0 |
| x3009 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | BRnp x300B |
| x300A | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | R2 <- R2 + 1 |
| x300B | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | R3 <- R3 + 1 |
| x300C | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | R1 <- M[R3] |
| x300D | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | BRnzp x3004 |
| x300E | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | R0 <- M[x3013] |
| x300F | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | R0 <- R0 + R2 |
| x3010 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | TRAP x21 |
| x3011 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | TRAP x25 |
| x3012 | Starting address of file | | | | | | | | | | | | | | | | |
| x3013 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | ASCII TEMPLATE |

Compare character to x04 (EOT). If equal, exit loop. Otherwise, count if matches user input and read the next character.

Access the text alternative for slide images.

# Part 3: Output Count and HALT







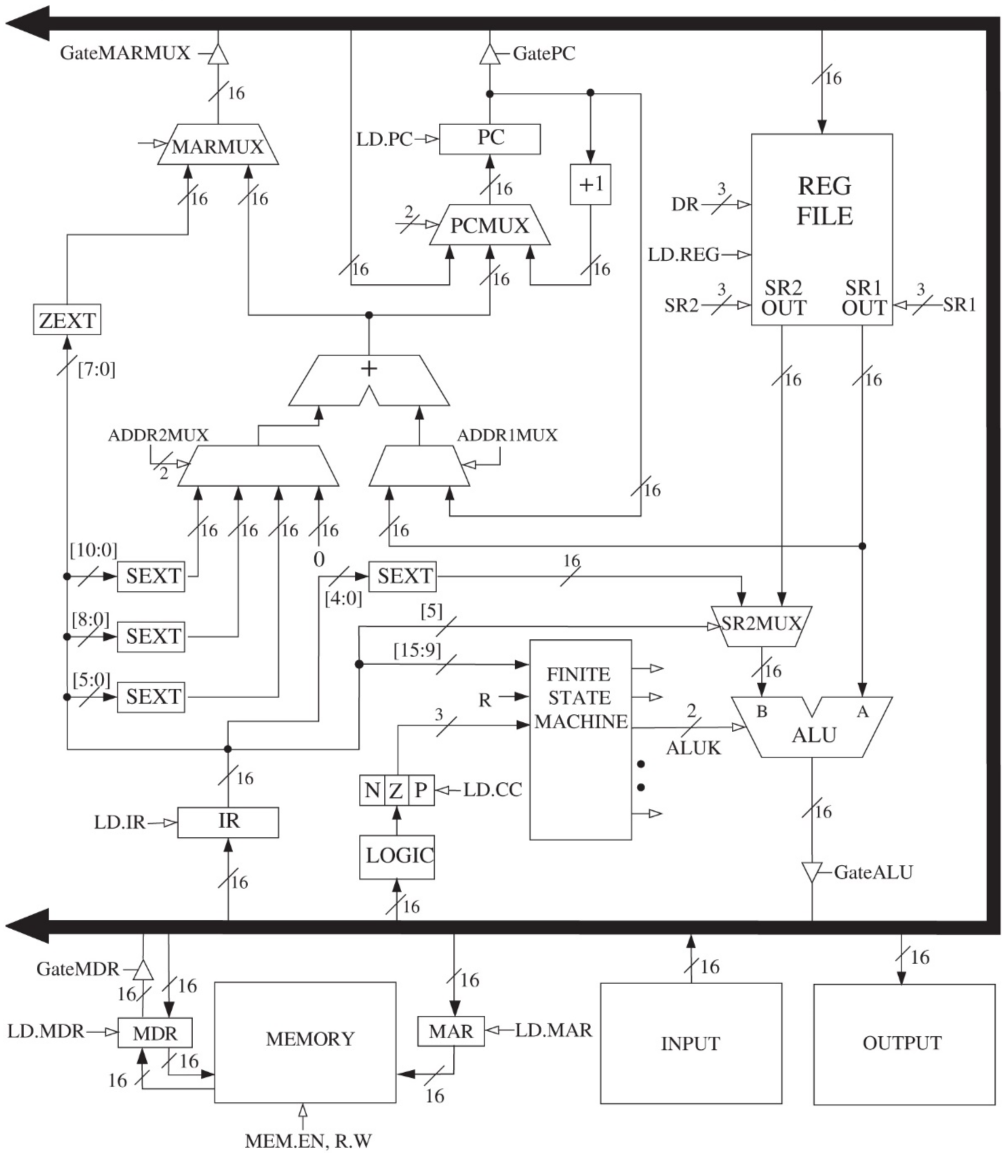

| Address | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x3000 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | R2 <- 0 |
| x3001 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | R3 <- M[x3012] |
| x3002 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | TRAP x23 |
| x3003 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | R1 <- M[R3] |
| x3004 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | R4 <- R1-4 |
| x3005 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | BRz x300E |
| x3006 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | R1 <- NOT R1 |
| x3007 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | R1 <- R1 + 1 |
| x3008 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | R1 <- R1 + R0 |
| x3009 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | BRnp x300B |
| x300A | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | R2 <- R2 + 1 |
| x300B | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | R3 <- R3 + 1 |
| x300C | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | R1 <- M[R3] |
| x300D | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | BRnzp x3004 |
| x300E | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | R0 <- M[x3013] |
| x300F | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | R0 <- R0 + R2 |
| x3010 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | TRAP x21 |
| x3011 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | TRAP x25 |
| x3012 | Starting address of file | | | | | | | | | | | | | | | | |
| x3013 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | ASCII TEMPLATE |

When loop is finished, convert count (R2) to the corresponding ASCII character by adding '0' (x30). Output the character and halt the program.

Access the text alternative for slide images.

# LC-3 Data Path 1






Data path is used to execute LC-3 programs.

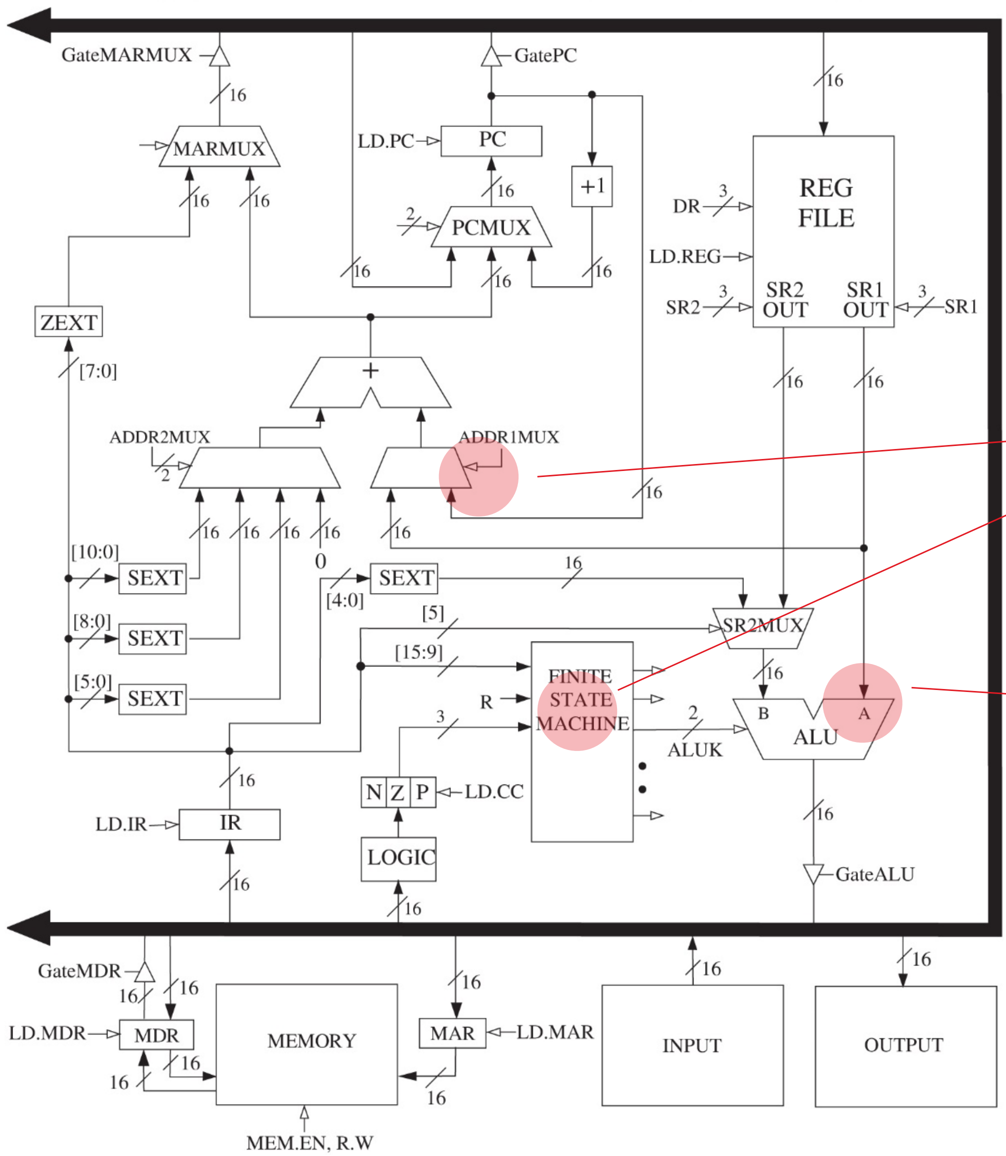
PC is initialized to point to the first instruction. Clock is enabled, and the control unit takes over.

Next slides will give a little more detail on various components.

Access the text alternative for slide images

# LC-3 Data Path 2





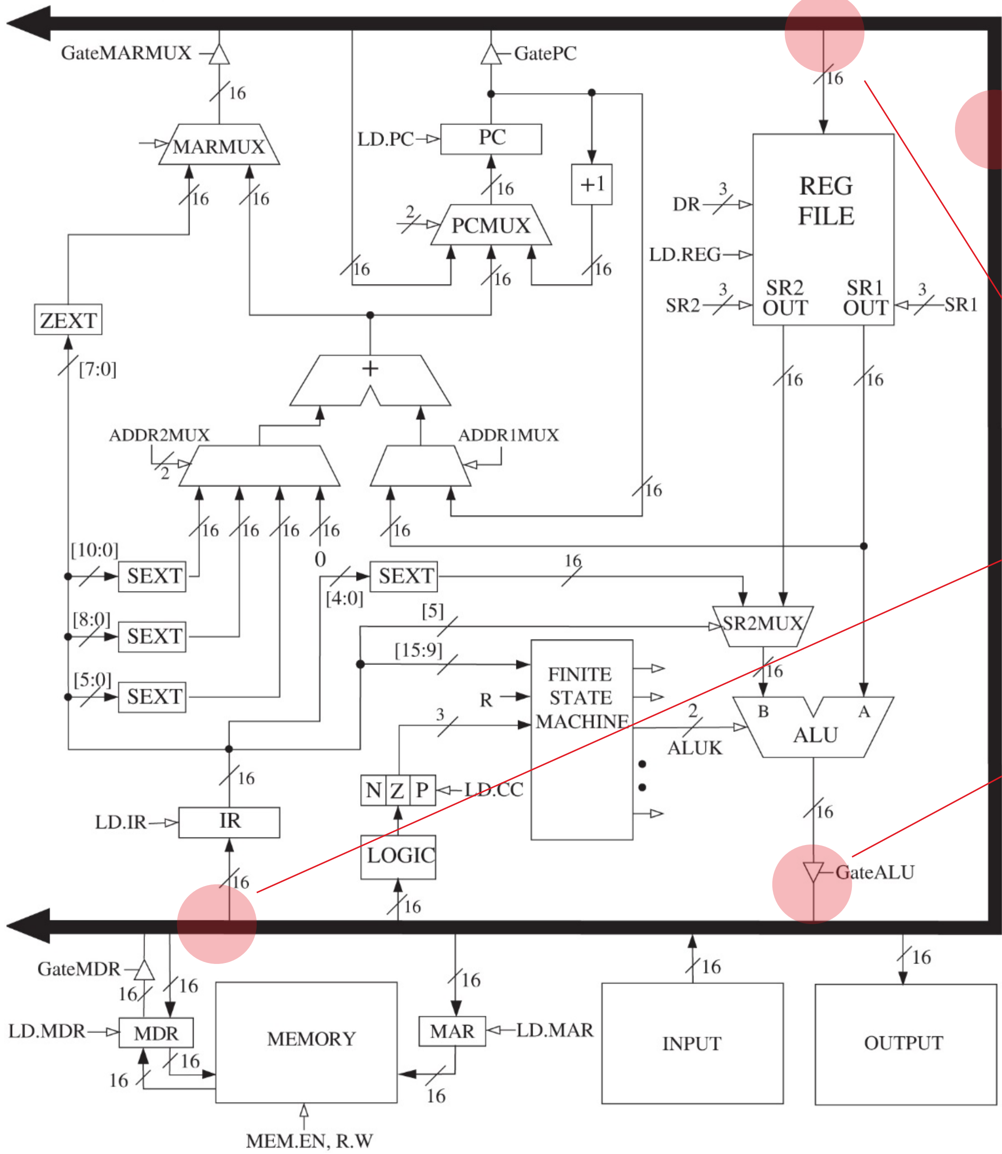

Arrows with open heads represent control signals from FSM.

Arrows with filled heads represent data that is processed.

Access the text alternative for slide images

# LC-3 Data Path 3






**Global bus** is a set of wires that allow various components to transfer 16-bit data to other components.

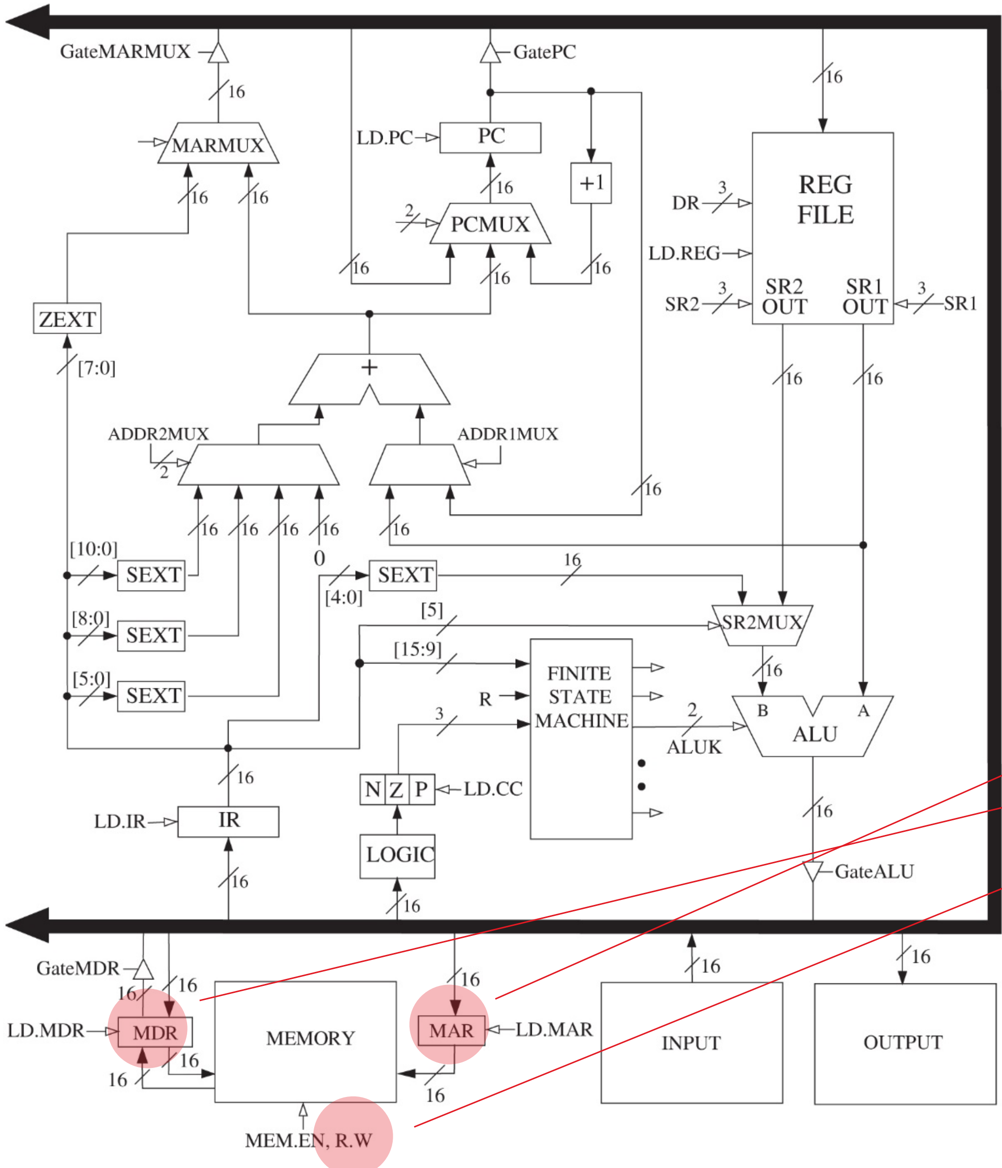
One or more components may read data from the bus on any cycle.

Tri-state device determines which component puts data on the bus. Only one source of data at any time.

Access the text alternative for slide images

# LC-3 Data Path $_4$






Memory interface:

MAR

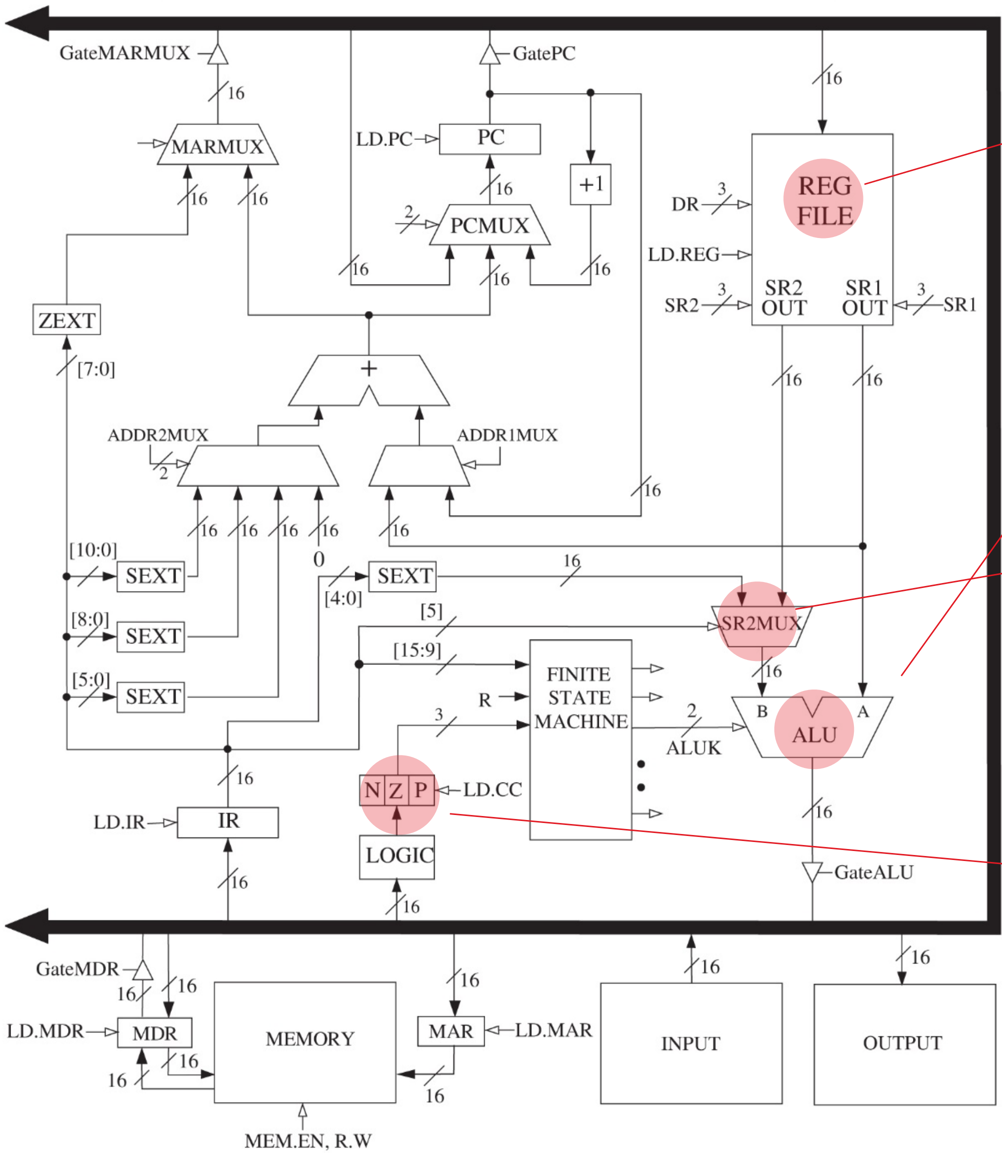
MDR

Read/Write control

Access the text alternative for slide images

# LC-3 Data Path 5






**Register File (R0-R7)**
Control signals specify two source register (SR1, SR2) and one destination (DR).

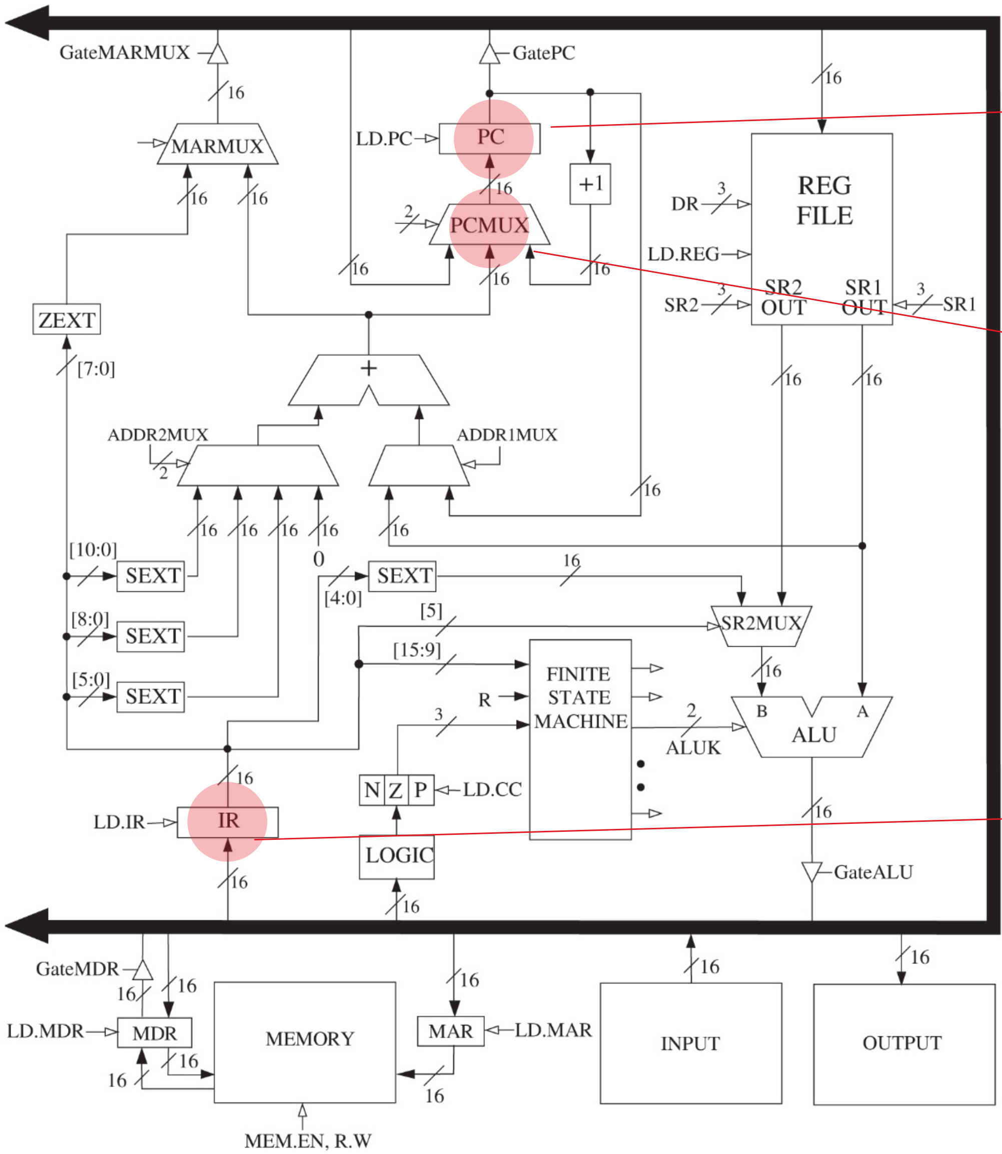
**ALU** performs ADD, AND, NOT. Operand A always comes from register file. Operand B is from register file or IR. Output goes to bus, to be written into register file.

**Condition codes** are set by looking at data placed on the bus by ALU or memory (MDR).

Access the text alternative for slide images

# LC-3 Data Path 6






**PC** puts address on bus. Placed in MAR during Fetch.

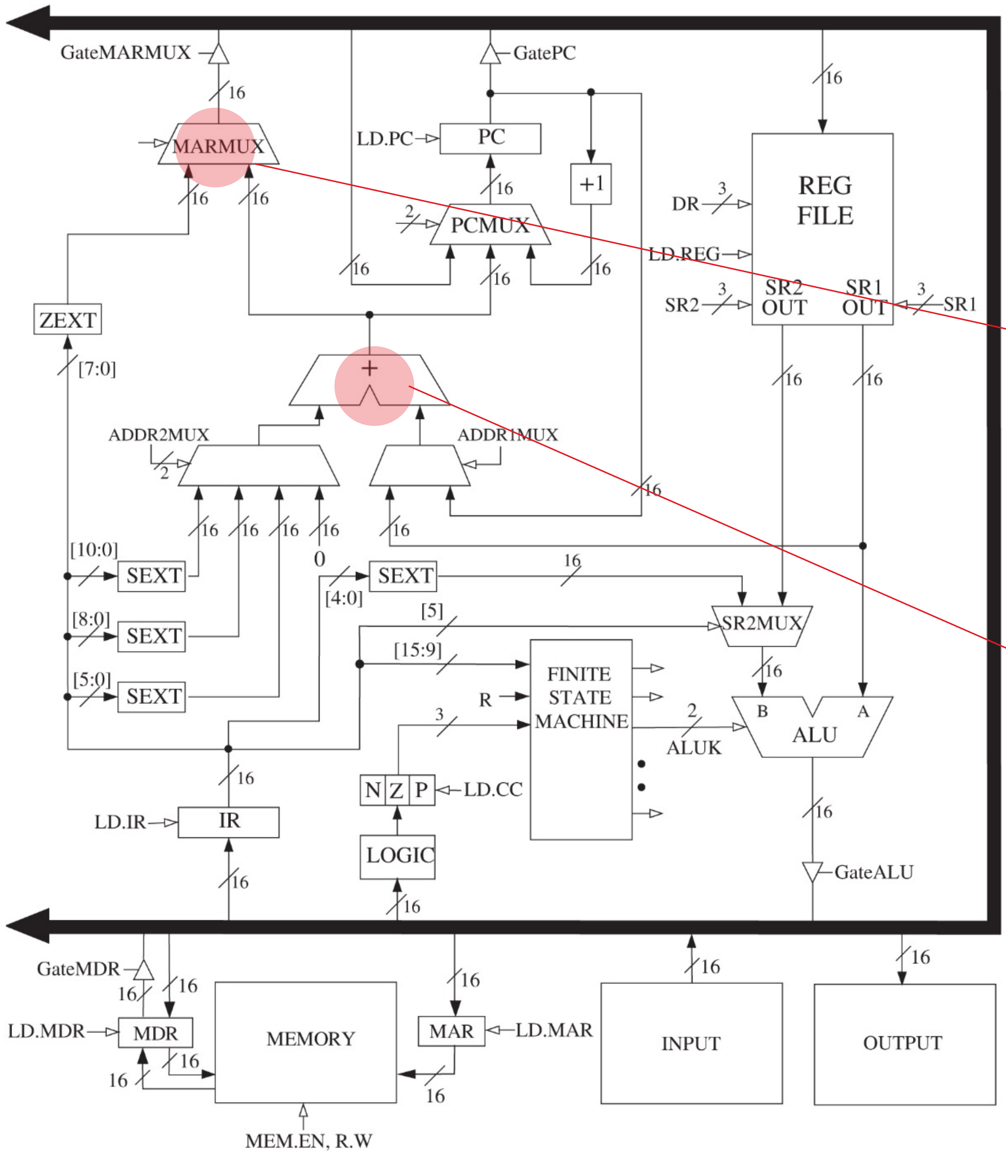
**PCMUX** allows various values to be written to PC: incremented PC (Fetch), computed address (BR), or register data from bus (JMP).

**IR** gets data from bus (MDR) during Fetch.

Access the text alternative for slide images

# LC-3 Data Path 






**MARMUX** chooses value to be written to MAR during load, store, or TRAP.

**Evaluate Address** phase adds offset to PC or register for load, store, BR.

Access the text alternative for slide images

## Chapter Summary

LC-3 is a simple, but complete, computer architecture.

Remember Chapter 1? LC-3 is capable of computing anything, given enough time and memory. Memory is limited to 64K words, but that's enough to solve many interesting problems.

We have presented the details of 12 opcodes -- the other 3 will be covered in Chapters 8 and 9.

We will now focus more on **software**:

- Writing algorithms to solve computational problems.
- Expressing those algorithms in machine instructions (Ch 6), assembly language (Ch 7 to 10), and higher-level languages (Ch 11 to 20).
- Using data structures -- arrangements of data in memory -- and subroutines/functions to write modular, reusable, and maintainable programs.

Because learning changes everything.®

www.mheducation.com