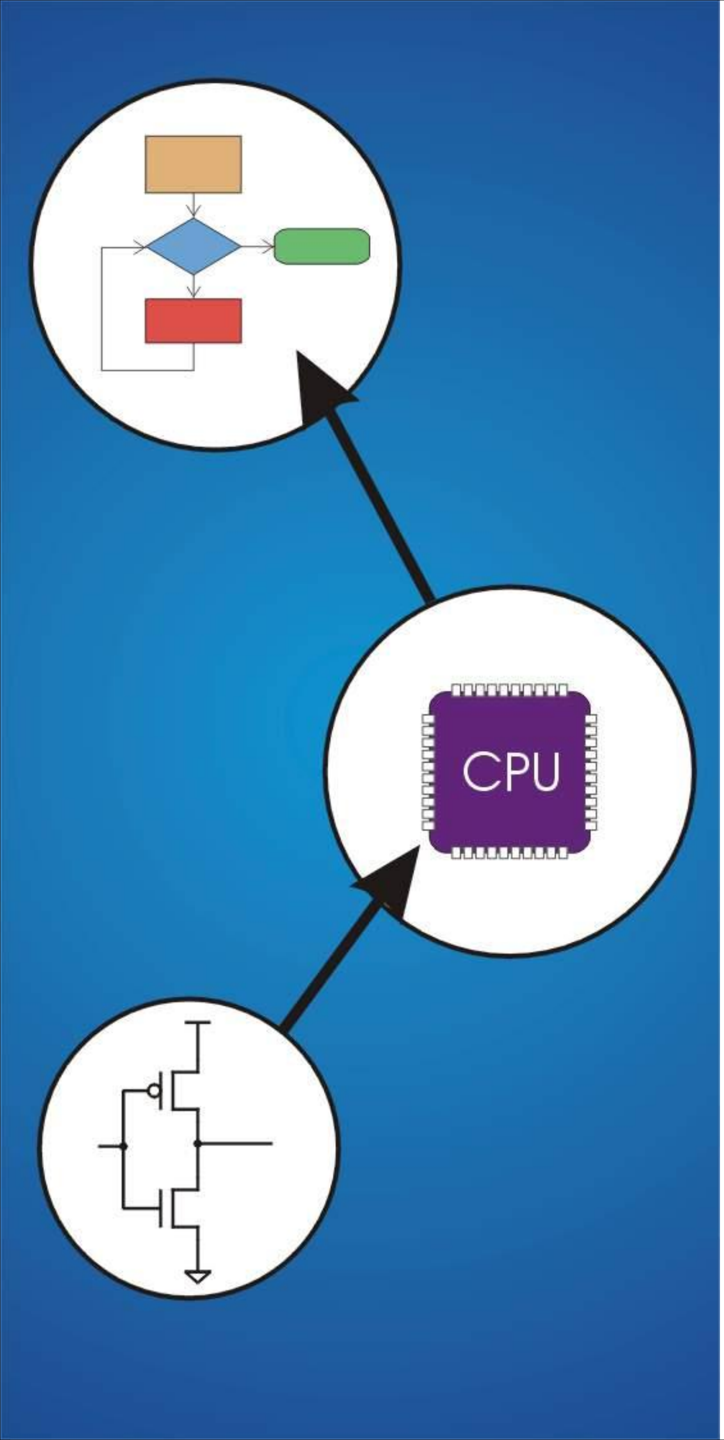


Chapter 1

The Von Neumann Model LC-3 at ISA and Assembly Language

(Review)

Scamper …
The Stored Program Computer

1943: ENIAC

- **Presper Eckert and John Mauchly -- first general electronic computer.** (or was it John V. Atanasoff in 1939?)
- **Hard-wired program -- settings of dials and switches.**

1944: Beginnings of EDVAC

- **among other improvements, includes program stored in memory**

1945: John von Neumann

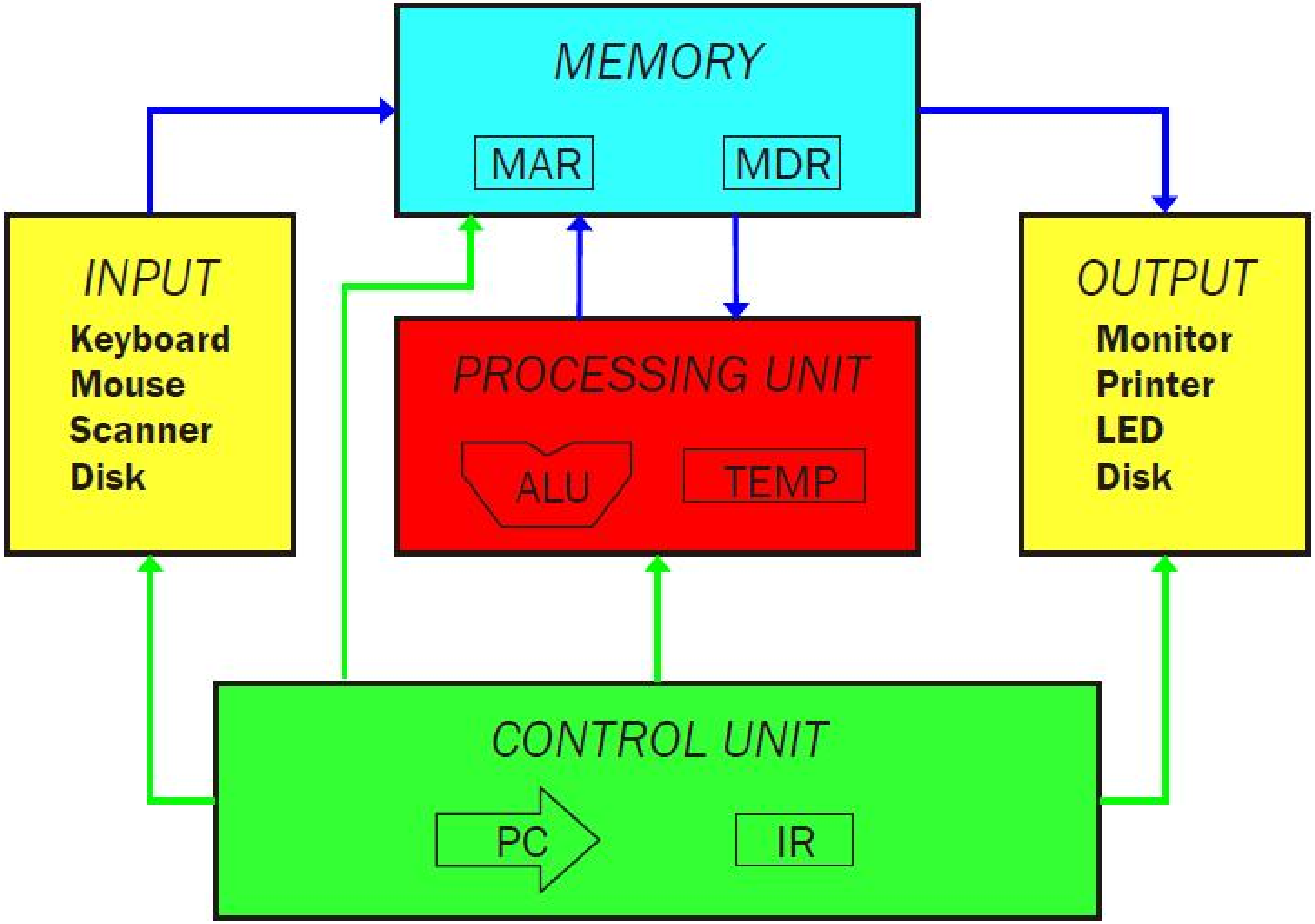
- **wrote a report on the stored program concept, known as the *First Draft of a Report on EDVAC***

The basic structure proposed in the draft became known as the "von Neumann machine" (or model).

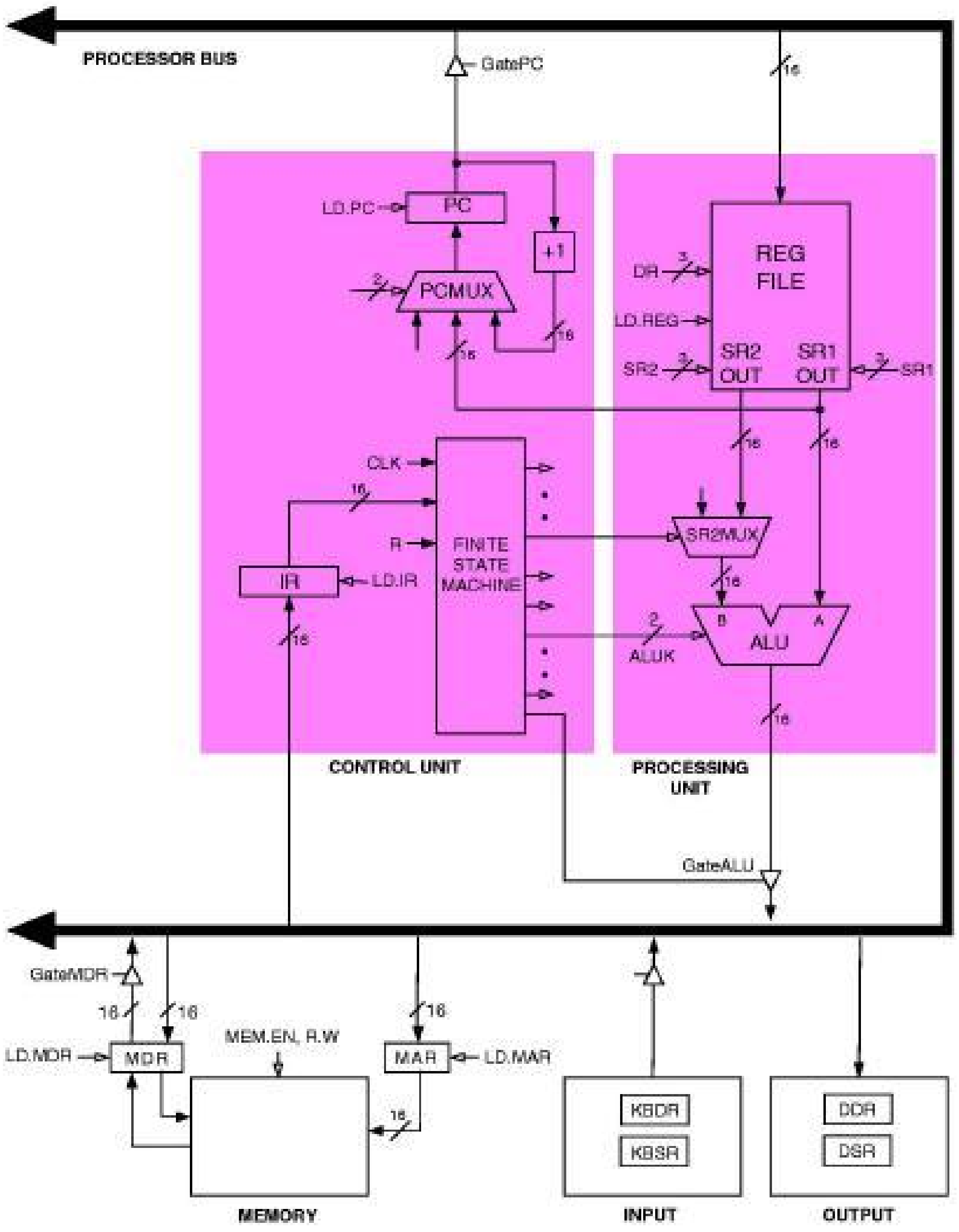
- **a *memory*, containing instructions and data**
- **a *processing unit*, for performing arithmetic and logical operations**
- **a *control unit*, for interpreting instructions**

For more history, see http://www.maxmon.com/history.htm

Von Neumann Model



The LC-3 as a von Neumann machine



Memory

2^k x *m* array of stored bits

Address

- **unique (*k*-bit) identifier of location**

Contents

- ***m*-bit value stored in location**

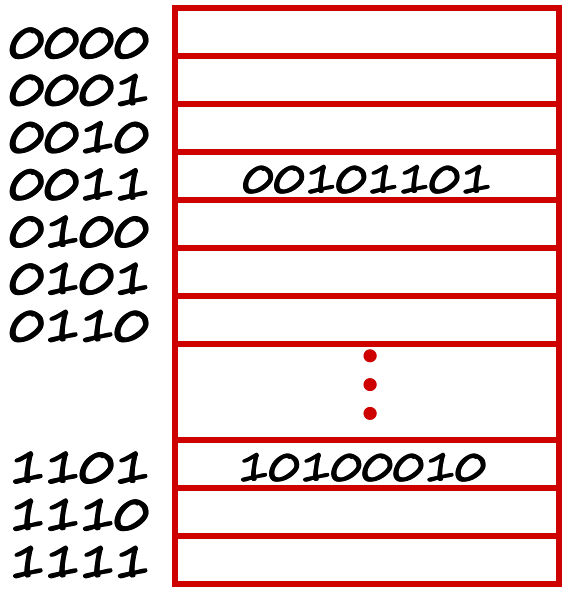
Basic Operations:

LOAD

- **read a value from a memory location**

STORE

- **write a value to a memory location**



Interface to Memory

How does processing unit get data to/from memory?

MAR: Memory Address Register

MDR: Memory Data Register

To LOAD a location (A):

1. **Write the address (A) into the MAR.**
2. **Send a “read” signal to the memory.**
3. **Read the data from MDR.**

To STORE a value (X) to a location (A):

1. **Write the data (X) to the MDR.**
2. **Write the address (A) into the MAR.**
3. **Send a “write” signal to the memory.**

Processing Unit

Functional Units

- ALU = Arithmetic and Logic Unit
- could have many functional units. some of them special-purpose (multiply, square root, …)
- LC-3 performs ADD, AND, NOT

Registers

- Small, temporary storage
- Operands and results of functional units
- LC-3 has eight registers (R0, …, R7), each 16 bits wide

Word Size

- number of bits normally processed by ALU in one instruction
- also width of registers
- LC-3 is 16 bits

Input and Output

Devices for getting data into and out of computer memory

Each device has its own interface, usually a set of registers like the memory's MAR and MDR

- LC-3 supports keyboard (input) and monitor (output)
- keyboard: data register (KBDR) and status register (KBSR)
- monitor: data register (DDR) and status register (DSR)

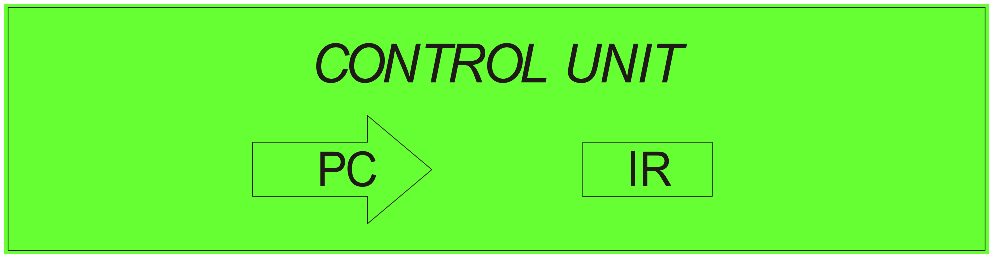
Some devices provide both input and output

- disk, network

Program that controls access to a device is usually called a *driver*.

Control Unit

Orchestrates execution of the program



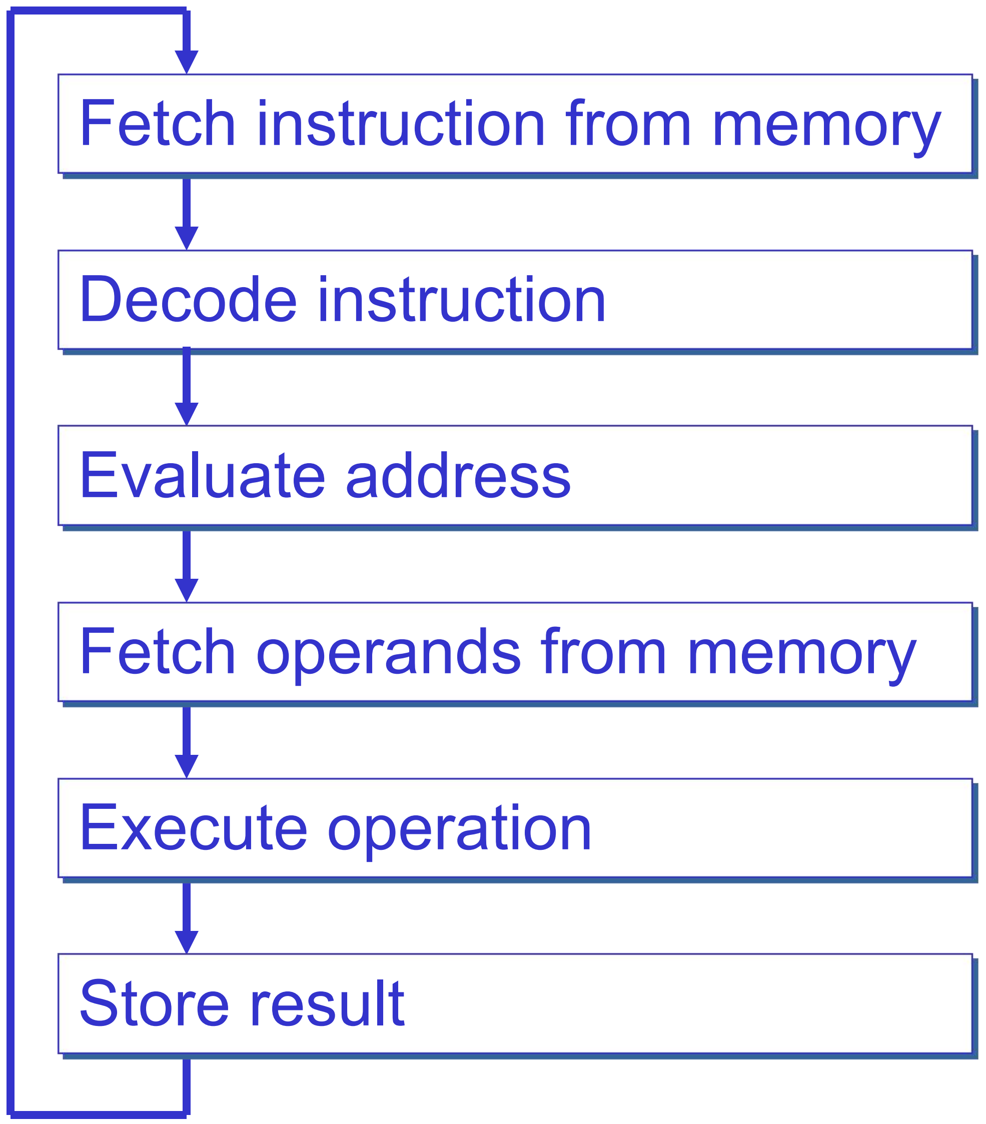
Instruction Register (IR) contains the *current instruction*.

Program Counter (PC) contains the *address* of the next instruction to be executed.

Control unit:

- **reads an instruction from memory**
  - **the instruction's address is in the PC**
- **interprets the instruction, generating signals that tell the other components what to do**
  - **an instruction may take many *machine cycles* to complete**

Instruction Processing



Instruction

The instruction is the fundamental unit of work.

Specifies two things:

- ***opcode*: operation to be performed**
- ***operands*: data/locations to be used for operation**

An instruction is encoded as a sequence of bits.
(Just like data!)

- Often, but not always, instructions have a fixed length, such as 16 or 32 bits.
- Control unit interprets instruction: generates sequence of control signals to carry out operation.
- Operation is either executed completely, or not at all.

A computer's instructions and their formats is known as its *Instruction Set Architecture (ISA)*.

Example: LC-3 ADD Instruction

LC-3 has 16-bit instructions.

- **Each instruction has a four-bit opcode, bits [15:12].**

LC-3 has eight *registers* (R0-R7) for temporary storage.

- **Sources and destination of ADD are registers.**

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

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

"Add the contents of R2 to the contents of R6, and store the result in R6."

Instruction Processing Summary

Instructions look just like data -- it's all interpretation.

Three basic kinds of instructions:

- **computational instructions (ADD, AND, …)**
- **data movement instructions (LD, ST, …)**
- **control instructions (JMP, BRnz, …)**

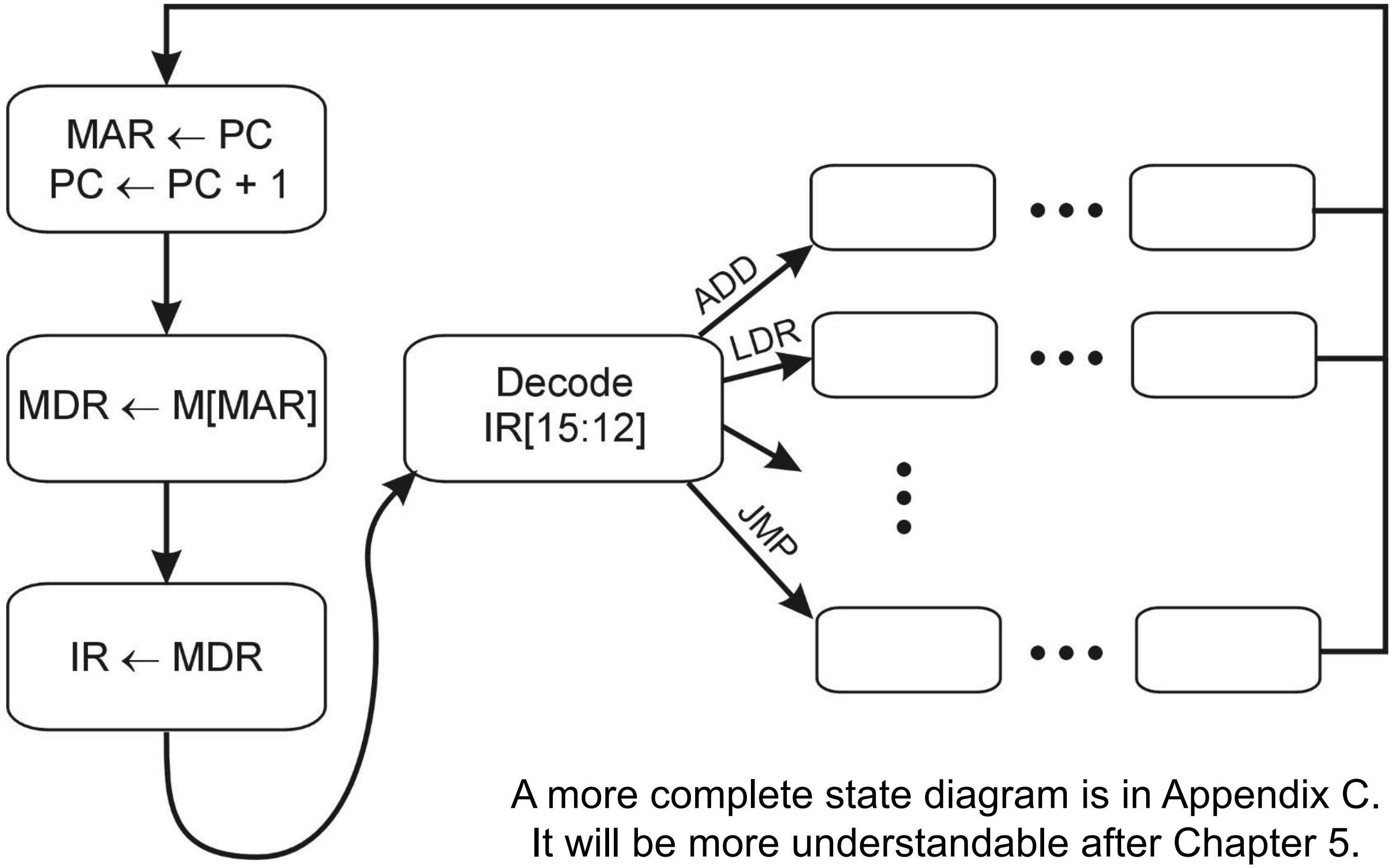
Six basic phases of instruction processing:

$$F \rightarrow D \rightarrow EA \rightarrow OP \rightarrow EX \rightarrow S$$

- **not all phases are needed by every instruction**
- **phases may take variable number of machine cycles**

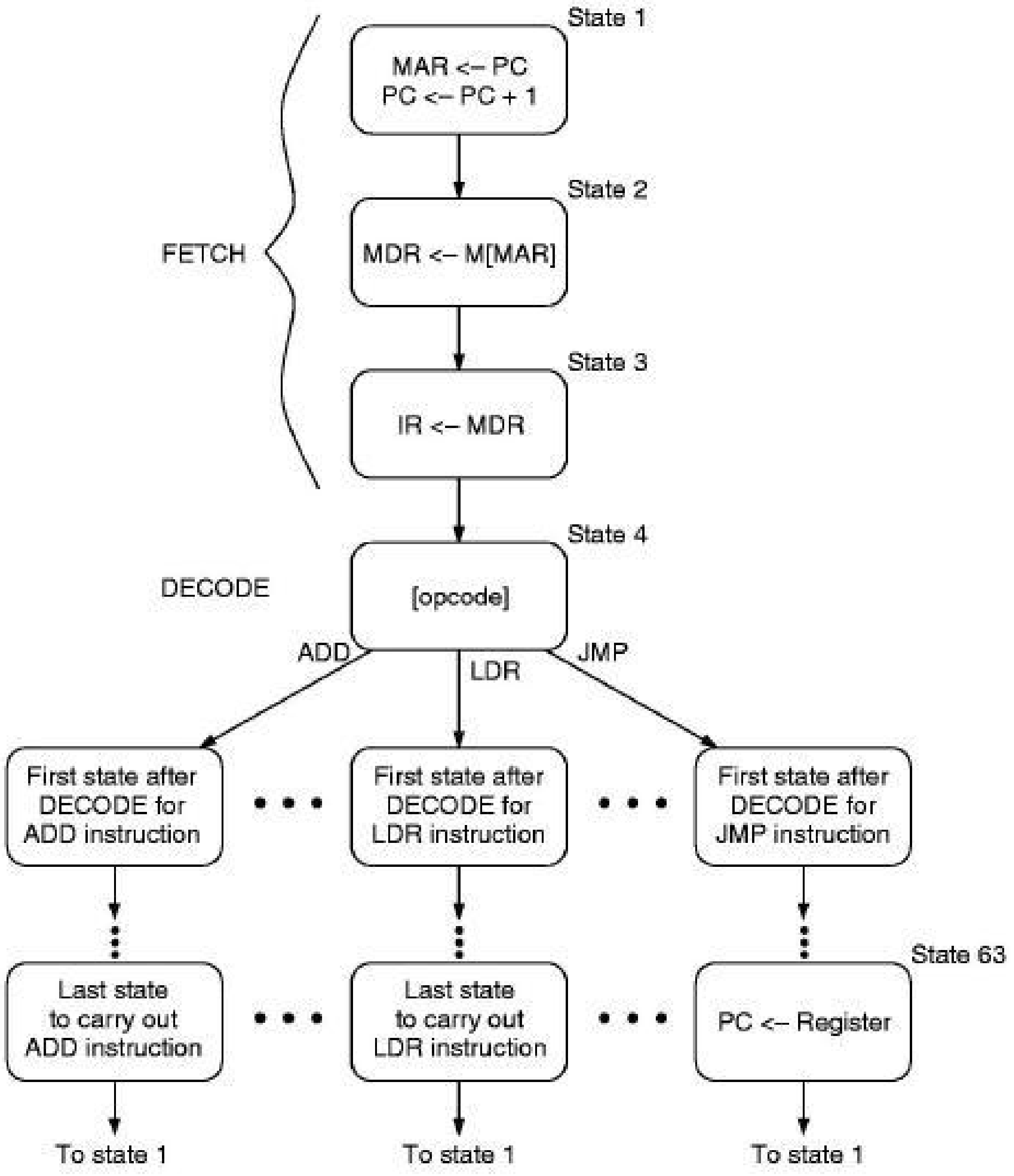
Control Unit State Diagram

The control unit is a state machine. Here is part of a simplified state diagram for the LC-3:



A more complete state diagram is in Appendix C.
It will be more understandable after Chapter 5.

LC3 FSM diagram


State 1
MAR <– PC
PC <– PC + 1
FETCH
State 2
MDR <– M[MAR]
State 3
IR <– MDR
State 4
DECODE
[opcode]
ADD
LDR
JMP
First state after DECODE for ADD instruction
First state after DECODE for LDR instruction
First state after DECODE for JMP instruction
Last state to carry out ADD instruction
Last state to carry out LDR instruction
State 63
PC <– Register
To state 1
To state 1
To state 1

The LC-3 Instruction set

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	
NOT	1001	DR	SR	111111		
BR	0000	n	z	p	PCoffset9	
JMP	1100	0	00	BaseR	000000	
JSR	0100	1	PCoffset11			
JSRR	0100	0	00	BaseR	000000	
RET	1100	0	00	111	000000	

LD	0010	DR	PCoffset9	
LDI	1010	DR	PCoffset9	
LDR	0110	DR	BaseR	offset6
LEA	1110	DR	PCoffset9	
ST	0011	SR	PCoffset9	
STI	1011	SR	PCoffset9	
STR	0111	SR	BaseR	offset6
TRAP	1111	0000	trapvect8	
RTI	1000	000000000000		
reserved	1101			

Instruction Set Architecture

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

- memory organization
  - address space -- how may locations can be addressed?
  - addressibility -- 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*

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	
NOT	1001	DR	SR	111111		
BR	0000	n	z	p	PCoffset9	
JMP	1100	0	00	BaseR	000000	
JSR	0100	1	PCoffset11			
JSRR	0100	0	00	BaseR	000000	
RET	1100	0	00	111	000000	

LD	0010	DR	PCoffset9	
LDI	1010	DR	PCoffset9	
LDR	0110	DR	BaseR	offset6
LEA	1110	DR	PCoffset9	
ST	0011	SR	PCoffset9	
STI	1011	SR	PCoffset9	
STR	0111	SR	BaseR	offset6
TRAP	1111	0000	trapvect8	
RTI	1000	000000000000		
reserved	1101			

Operate Instructions

Only three operations: ADD, AND, NOT

Source and destination operands are registers

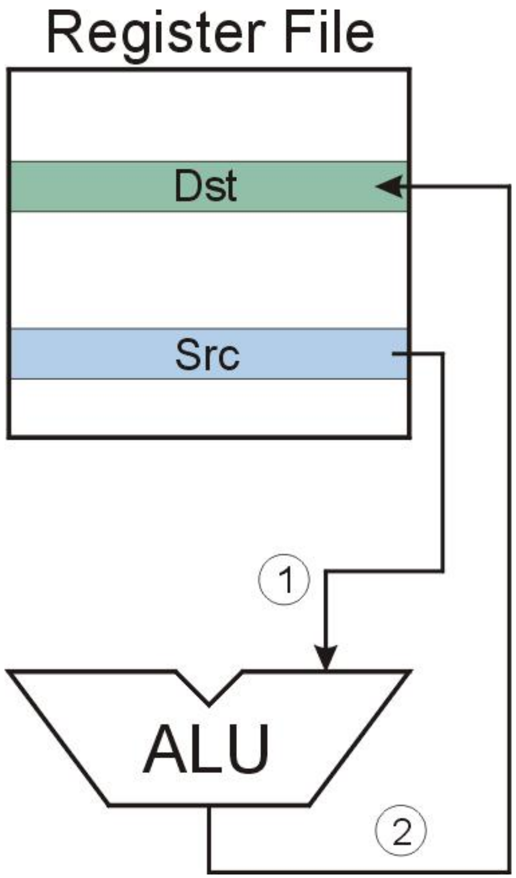
- **These instructions *do not* reference memory.**
- **ADD and AND can use "immediate" mode, where one operand is hard-wired into the instruction.**

Will show dataflow diagram with each instruction.

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

NOT (Register)

	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



Note: Src and Dst could be the <u>same</u> register.

NOT: Bitwise Logical NOT

Assembler Inst.

```
NOT  DR, SR        ; DR = NOT SR
```

Encoding

```
1001  DR  SR  111111
```

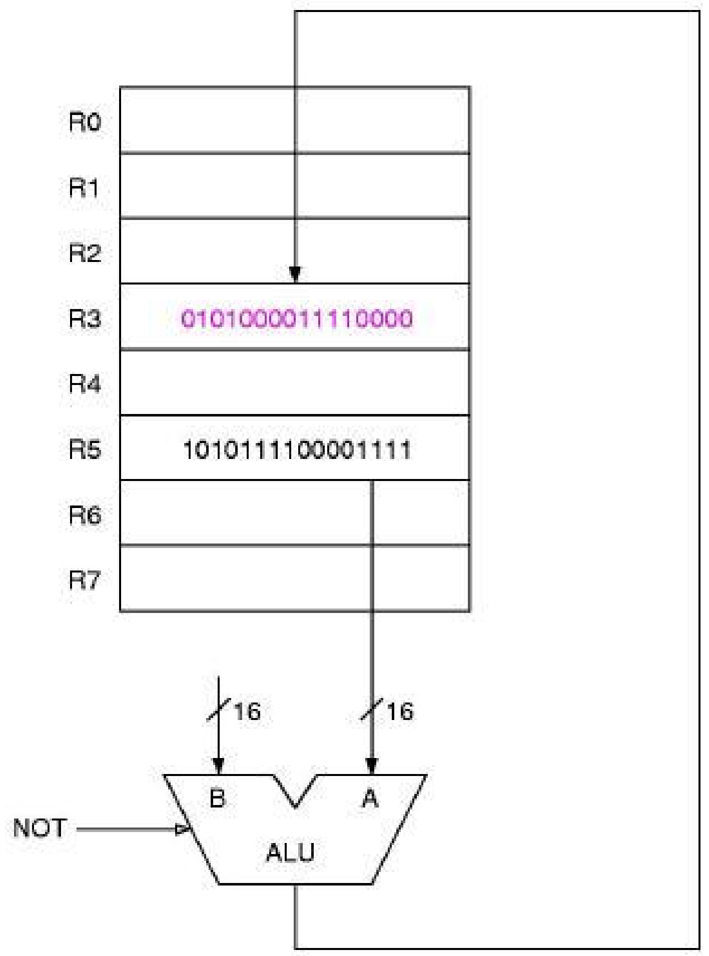
Example

```
NOT   R2, R6
```

- **Note: Condition codes are set.**

NOT data path

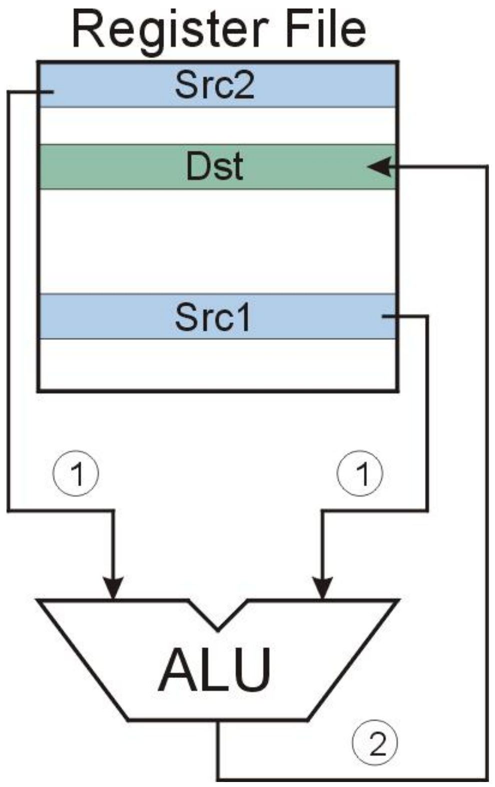
NOT R3, R5



ADD/AND (Register)

this zero means "register mode"

	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		



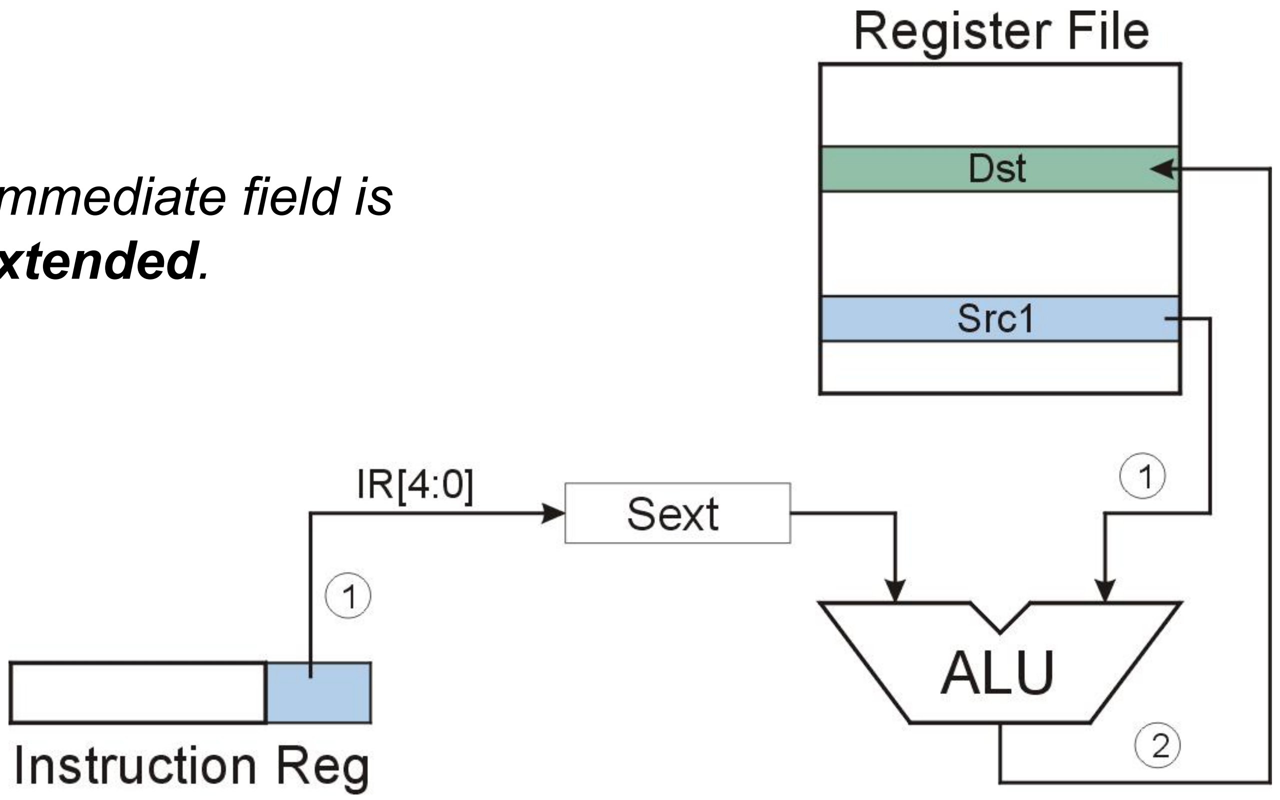
ADD/AND (Immediate)

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

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



ADD: Two's complement 16-bit Addition

Assembler Instruction

```
ADD  DR, SR1, SR2    ; DR = SR1 + SR2  (register addressing)
ADD  DR, SR1, imm5   ; DR = SR1 + Sext(imm5)  (immediate addressing)
```

Encoding

```
0001  DR  SR1  0  00  SR2
0001  DR  SR1  1   imm5
```

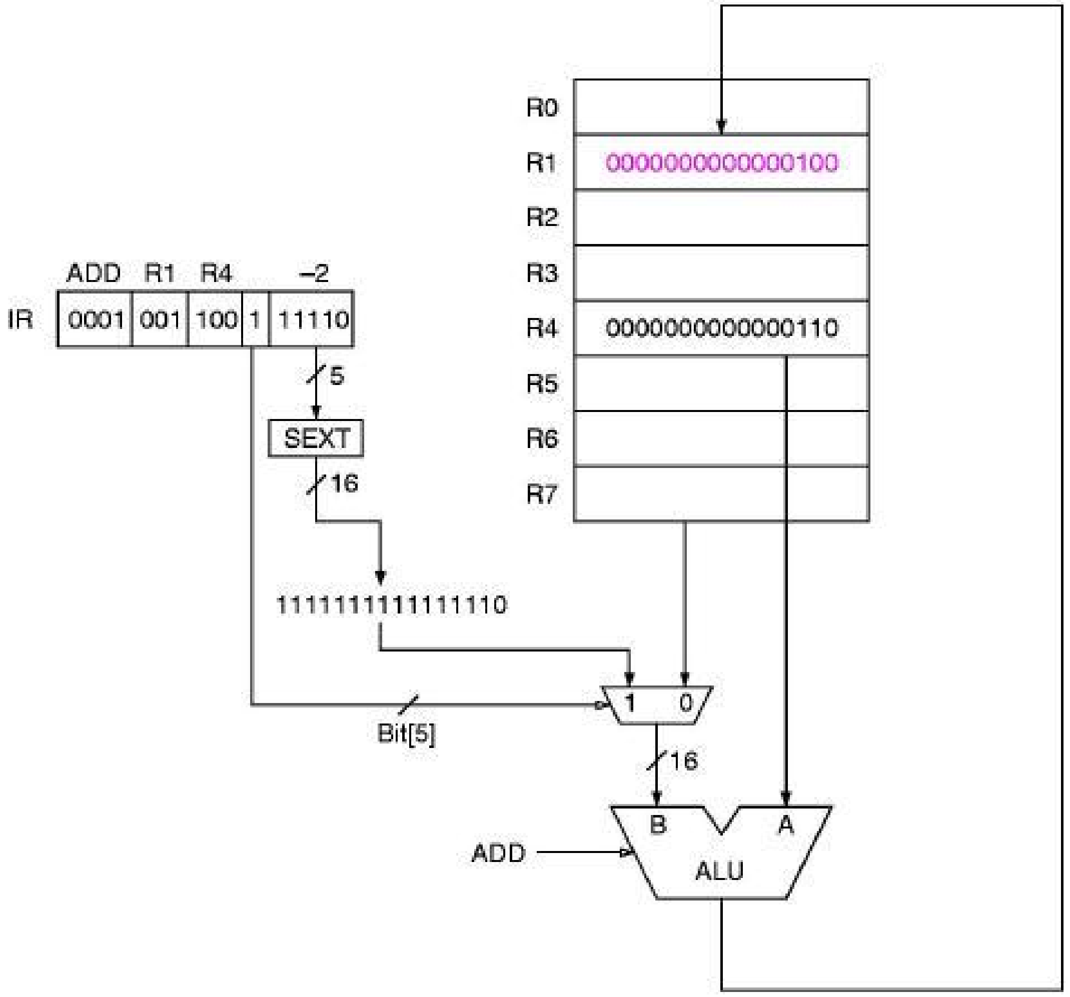
Examples

```
ADD   R1, R4, R5
ADD   R1, R4, # -2
```

- **Note: Condition codes are set**

ADD data path

ADD R1, R4, # -2



AND: Bitwise AND

Assembler Instruction

```
AND  DR, SR1, SR2   ; DR = SR1  AND SR2
AND  DR, SR1, imm5  ; DR = SR1 AND Sext(imm5)
```

Encoding

```
0101  DR  SR1  0  00  SR2
0101  DR  SR1  1   imm5
```

Examples

```
AND   R2, R3, R6
AND   R2, R2, #0    ; Clear R2 to 0
```

Question: ***if the immediate value is only 6 bits, how can it mask the whole of R2?***

- **Note: Condition codes are set.**

How to do them?

1.

```
R2  <-  5
R3  <- -R2
```

2.

```
R2  <-  35
R3  <- -R2
```

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.**

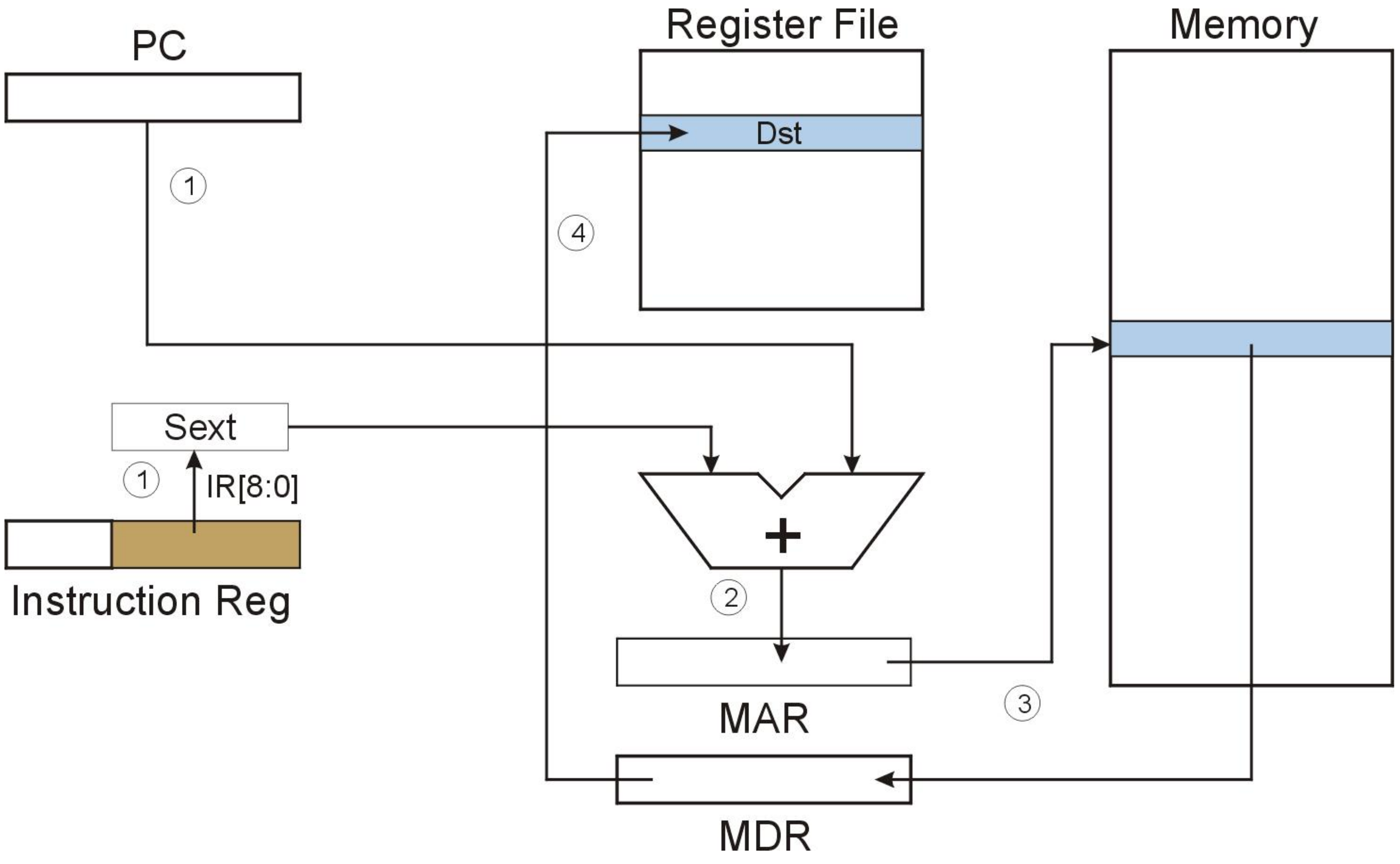
9 bits: $-256 \leq \text{offset} \leq +255$

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)

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



LD: Load Direct

Assembler Inst.

```
LD  DR, LABEL            ; DR <= Mem[LABEL]
```

Encoding

```
0010  DR  PCoffset9
```

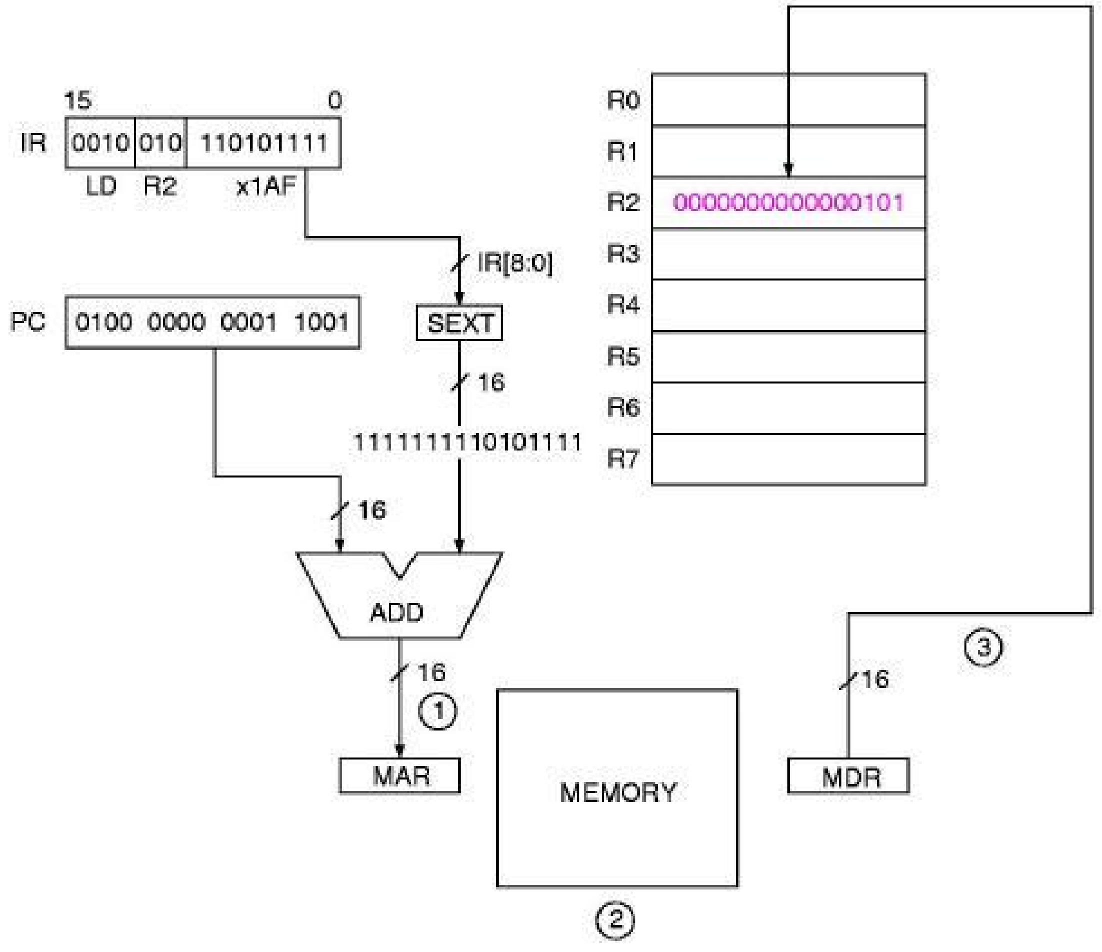
Examples

```
LD   R2, param           ; R2 <= Mem[param]
```

Notes: The LABEL must be within +256/-255 lines of the instruction.
Condition codes are set.

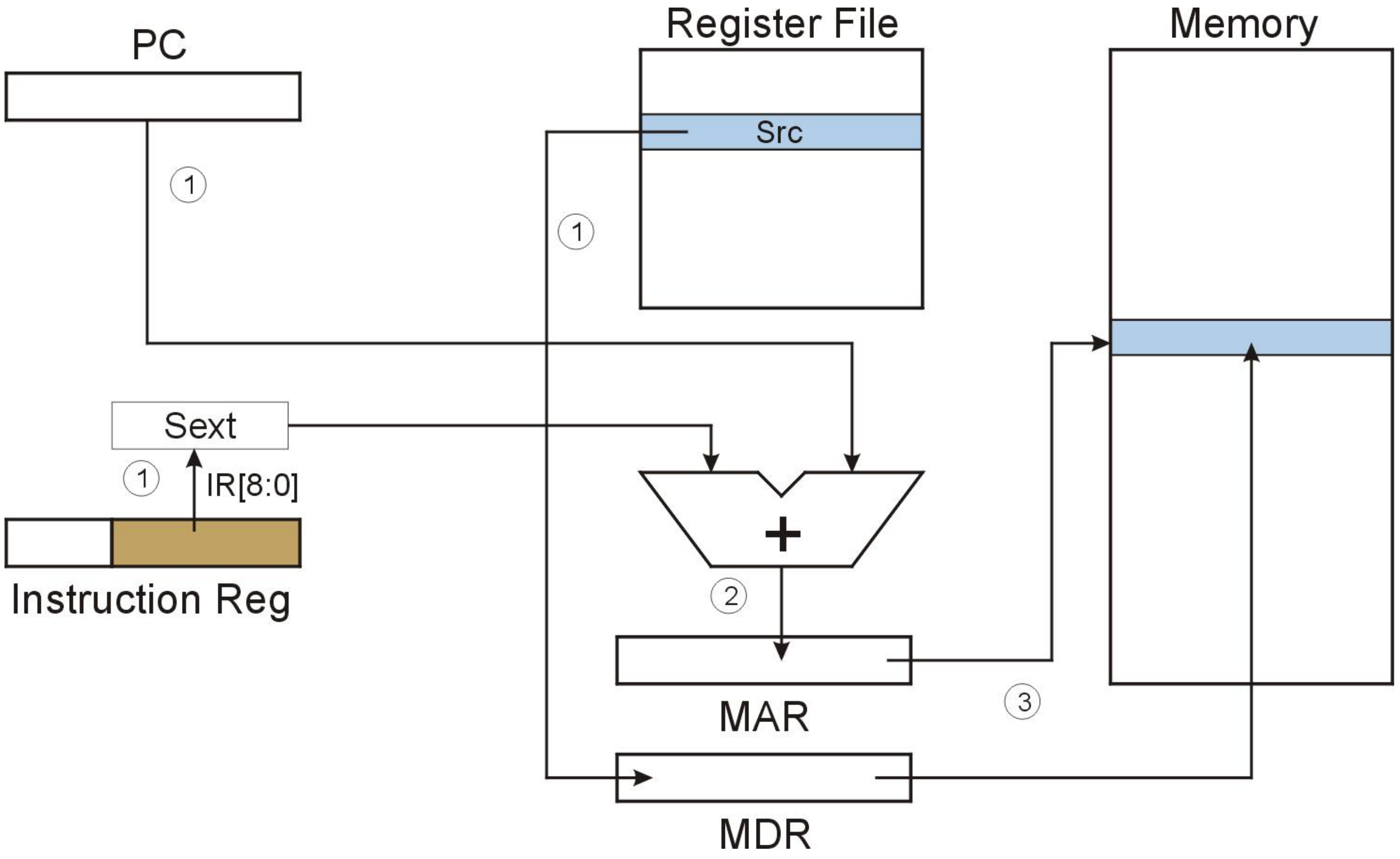
LD data path

LD R2, x1AF



ST (PC-Relative)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ST	0	0	1	1	Src			PCoffset9								



ST: Store Direct

Assembler Inst.

```
ST  SR, LABEL        ;  Mem[LABEL] <= SR
```

Encoding

```
0011  SR  offset9
```

Examples

```
ST   R2, VALUE      ; Mem[VALUE] <= R2
```

Notes: The LABEL must be within +/- 256 lines of the instruction.
Condition codes are NOT set.

Indirect Addressing Mode

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

- **What about the rest of memory?**

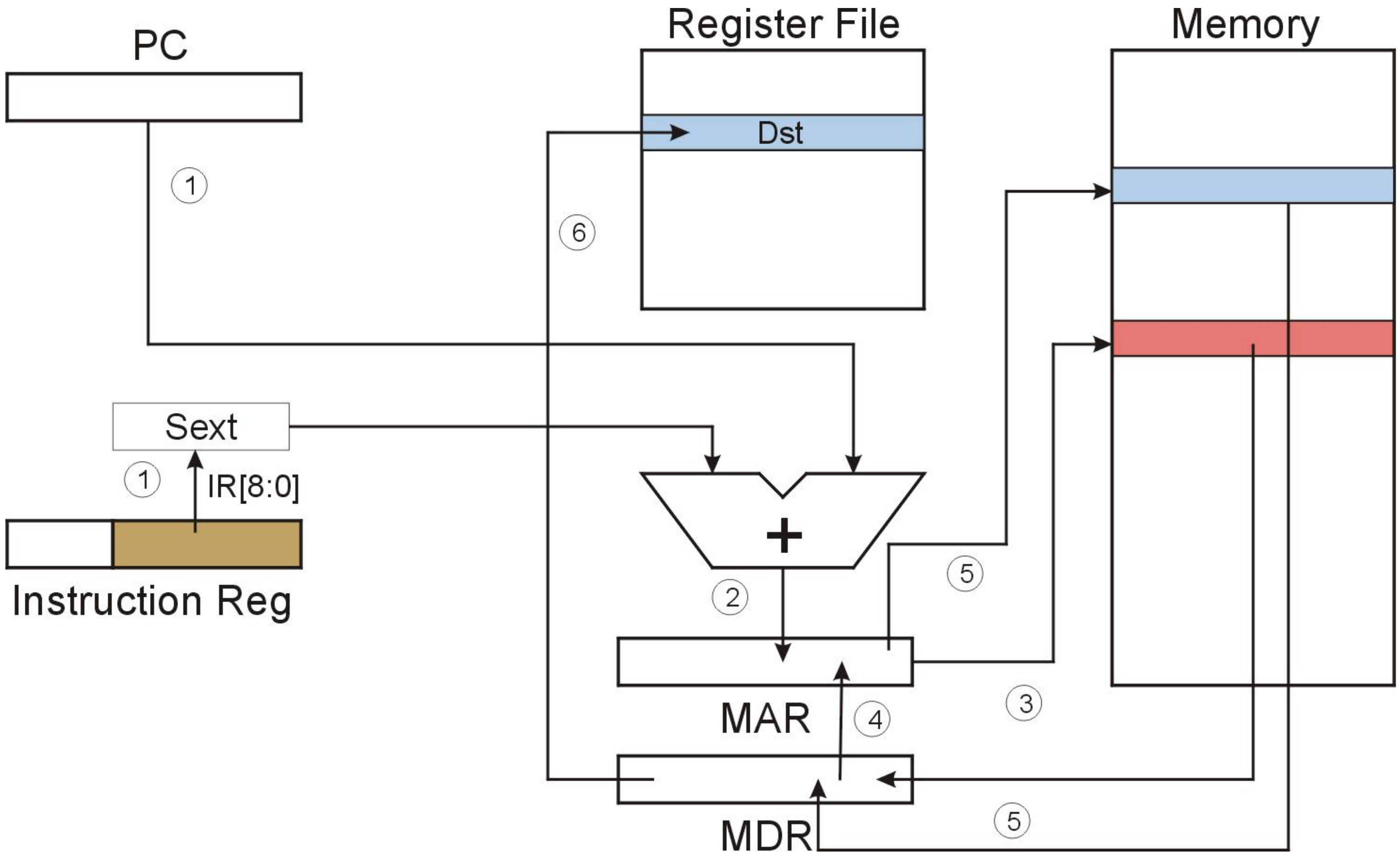
Solution #1:

- **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)

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



LDI: Load Indirect

Assembler Inst.

```
LDI  DR, LABEL            ; DR <= Mem[Mem[LABEL]]
```

Encoding

```
1010  DR  PCoffset9
```

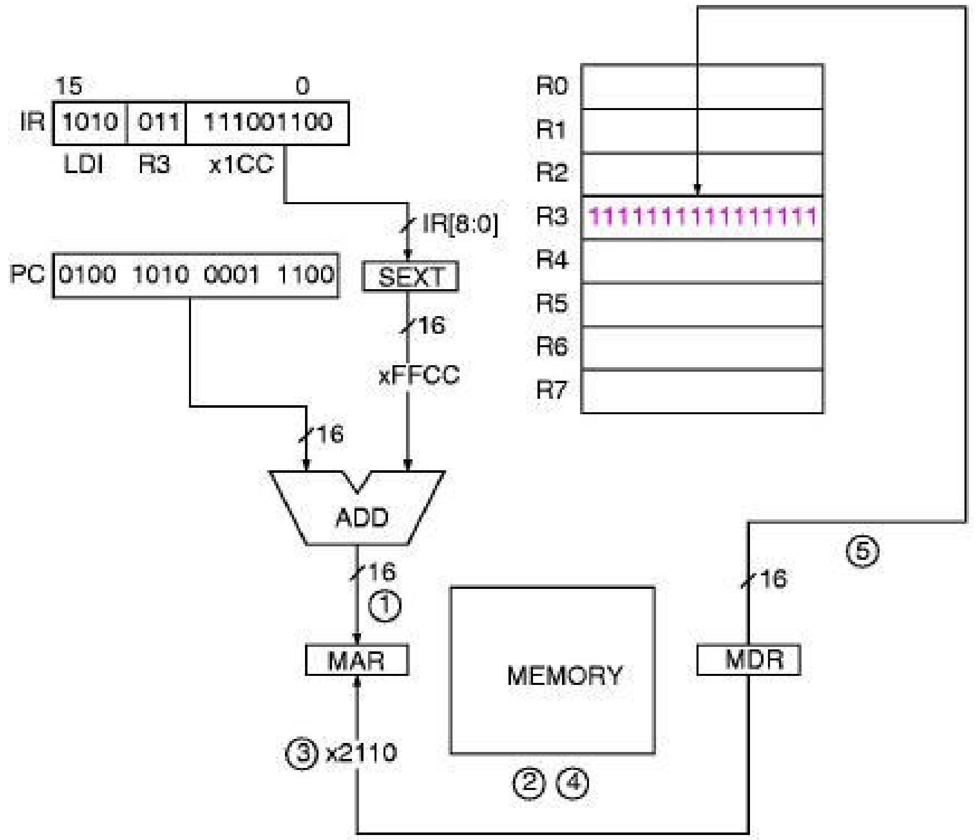
Examples

```
LDI  R2, POINTER          ; R2 <= Mem[Mem[POINTER]]
```

Notes: The LABEL must be within +256/-255 lines of the instruction.
Condition codes are set.

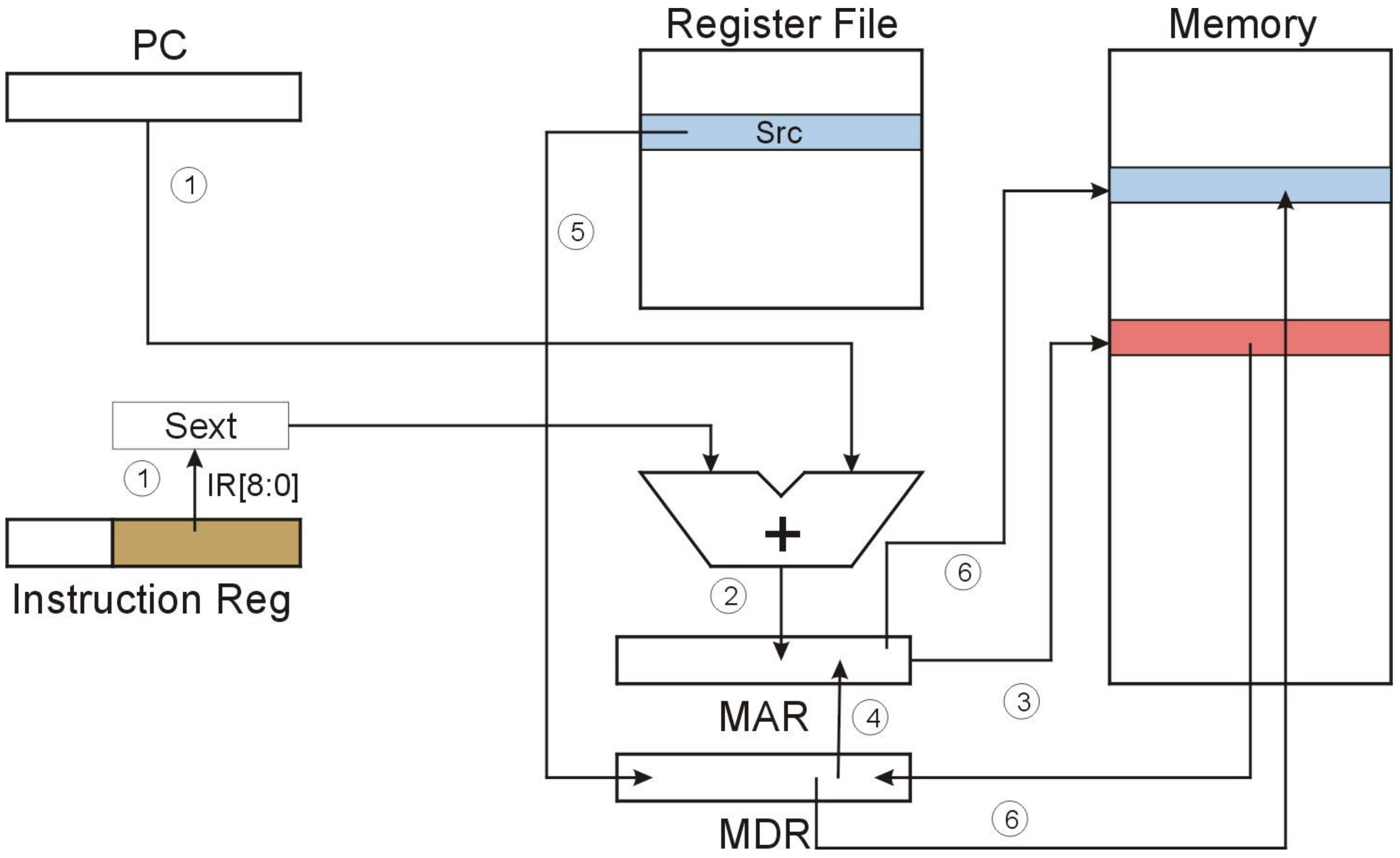
LDI data path

LDI R3, x1CC



STI (Indirect)

	15	14	13	12	11 10 9	8 7 6 5 4 3 2 1 0
STI	1	0	1	1	Src	PCoffset9



STI: Store Indirect

Assembler Inst.

```
STI  SR, LABEL    ;  Mem[Mem[LABEL]] <= SR
```

Encoding

```
0011  SR  offset9
```

Examples

```
STI  R2, POINTER  ; Mem[Mem[POINTER]] <= R2
```

Notes: The LABEL must be within +/- 256 lines of the instruction.
Condition codes are NOT set.

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:

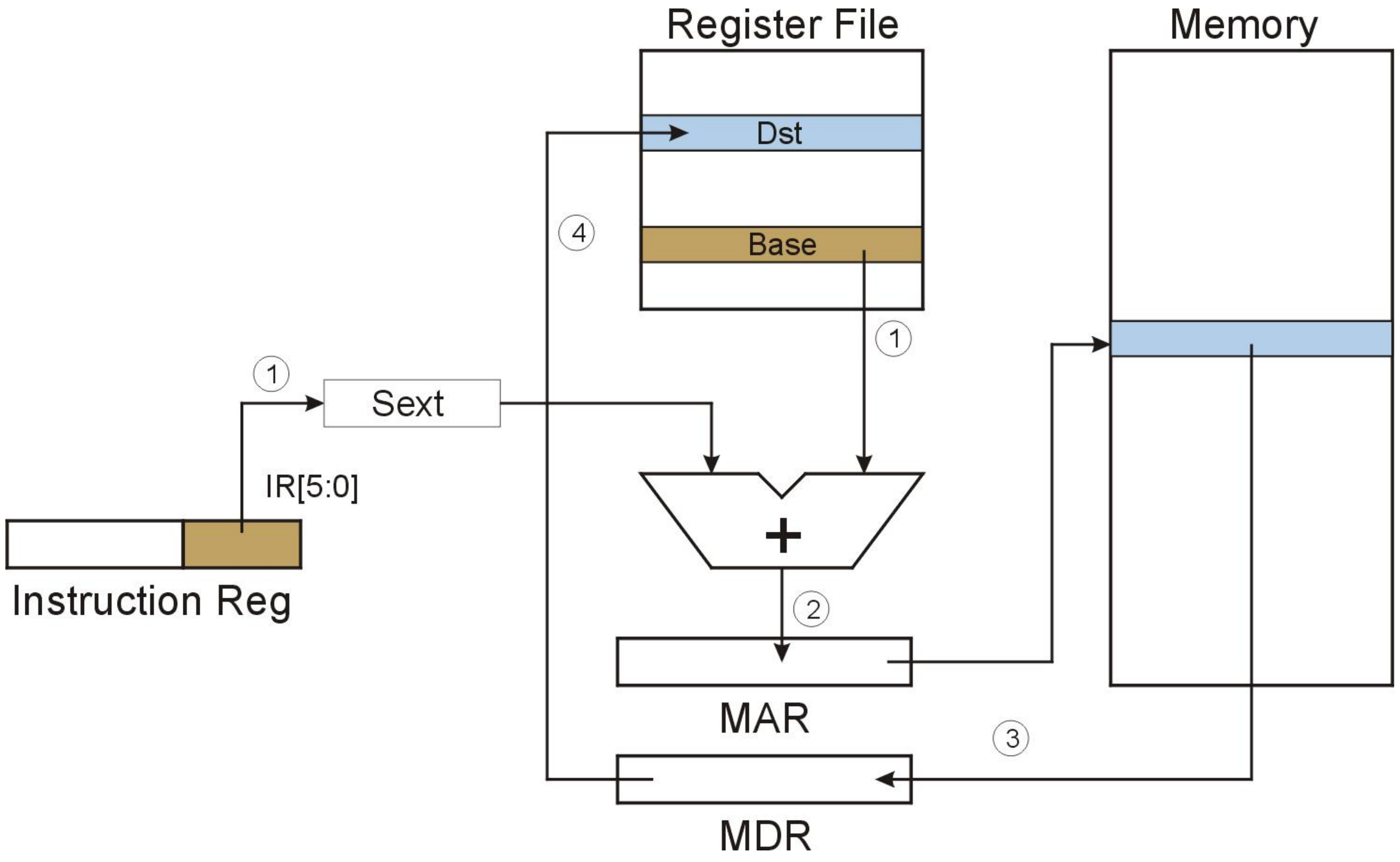
- **Use a register to generate a full 16-bit address.**

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)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDR	0	1	1	0	Dst			Base			offset6					



LDR: Load Base+Index

Assembler Inst.

```
LDR DR, BaseR, offset        ; DR <= Mem[ BaseR+SEXT( IR[5:0] )]
```

Encoding

```
0110  DR  BaseR  offset6
```

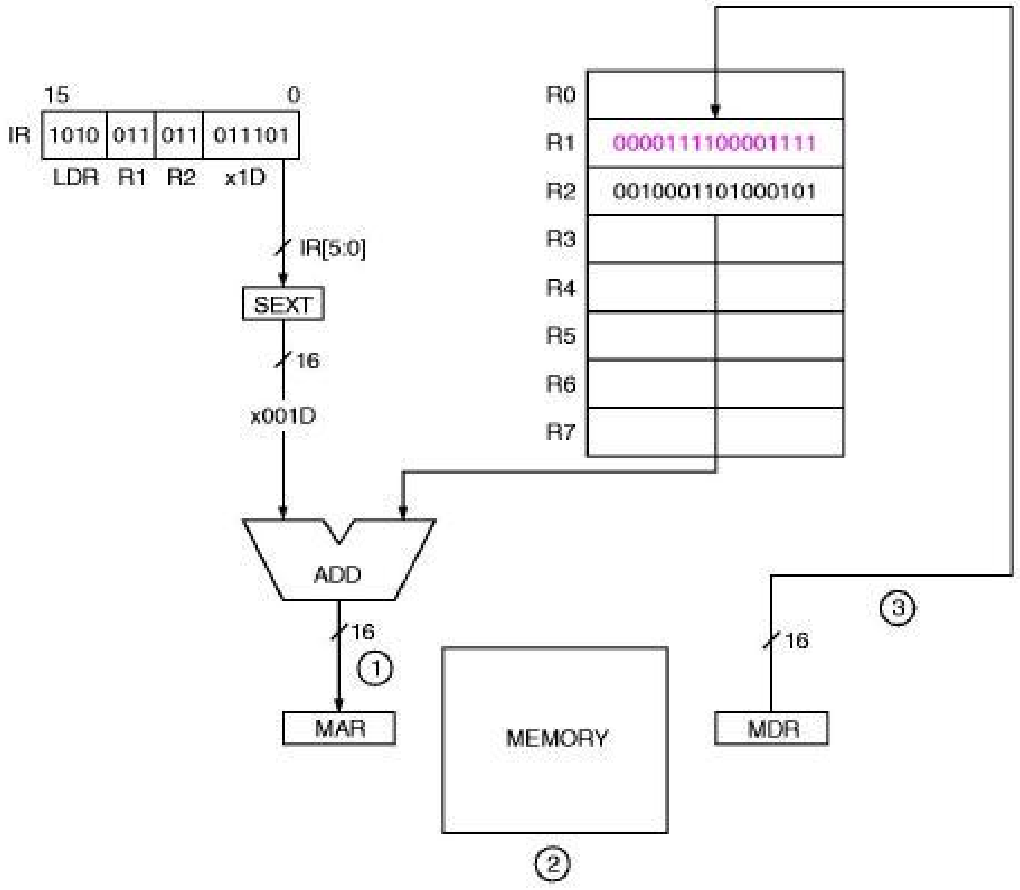
Examples

```
LD   R2, R3, #15             ; R2 <= Mem[(R3)+15]
```

**Notes: The 6 bit offset is a 2's complement number, so range is -32 to +31.
Condition codes are set.**

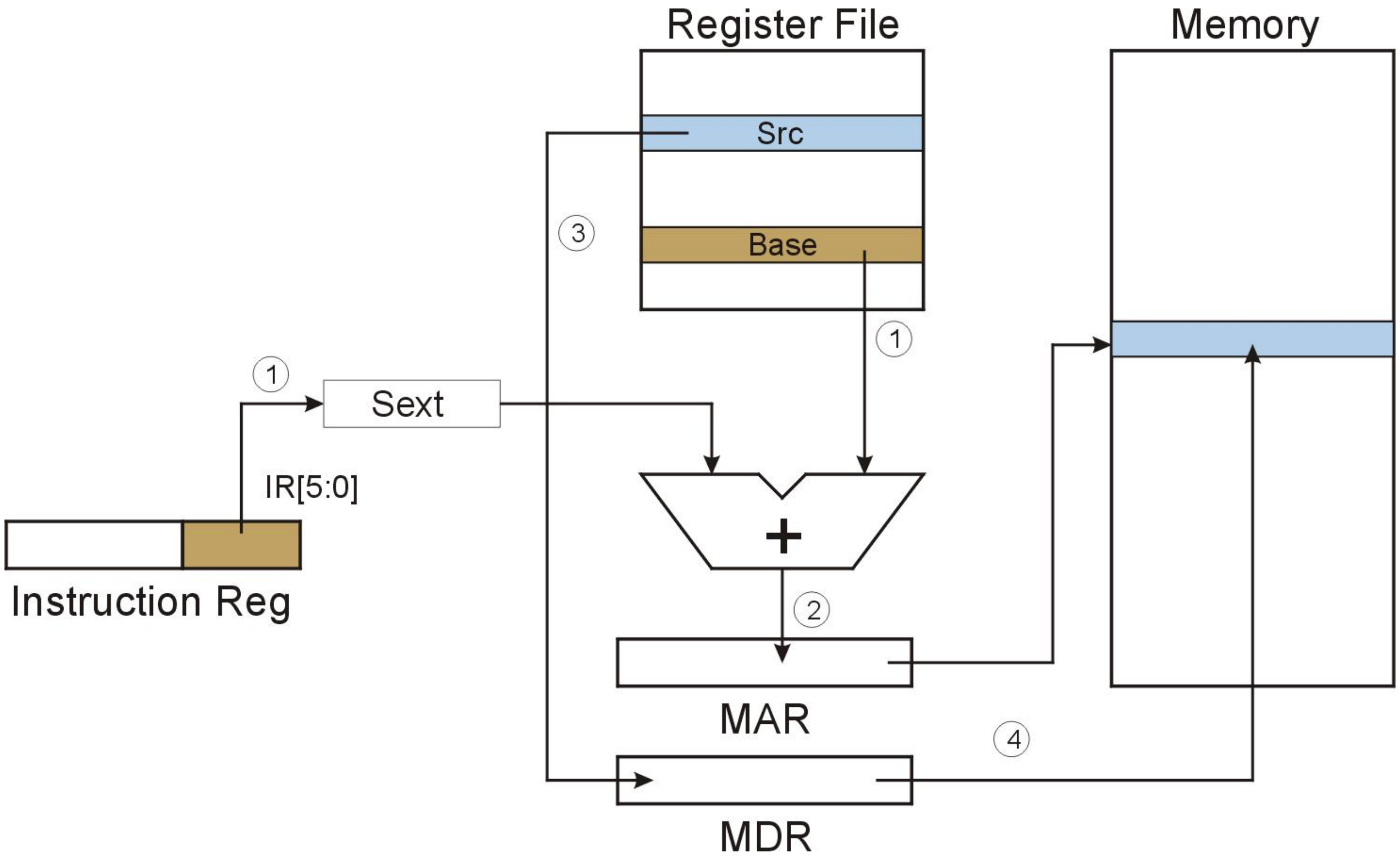
LDR data path

LDR R1, R2, x1D



STR (Base+Offset)

	15	14	13	12	11 10 9	8 7 6	5 4 3 2 1 0
STR	0	1	1	1	Src	Base	offset6



STR: Store Base+Index

Assembler Inst.

```
STR SR, BaseR, offset6 ;     Mem[BaseR+SEXT(offset6)] <= (SR)
```

Encoding

```
0111  SR  BaseR offset6
```

Examples

```
STR   R2, R4, #15   ;     Mem[R4+15] <= (R2)
```

Notes: The offset is sign-extended to 16 bits.
Condition codes are not set.

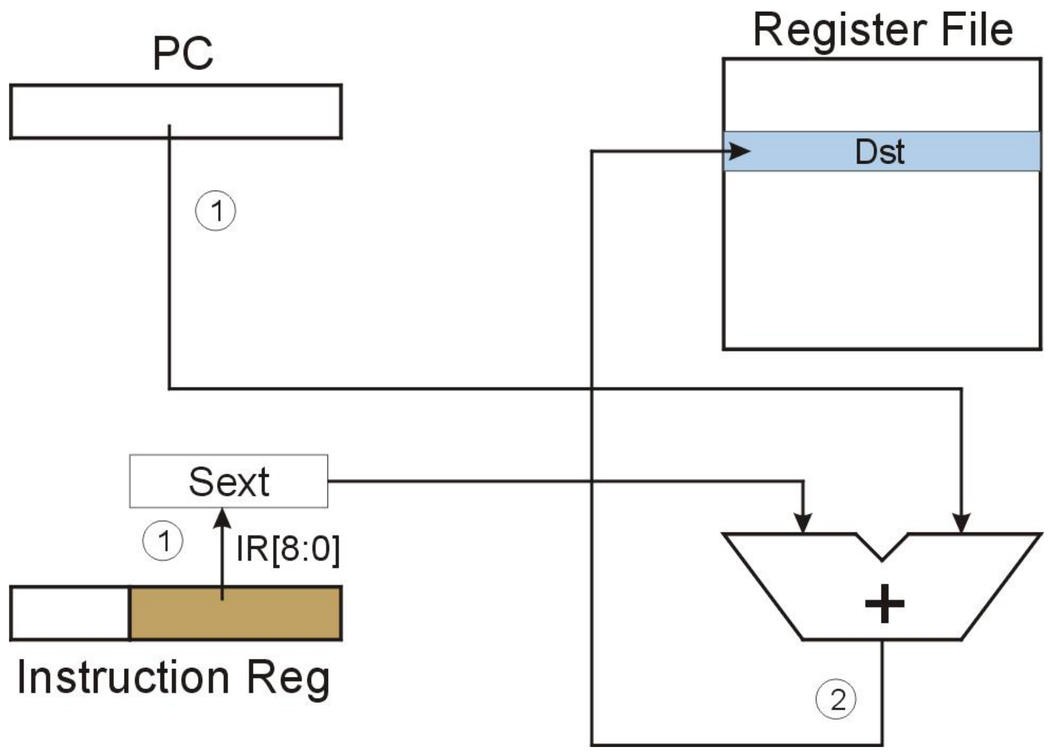
Load Effective Address

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)

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



LEA: Load Effective Address

Assembler Inst.

```
LEA  DR, LABEL       ; DR <= LABEL
```

Encoding

1110 DR offset9 (i.e. address of LABEL = (PC) + SEXT(offset9)

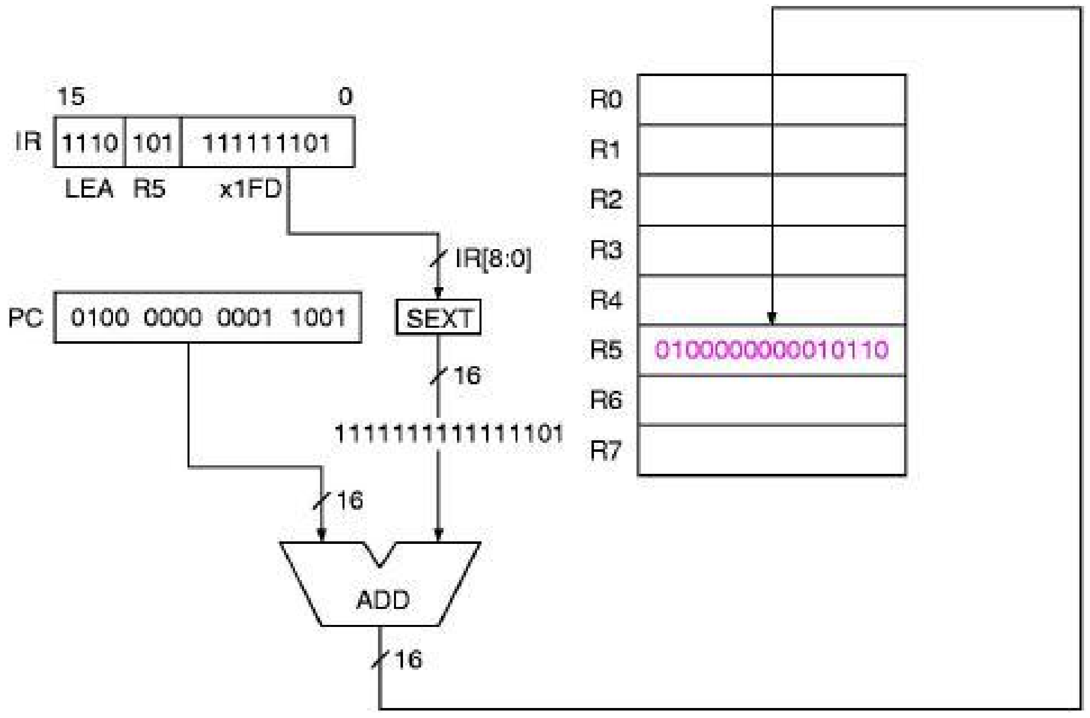
Examples

```
LEA   R2, DATA ; R2 gets the address of DATA
```

Notes: The LABEL must be within +/- 256 lines of the instruction.
Condition codes are set.

LEA data path

LEA R5, # -3



Example

PC = x30F6

Address	*Instruction*	*Comments*
x30F6	1 1 1 0 0 0 1 1 1 1 1 1 1 1 0 1	R1 ← PC – 3 = x30F4
x30F7	0 0 0 1 0 1 0 0 0 1 1 0 1 1 1 0	R2 ← R1 + 14 = x3102
x30F8	0 0 1 1 0 1 0 1 1 1 1 1 1 0 1 1	M[PC - 5] ← R2 M[x30F4] ← x3102
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 = 5
x30FB	0 1 1 1 0 1 0 0 0 1 0 0 1 1 1 0	M[R1+14] ← R2 M[x3102] ← 5

x30FC	1 0 1 0 0 1 1 1 1 1 1 1 0 1 1 1	R3 ← M[M[x30F4]]
		R3 ← M[x3102]
		R3 ← 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.

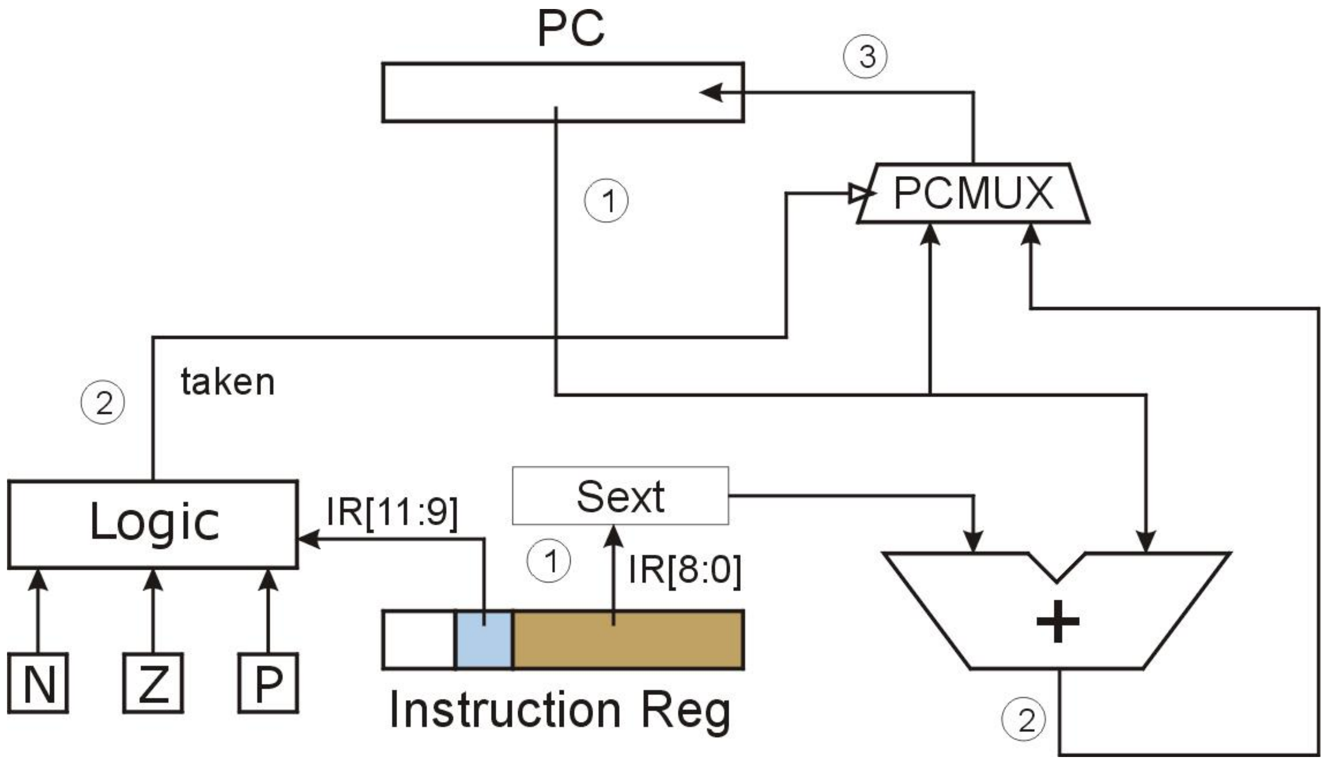
If the set bit is specified, the branch is taken.

- **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)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR	0	0	0	0	n	z	p	PCoffset9								



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

BR: Conditional Branch

Assembler Inst.

BRx LABEL

where x = n, z, p, nz, np, zp, or nzp

Branch to LABEL if the selected condition code are set

Encoding

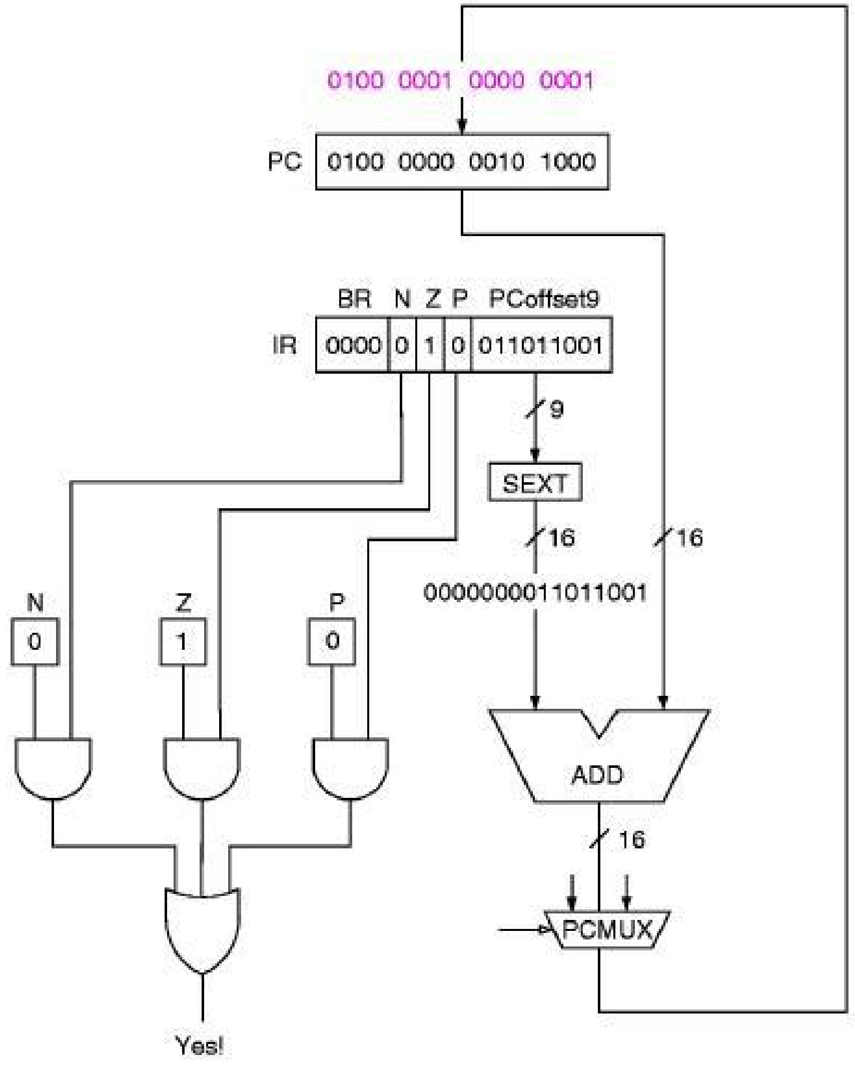
0000 n z p PCoffset9

Examples

BRzp LOOP ; branch to LOOP if previous op returned zero or positive.

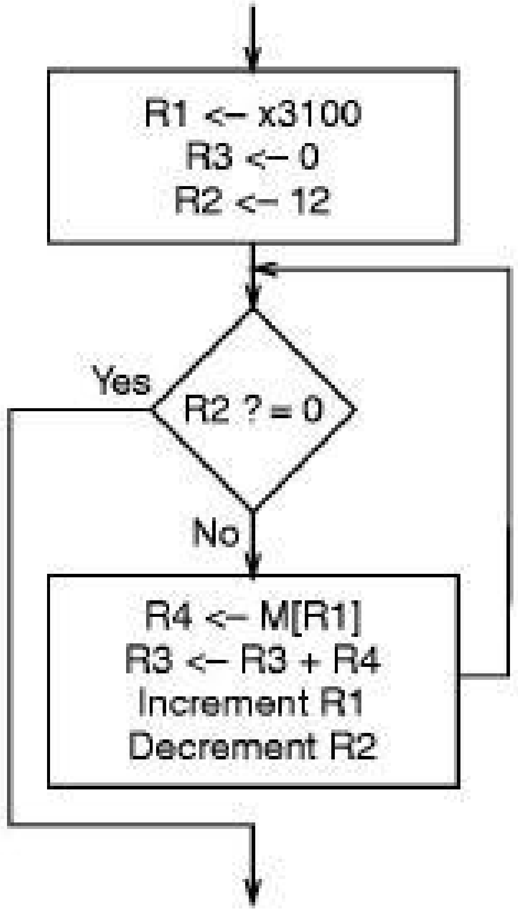
BR data path

BRz x0D9

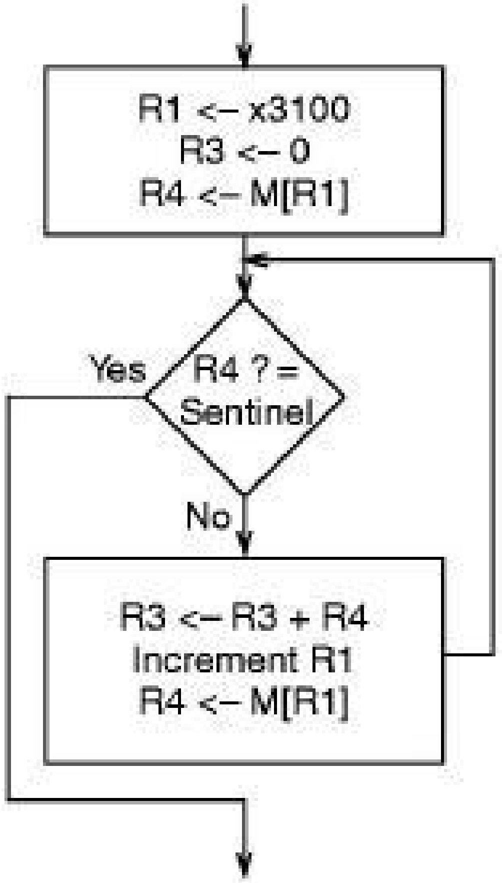


Building loops using BR

Counter control



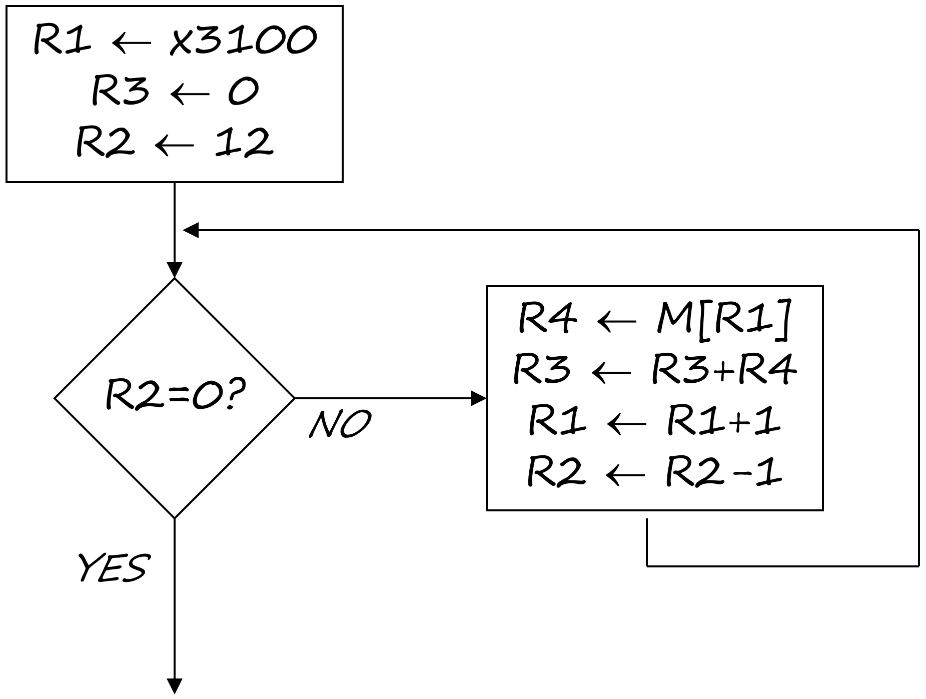
Sentinel control



Using Branch Instructions

Compute sum of 12 integers.

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



Sample Program

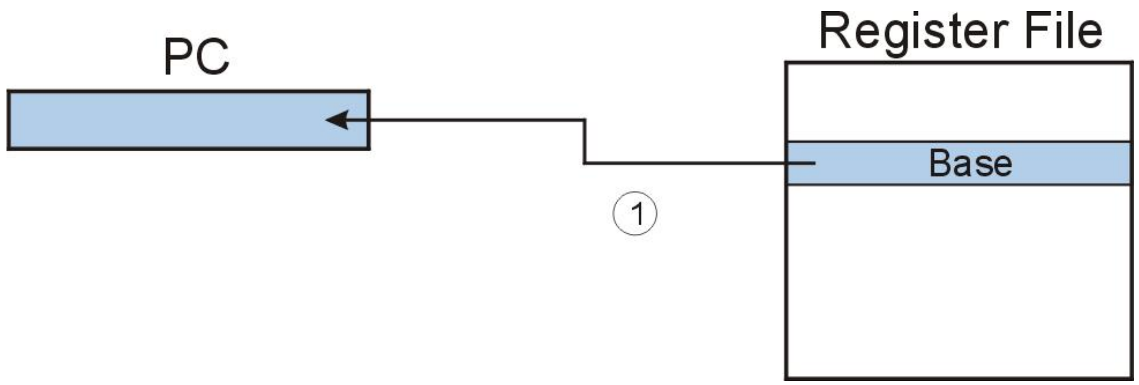
Address	Instruction	Comments
x3000	1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1	R1 ← x3100 (PC+0xFF)
x3001	0 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0	R3 ← 0
x3002	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	R2 ← 0
x3003	0 0 0 1 0 1 0 0 1 0 1 0 1 1 0 0	R2 ← 12
x3004	0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1	If Z, goto x300A (PC+5)
x3005	0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0	Load next value to R4
x3006	0 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0	Add to R3
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	Increment R1 (pointer)
X3008	0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1	Decrement R2 (counter)
x3009	0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 0	Goto x3004 (PC-6)

JMP (Register)

Jump is an unconditional branch -- *always* taken.

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

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



JMP: Jump or Go To

Assembler Inst.

JMP BaseR

Take the next instruction from the address stored in BaseR

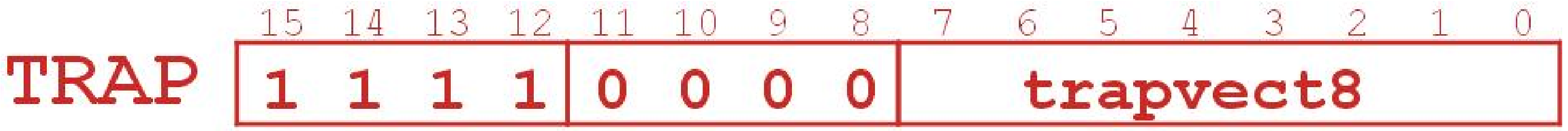
Encoding

1100 000 BaseR 00 0000

Example

JMP R5   ; if (R5) = x3500, the address x3500 is written to the PC

TRAP



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

vector	*symbol*	*routine*
x20	GETC	read a single character (no echo)
x21	OUT	output a character to the monitor
x22	PUTS	write a string to the console
x23	IN	print prompt to console, read and echo character from keyboard
X23	PUTSP	write a string to the console; two chars per memory location
x25	HALT	halt the program
x26		write a number to the console (undocumented)

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

TRAP: Invoke a system routine

Assembler Inst.

TRAP trapvec

Encoding

1111 0000 trapvect8

Examples

TRAP x23

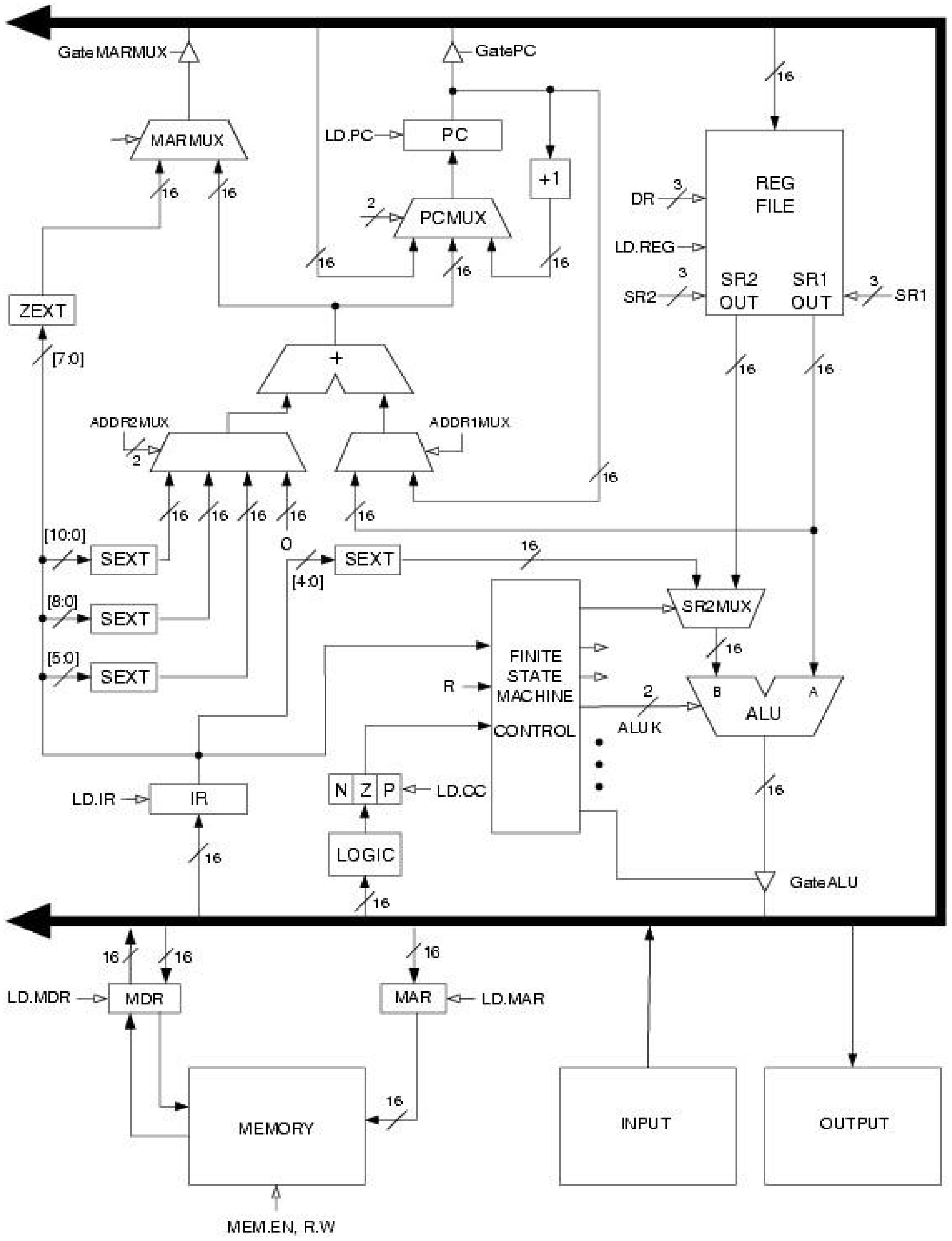
Note: **R7 <= (PC) (for eventual return)**

PC <= Mem[Zext(trapvect8)]

JSR, JSRR and RET for Subroutines

- **Mentioned in Chapter 3: TRAP and Subroutines**

LC-3 Data Path Revisited



Filled arrow
= info to be processed.

Unfilled arrow
= control signal.

Please explain the way instructions run!

1. **ADD**

2. **LDR**

3. **JMP**

The others? Home

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 N, Z, P are 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.**

LC 3 Assembly Language

Human-Readable Machine Language

Computers like ones and zeros…

```
0001110010000110
```

Humans like symbols…

```
ADD   R6,R2,R6   ; increment index reg.
```

Assembler is a program that turns symbols into machine instructions.

- ISA-specific:
  close correspondence between symbols and instruction set
  - mnemonics for opcodes
  - labels for memory locations
- additional operations for allocating storage and initializing data

An Assembly Language Program

```
;
; Program to multiply a number by the constant 6
;
       .ORIG  x3050
       LD     R1, SIX
       LD     R2, NUMBER
       AND    R3, R3, #0     ; Clear R3.  It will
                             ; contain the product.
; The inner loop
;
AGAIN  ADD    R3, R3, R2
       ADD    R1, R1, #-1    ; R1 keeps track of
       BRp    AGAIN          ; the iteration.
;
       HALT
;
NUMBER .BLKW  1
SIX    .FILL  x0006
;
       .END
```

LC-3 Assembly Language Syntax

Each line of a program is one of the following:

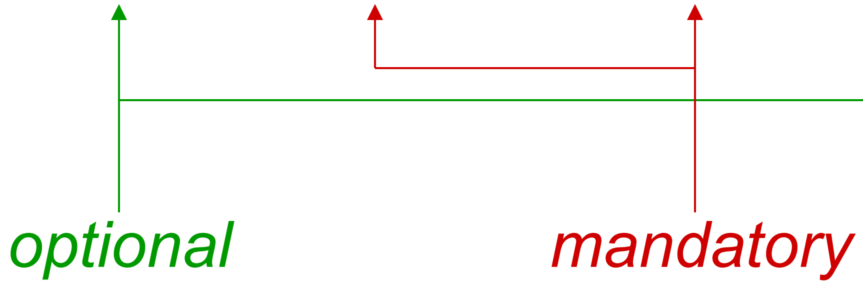
- **an instruction**
- **an assember directive (or pseudo-op)**
- **a comment**

Whitespace (between symbols) and case are ignored.

Comments (beginning with “;”) are also ignored.

An instruction has the following format:

```
LABEL OPCODE OPERANDS ; COMMENTS
```



Opcodes and Operands

Opcodes

- reserved symbols that correspond to LC-3 instructions
- listed in Appendix A
  - ex: `ADD`, `AND`, `LD`, `LDR`, …

Operands

- registers -- specified by Rn, where n is the register number
- numbers -- indicated by # (decimal) or x (hex)
- label -- symbolic name of memory location
- separated by comma
- number, order, and type correspond to instruction format
  - ex:

```
ADD R1,R1,R3
ADD R1,R1,#3
LD  R6,NUMBER
BRz LOOP
```

Labels and Comments

Label

- placed at the beginning of the line
- assigns a symbolic name to the address corresponding to line
  - ex:

```
LOOP  ADD R1,R1,#-1
      BRp LOOP
```

Comment

- anything after a semicolon is a comment
- ignored by assembler
- used by humans to document/understand programs
- tips for useful comments:
  - avoid restating the obvious, as “decrement R1”
  - provide additional insight, as in “accumulate product in R6”
  - use comments to separate pieces of program

Assembler Directives

Pseudo-operations

- do not refer to operations executed by program
- used by assembler
- look like instruction, but "opcode" starts with dot

Opcode	*Operand*	*Meaning*
`.ORIG`	address	starting address of program
`.END`		end of program
`.BLKW`	n	allocate n words of storage
`.FILL`	n	allocate one word, initialize with value n
`.STRINGZ`	n-character string	allocate n+1 locations, initialize w/characters in "…" and null terminator

Trap Codes

LC-3 assembler provides “pseudo-instructions” for each trap code, so you don’t have to remember them.

Code	*Equivalent*	*Description*
`HALT`	`TRAP x25`	Halt execution and print message to console.
`IN`	`TRAP x23`	Print prompt on console “Input a character>”, read (and echo) one character from keybd. Character stored in R0[7:0].
`OUT`	`TRAP x21`	Write one character (in R0[7:0]) to console.
`GETC`	`TRAP x20`	Read one character from keyboard without prompt. Character stored in R0[7:0].
`PUTS`	`TRAP x22`	Write null-terminated string to console. Address of string is in R0.

Example: print out a prompt

```
LEA   R0, P_out          ; Output P_out
PUTS                     ;  Trap x22
   ……
P_out    .STRINGZ  "New character is “
```

Style Guidelines

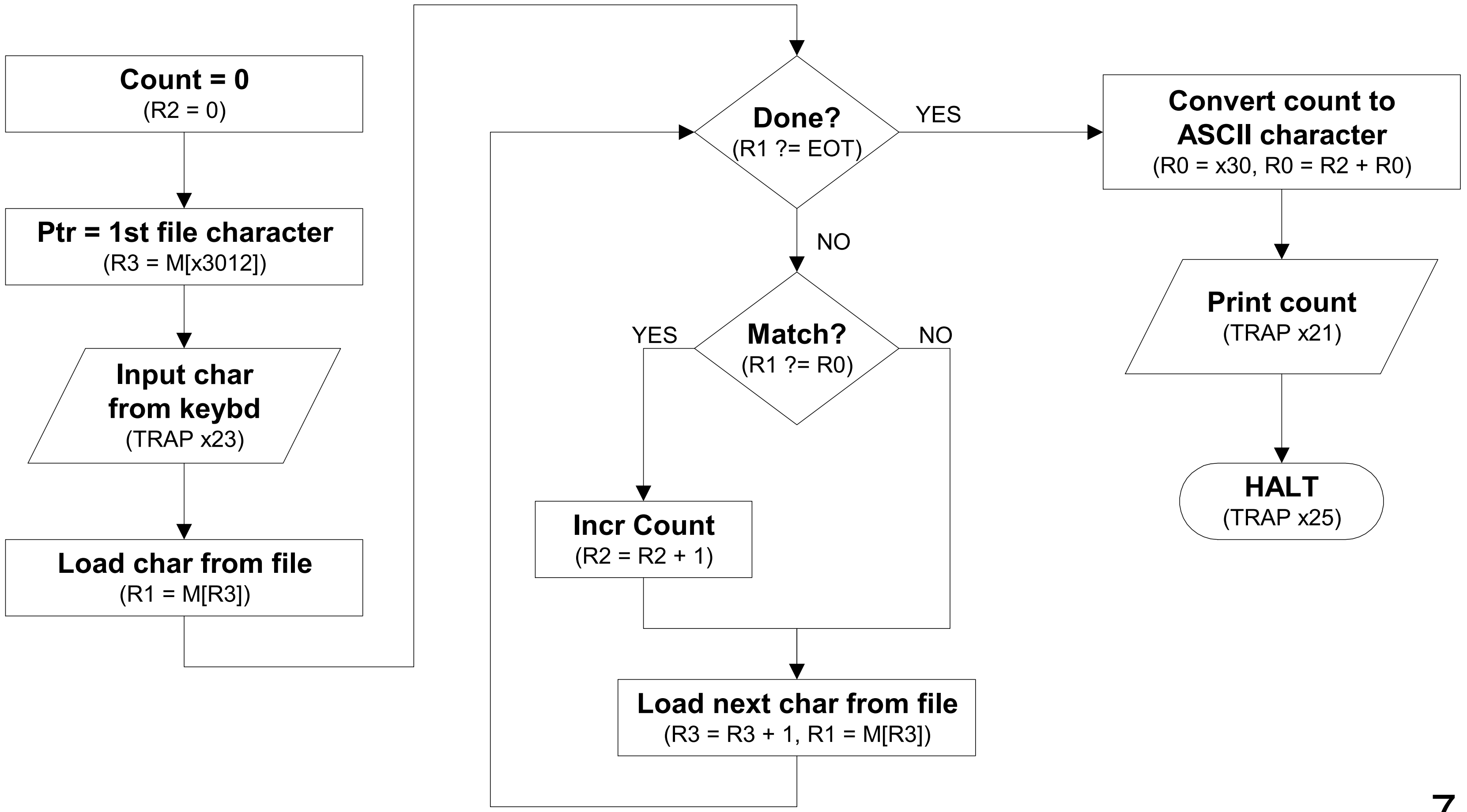
Use the following style guidelines to improve the readability and understandability of your programs:

1. **Provide a program header, with author's name, date, etc., and purpose of program.**
2. **Start labels, opcode, operands, and comments in same column for each line.** (Unless entire line is a comment.)
3. **Use comments to explain what each register does.**
4. **Give explanatory comment for most instructions.**
5. **Use meaningful symbolic names.**
   - **Mixed upper and lower case for readability.**
   - ASCIItoBinary, InputRoutine, SaveR1
6. **Provide comments between program sections.**
7. **Each line must fit on the page -- no wraparound or truncations.**
   - **Long statements split in aesthetically pleasing manner.**

Sample Program

Count the occurrences of a character in a file.

Remember this?



Char Count in Assembly Language (1 of 3)

```
;
; Program to count occurrences of a character in a file.
; Character to be input from the keyboard.
; Result to be displayed on the monitor.
; Program only works if no more than 9 occurrences are found.
;
;
; Initialization
;
        .ORIG   x3000
        AND     R2, R2, #0      ; R2 is counter, initially 0
        LD      R3, PTR         ; R3 is pointer to characters
        GETC                    ; R0 gets character input
        LDR     R1, R3, #0      ; R1 gets first character
;
; Test character for end of file
;
TEST    ADD     R4, R1, #-4     ; Test for EOT (ASCII x04)
        BRz     OUTPUT          ; If done, prepare the output
```

Char Count in Assembly Language (2 of 3)

```
;
; Test character for match.  If a match, increment count.
;
        NOT     R1, R1
        ADD     R1, R1, R0  ; If match, R1 = xFFFF
        NOT     R1, R1      ; If match, R1 = x0000
        BRnp    GETCHAR     ; If no match, do not increment
        ADD     R2, R2, #1
;
; Get next character from file.
;
GETCHAR ADD     R3, R3, #1  ; Point to next character.
        LDR     R1, R3, #0  ; R1 gets next char to test
        BRnzp   TEST
;
; Output the count.
;
OUTPUT  LD      R0, ASCII   ; Load the ASCII template
        ADD     R0, R0, R2  ; Covert binary count to ASCII
        OUT                 ; ASCII code in R0 is displayed.
        HALT                ; Halt machine
```

Char Count in Assembly Language (3 of 3)

```
;
; Storage for pointer and ASCII template
;
ASCII    .FILL   x0030
PTR      .FILL   x4000
         .END
```

Homework

1. Input two digits in decimal system from 0-9. Compute their product, then print it on screen.

2. Input two digits in decimal system from 0-9. Do integer division to get their quotient and remainder. Print them out on screen.

3. Input two positive integers. Compute their product, then print it on screen.

Homework

4. Input two integers from 0-9. Compute their sum, difference, product, and their quotient and remainder. Print them out on screen.

5. Input two arbitrary integers. Compute their product, then print it on screen.

And more …

In the text book

7.1 – 7.25