Basic LC-3 Data Movement

Memory Access

- Programs need to read and write **memory**
  - Memory provides more storage space than the registers
  - But we need to copy information to registers in order to process it using the Operate instructions
- If we want to read or write memory, we need to indicate **where** in memory (the location)

Using a Label as a Location

- Similar to a variable in high-level languages
- Assembler directives reserve space in memory and give them names that we can use to refer to addresses in our assembly-language program
  - The assembler converts these labels to the addresses the processor needs (*...actually, a way to calculate them...*)

LC-3 Example:

```
NUM     .FILL x42
```

reserve one space in memory named NUM and initialize it to the hexadecimal value 42_{16}

Using Labels to Access Memory

- A **load** instruction copies a value from a memory location to a register

LC-3 Example:

"LoaD" Destination Source

```
LD   R0, NUM
```

copy whatever is in the memory location named NUM to register R0

- A **store** instruction copies a value from a register to a memory location

LC-3 Example:

"STore" Source Destination

```
ST   R6, NUM
```

copy whatever is in register R6 to the memory location named NUM

Simple Program Example

inc_mem.asm

```
; This program reads a value from mem,
; increments it, and writes it to mem
        .ORIG x0200
START
        LD   R0, NUM        ; memory read
        ADD  R6, R0, #1
        ST   R6, NUM        ; memory write
        BR   START          ; repeat forever

; inc_mem data
NUM     .FILL x0042
        .END
```

Actual Data

R0 = 0x0043
R6 = 0x0044

NUM = 0x0044

These are not good comments for a **real** program!

Comments should only provide information that is not already obvious from the instruction itself.

(NUM)

① Tell the computer to start at x0042 memory location

② Load the value from NUM to register R0

③ Add R0 and #1, store in R6

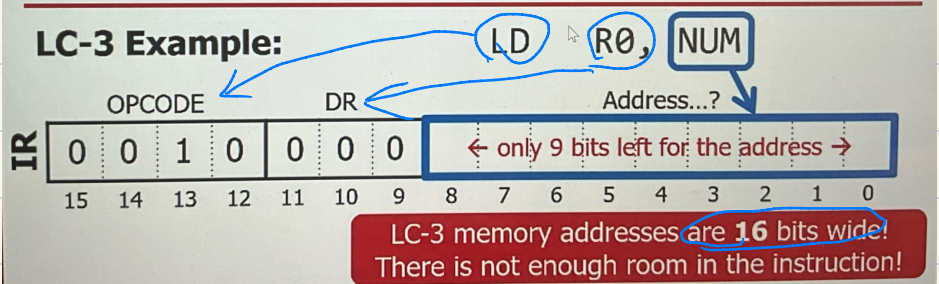
④ Store register R6's value back to NUM

⑤ Repeat the program forever

How Is This Implemented?

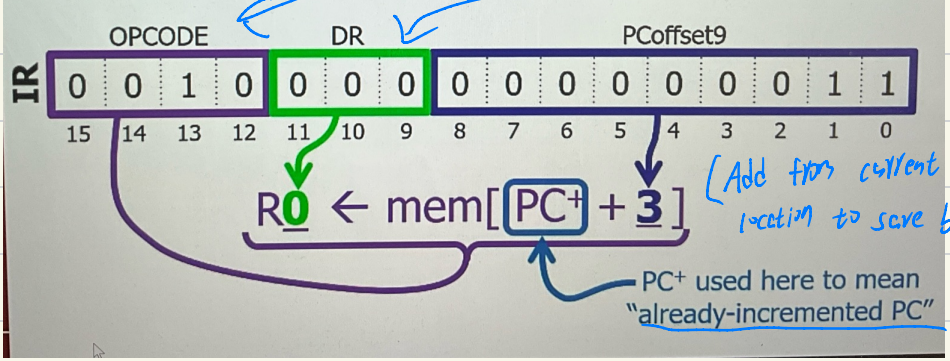
- Problem: need to encode the location into the instruction, but often do not have enough bits!
  - If the memory is large, then we need many bits to indicate which address to access
  - The instruction word is often too small to fit an address

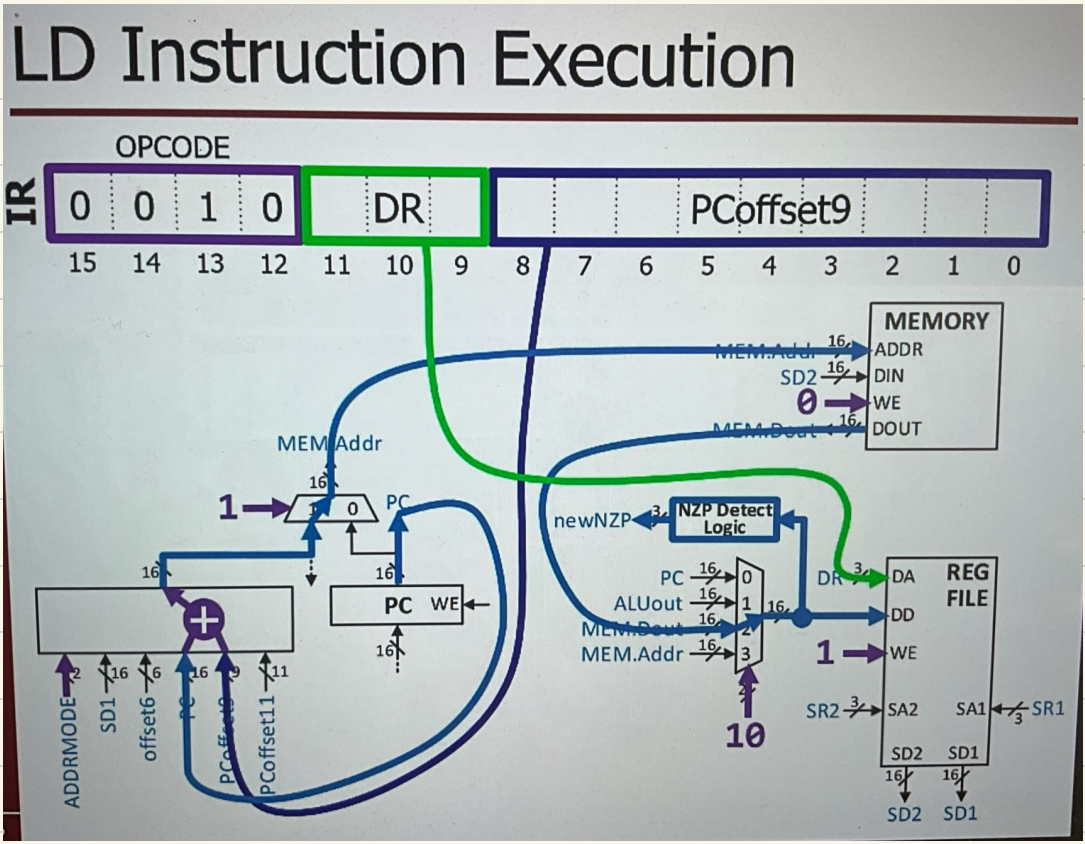
LC-3 Example: LD R0, NUM



- Idea: instead of encoding the actual **address**, encode **how far away it is** from where the Program Counter is pointing
  - This is called a **PC-Relative** addressing mode

LC-3 Example: LD R0, NUM



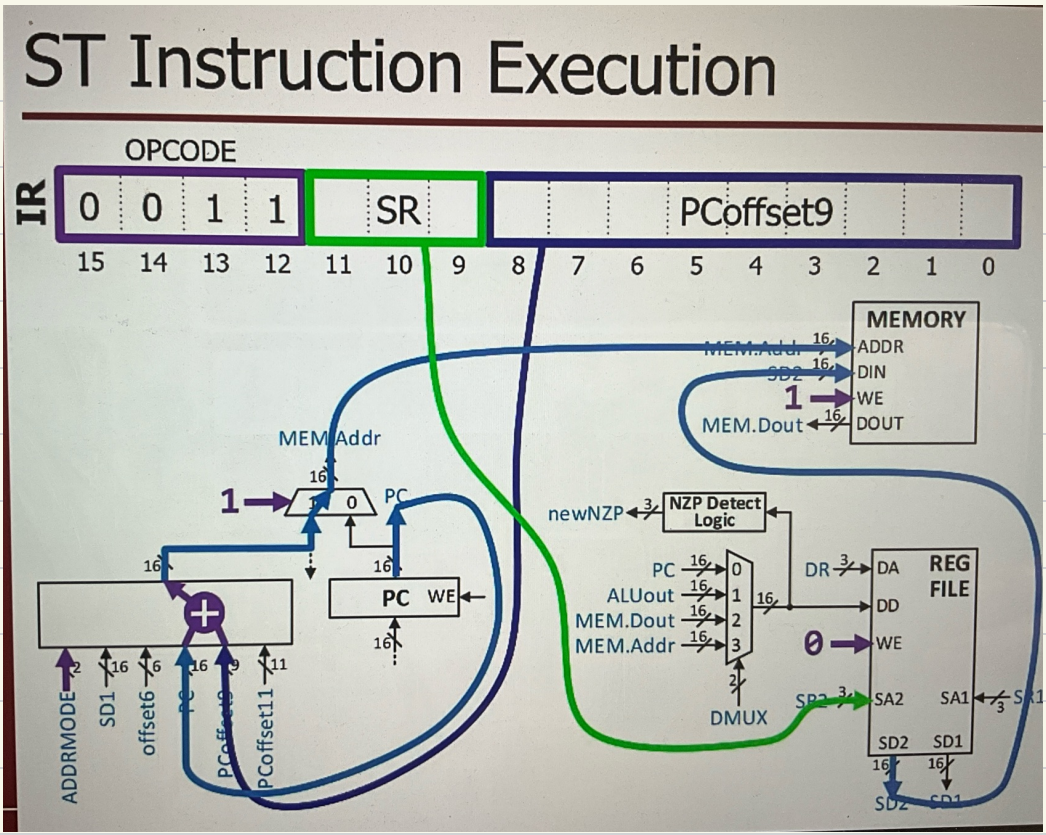


① Compute the effective address by adding PC and offset

② Load operation does not enable write in memory

③ value read from memory is sent to the register file → written to destination register

④ Condition codes updated based on the value read from memory



① Calculate the effective address same way as LD

② Copy value from source register to memory at the effective address

PC-Relative Load and Store

- These instructions are used for accessing memory "near" the current PC value
  - 9-bit 2's-complement offset means the address must be within [-256,+255] of the already-incremented PC
- Need to use different types of load/store instructions for memory addresses "further away"

More LC-3 Data Movement

- LDR and STR use **Base+Offset** addressing
  - The **base register** acts as a "pointer" into memory
  - The signed **offset** indicates how far away the effective address is from where the base register is pointing

Base+Offset Addressing

LC-3 Example:

"LoaD using BaseR + offset"

LDR DR, BaseR, offset6 (Destination)

LDR R5, R1, #2

read whatever is in memory two steps past the address in R1, and copy it to register R5

The syntax is given in the Programmer's Reference!

"STore using BaseR + offset"

STR SR, BaseR, offset6 (Source)

STR R5, R1, #2

copy whatever is in register R5 to memory two steps past the address in R1

Memory location

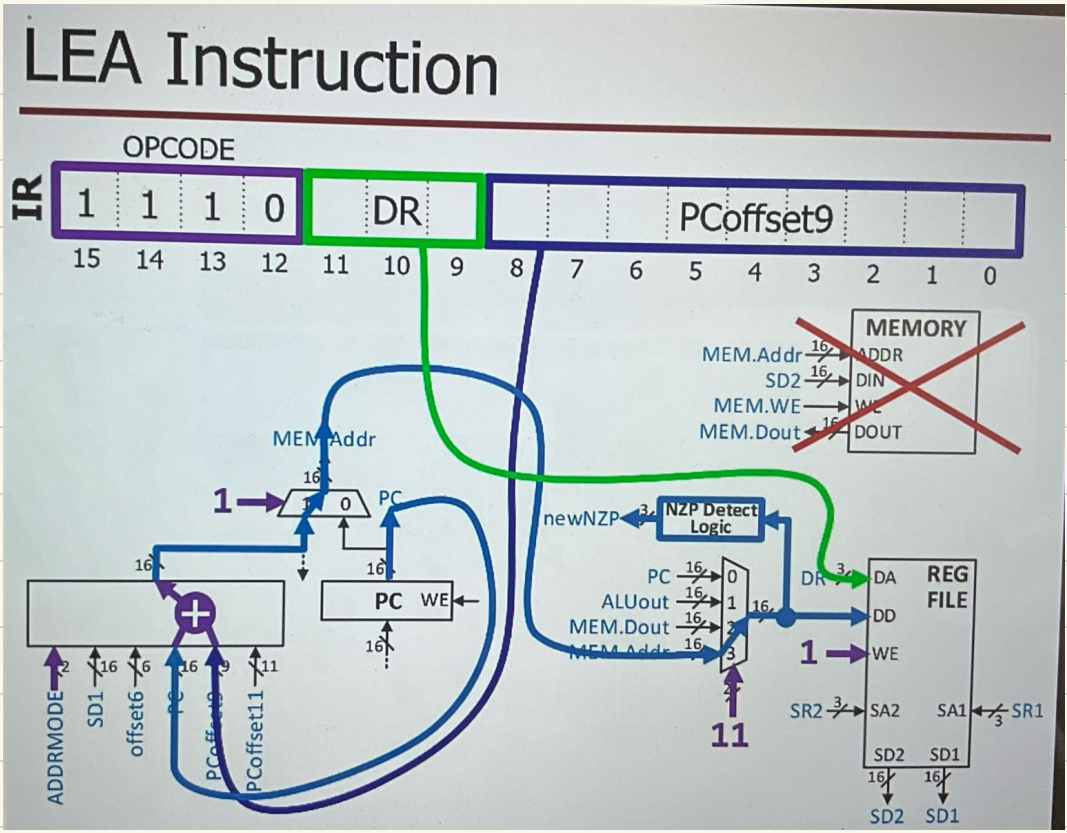
Getting the Base Address: LEA

- LDR and STR are only useful if we first put a useful address into a register!
- LEA copies a label's address to a register
  "**L**oad **E**ffective **A**ddress"
  - The **effective address** for LEA is calculated exactly the same way as it is for LD and ST (PC-Relative)

LC-3 Example:

```
LEA   R1, ARRAY
```

copy the address of the ARRAY label into register R1 so that we can use it as a base register for LDR/STR

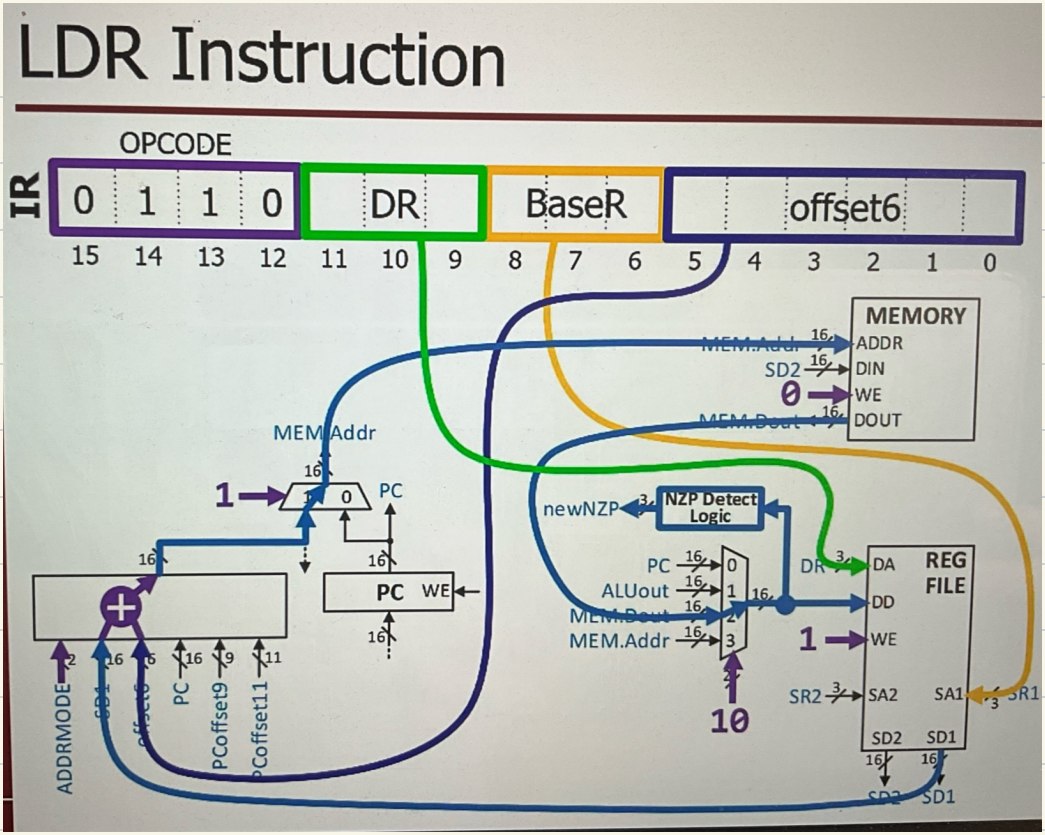


① OPcode → compute a PC-relative effective address

② The computed address is written to the destination register

③ The condition codes are updated

(Note: No memory involved at all)



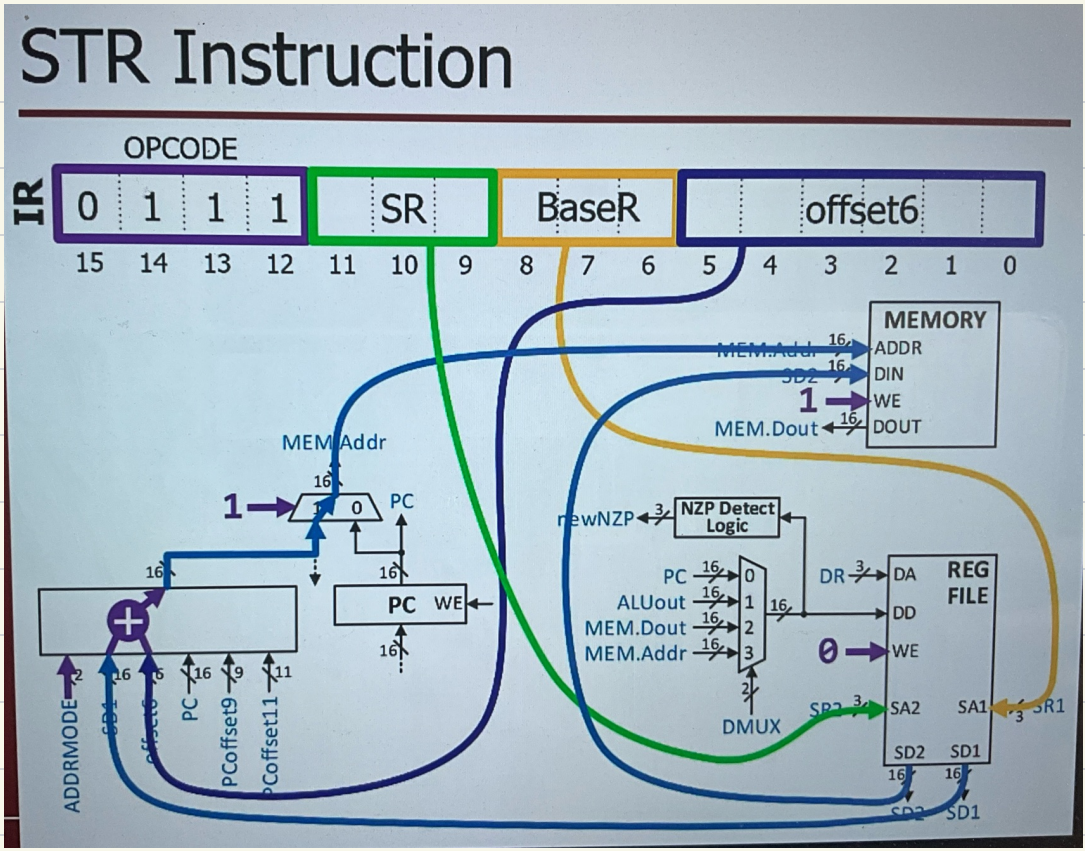
① Opcode → use the Base+offset mode

② Processor reads the base register's value and add to the offset

③ Sum for step 2 is the location of memory read

④ Value returned from memory is written to the destination register

⑤ Condition codes update



① compute effective address the same as LDR

② Read from the source register → write the value to memory

LC-3 Memory Indirect Addressing

Ways to Access Memory

- LD and ST allow us to access labelled memory locations "near" the instructions
  - The distance is limited by the size of the PC offset field
- LDR and STR are more flexible...
  - Can access locations around a "nearby" label
    - Use LEA to get a label address into the base register
    - Then use LDR/STR to access locations at/near base address
  - Can access any address
    - Use a .FILL directive to create a location holding an address
    - Use LD to load the address from memory
    - Then use LDR/STR to access locations at/near base address

...can also do this using LDI/STI...

Memory Indirect Addressing

LC-3 Example:

```
LD  R1, ADDR_R
LDR R4, R1, #0
```

R1 used as a pointer to the location for the load

"LoaD using Memory Indirect"

```
LDI R4, ADDR_R
```

read from the location labeled ADDR_R to get the effective address for a load that copies from memory to register R4

LC-3 Example:

```
LD  R2, ADDR_W
STR R4, R2, #0
```

R2 used as a pointer to the location for the store

"STore using Memory Indirect"

```
STI R4, ADDR_W
```

read from the location labeled ADDR_W to get the effective address for a store that copies from register R4 to memory

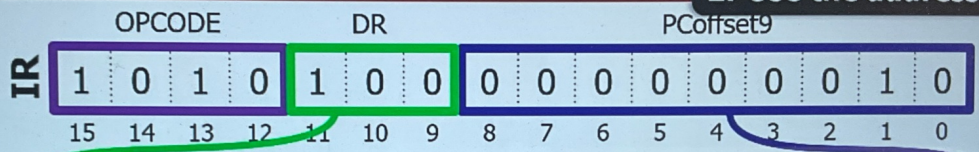
☆ Two steps:

① Get the address

② Use the address

LDI Execution

Two Steps:
1. Get the address
2. Use the address

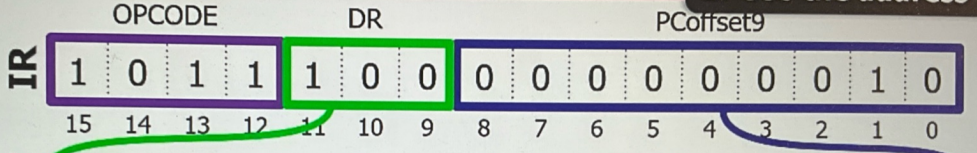


1. Compute PC-Relative address and read from memory
2. Use the value returned by memory as the **effective address** for the **load** operation (a second memory read) that copies from memory to the destination register and updates the condition codes

$$R4 \leftarrow mem[\ \underbrace{mem[PC^{+} + 2]}_{\text{address for load}}\ ]$$

STI Execution

Two Steps:
1. Get the address
2. Use the address



1. Compute PC-Relative address and read from memory
2. Use the value returned by memory as the **effective address** for a **store** operation (a memory write) that copies the source register value to memory

$$mem[\ \underbrace{mem[PC^{+} + 2]}_{\text{address for store}}\ ] \leftarrow R4$$