ISA in LC-3

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

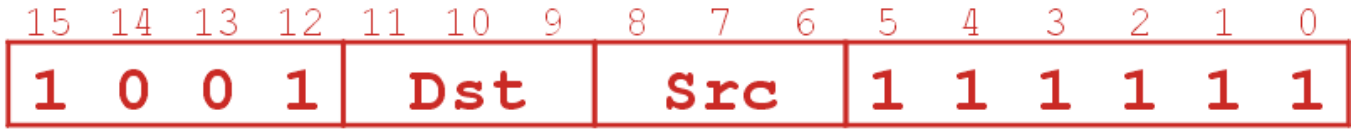
- Opcodes
  - 15 opcodes
  - *Operate* instructions: ADD, AND, NOT
  - *Data movement* instructions: LD, LDI, LDR, LEA, ST, STR, STI
  - *Control* instructions: BR, JSR/JSRR, JMP, RTI, TRAP
  - some opcodes set/clear *condition codes*, based on result:
    - N = negative, Z = zero, P = positive (> 0)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

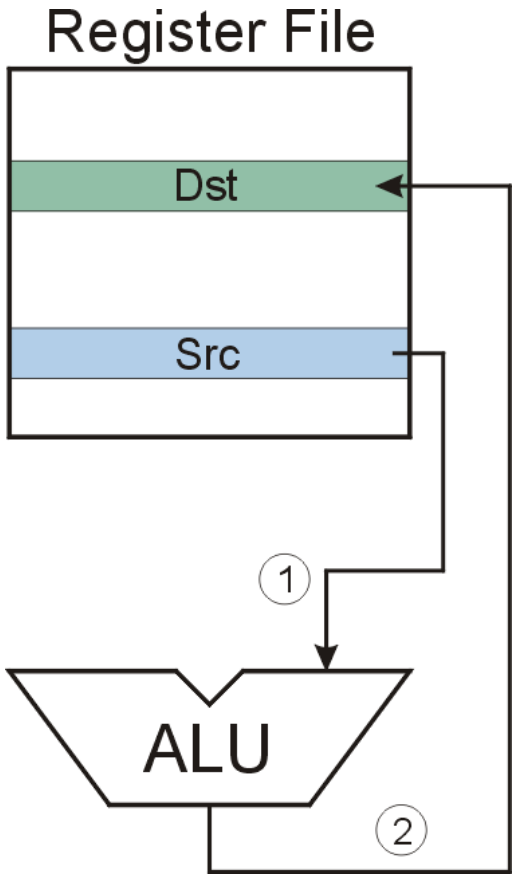
Operate Instructions

- 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)

NOT

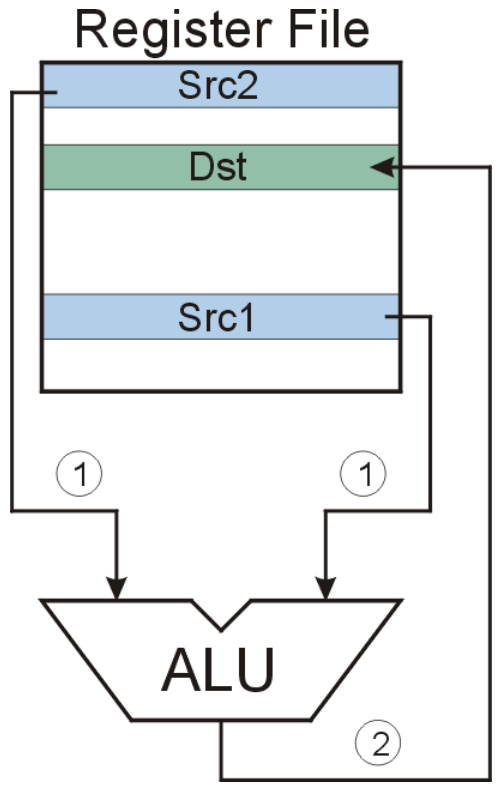




Note: Src and Dst could be the same register.

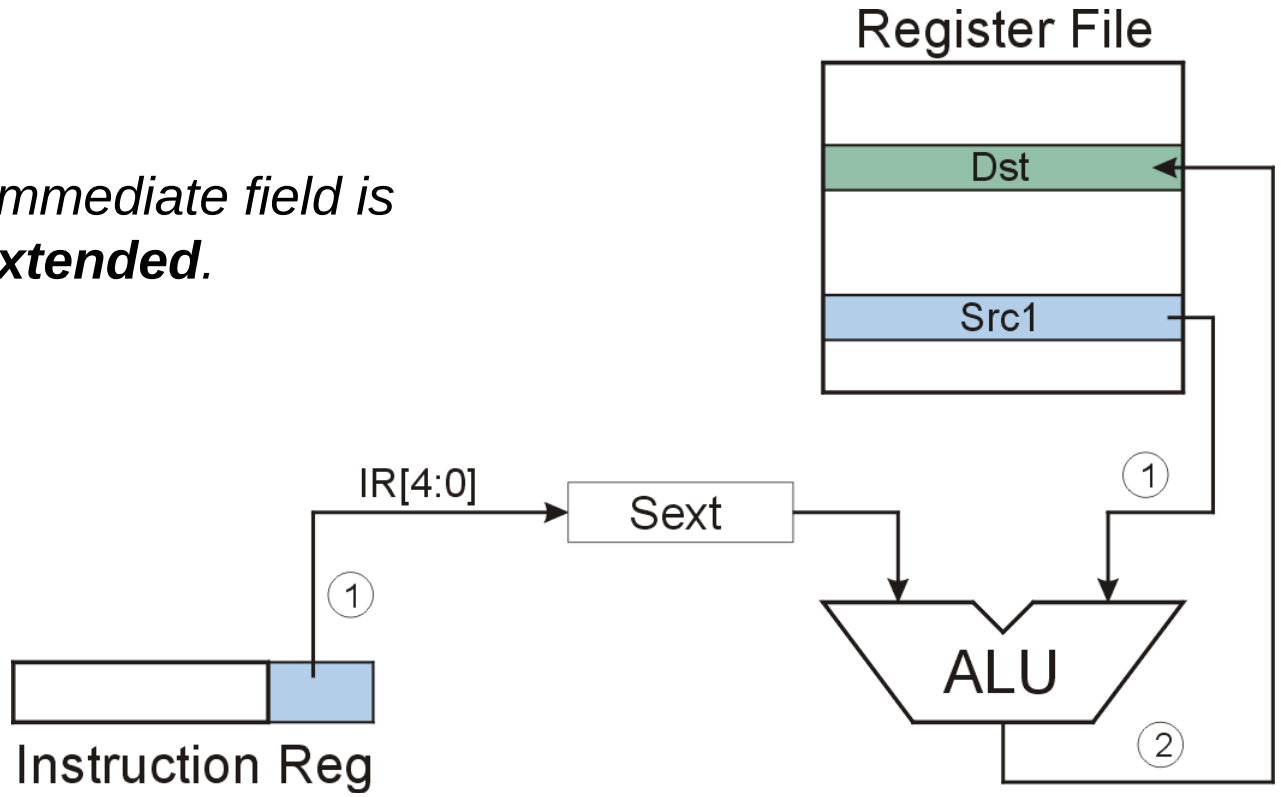
ADD/AND (Register)

this zero means "register mode"



ADD/AND (Immediate)

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



Sign extension

- if six bits are used to represent the number "00 1010" (decimal positive 10) and the sign extend operation increases the word length to 16 bits, then the new representation is simply "0000 0000 0000 1010". Thus, both the value and the fact that the value was positive are maintained.

- If ten bits are used to represent the value "11 1111 0001" (decimal negative 15) using two's complement, and this is sign extended to 16 bits, the new representation is "1111 1111 1111 0001". Thus, by padding the left side with ones, the negative sign and the value of the original number are maintained.

Twos compliment

- suppose we want to find how -28
- would be expressed in two's complement notation. First we write out 28 in binary form.

- 00011100

- Then we invert the digits. 0 becomes 1, 1 becomes 0.
- 11100011

- Then we add 1.
- 11100100

Using Operate Instructions

- With only ADD, AND, NOT…
  - How do we subtract?
  - 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.

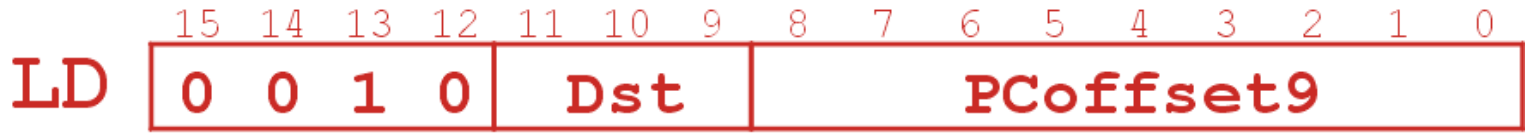
- 9 bits:
- Can form any address X, such that:

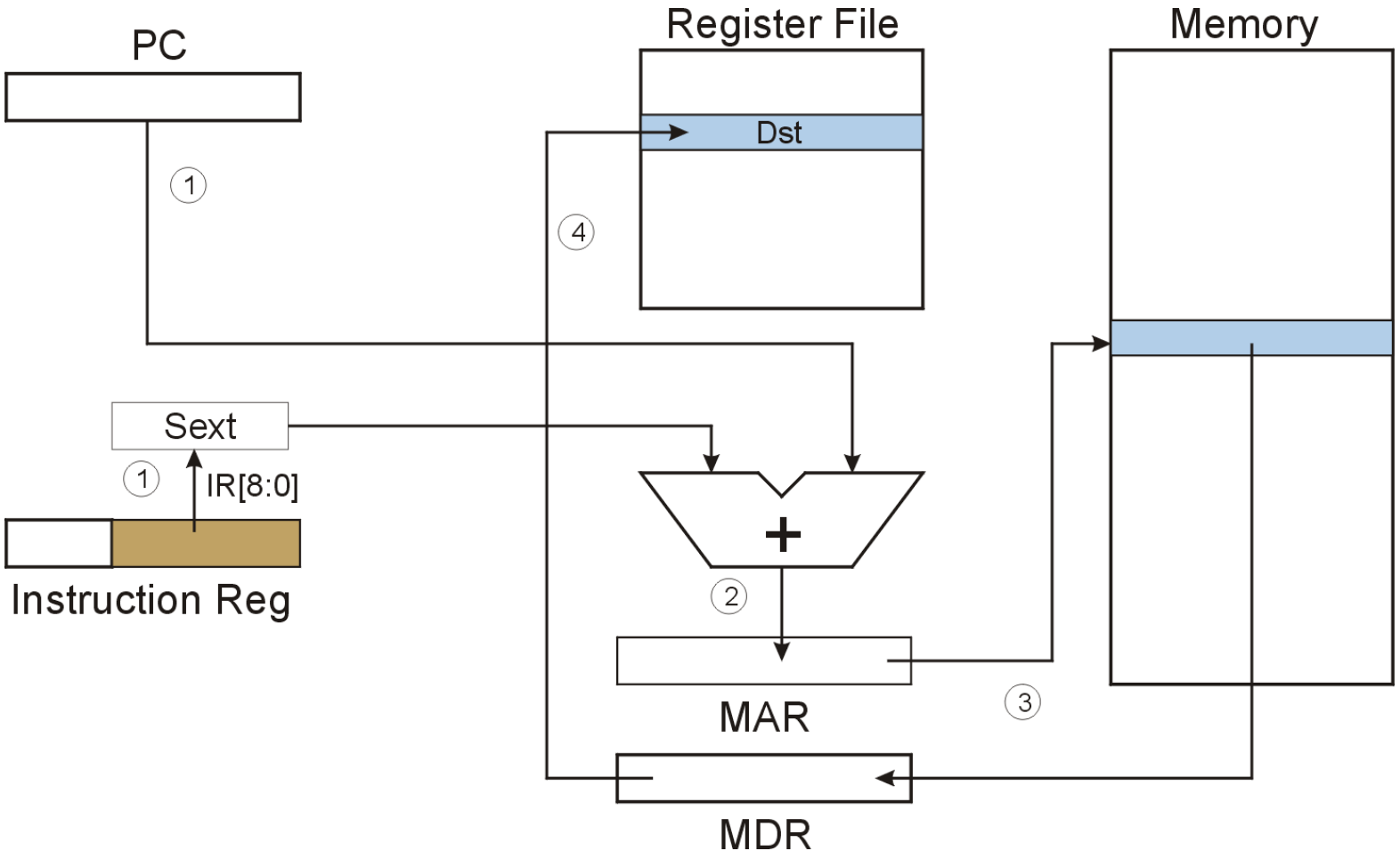
$$-256 \leq \text{offset} \leq +255$$

$$PC - 256 \leq X \leq PC + 255$$

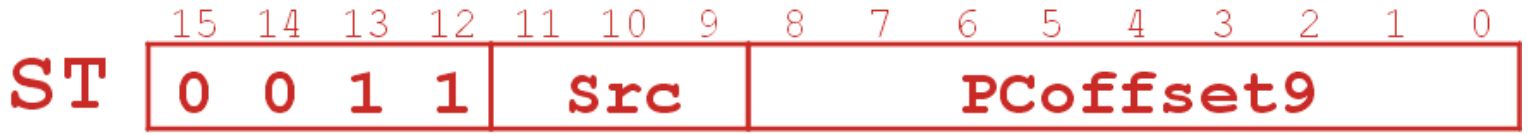
- Remember that PC is incremented as part of the FETCH phase;
- This is done before the EVALUATE ADDRESS stage.

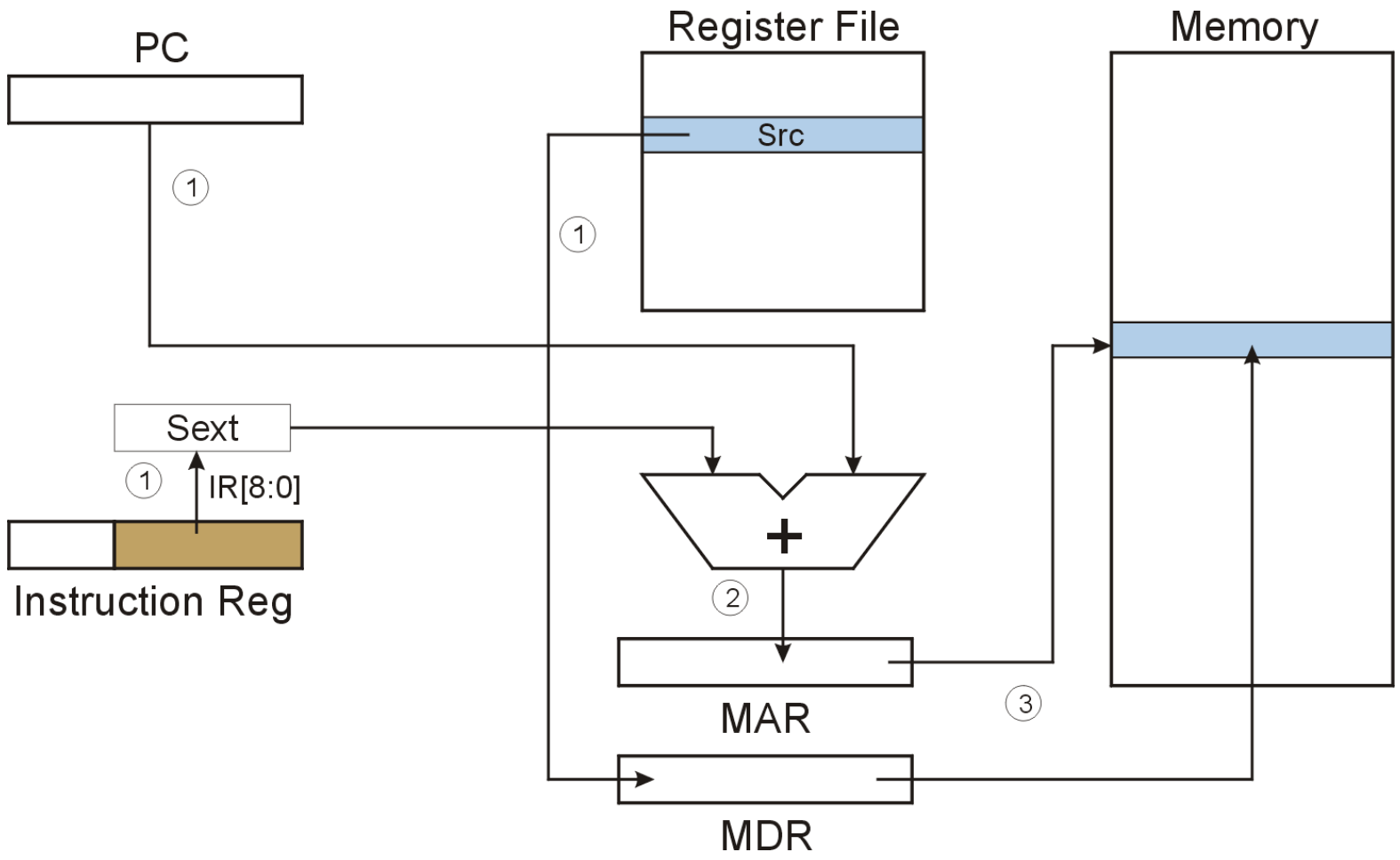
LD (PC-Relative)





ST (PC-Relative)





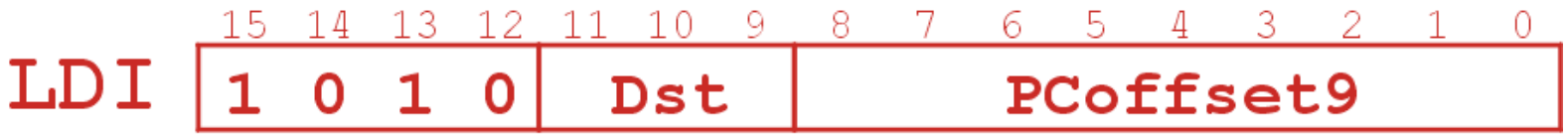
Indirect Addressing Mode

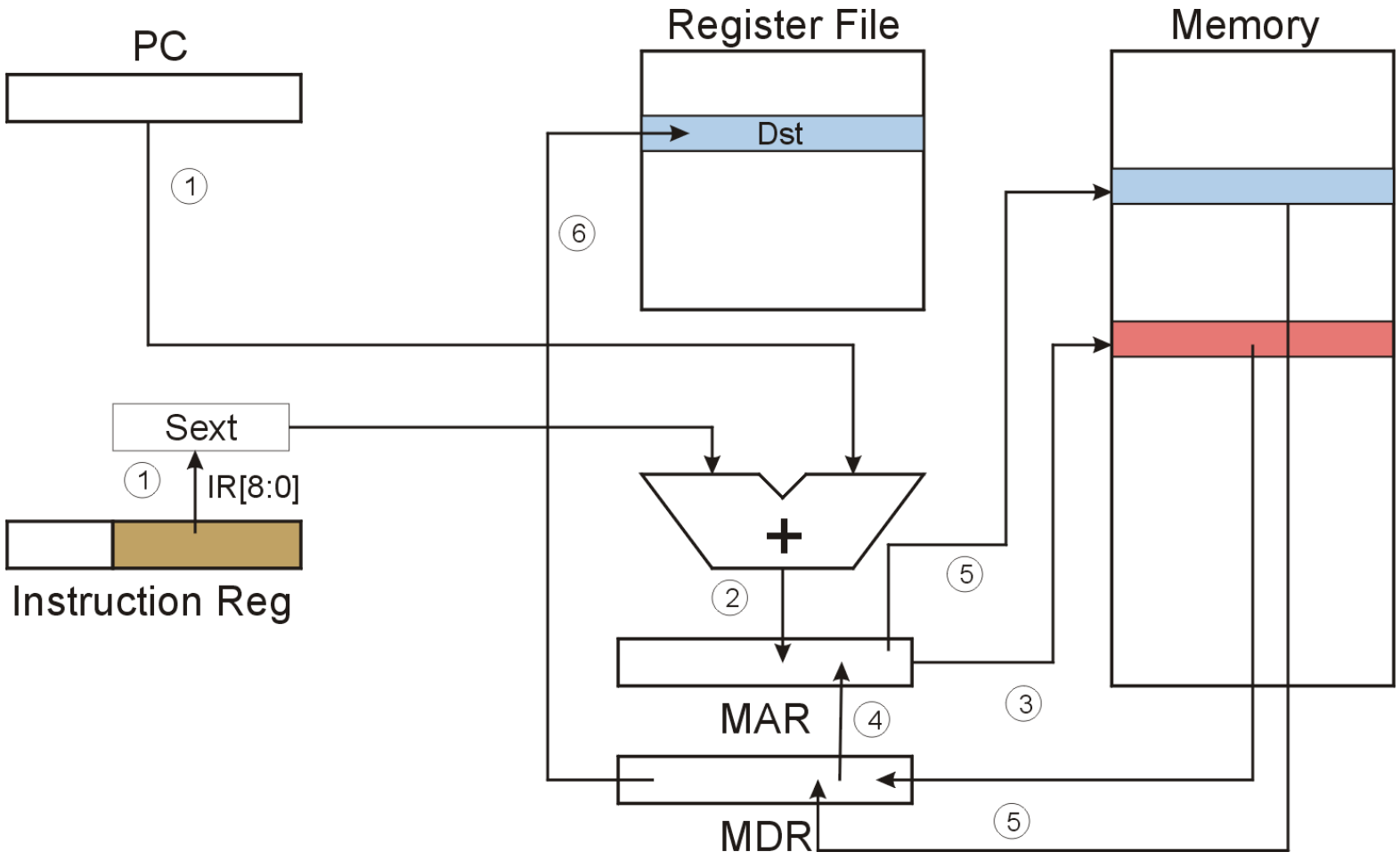
- With PC-relative mode, can only address data within 256 words of the instruction.
  - What about the rest of memory?

- Solution #1:
  - Read address from memory location, then load/store to that address.

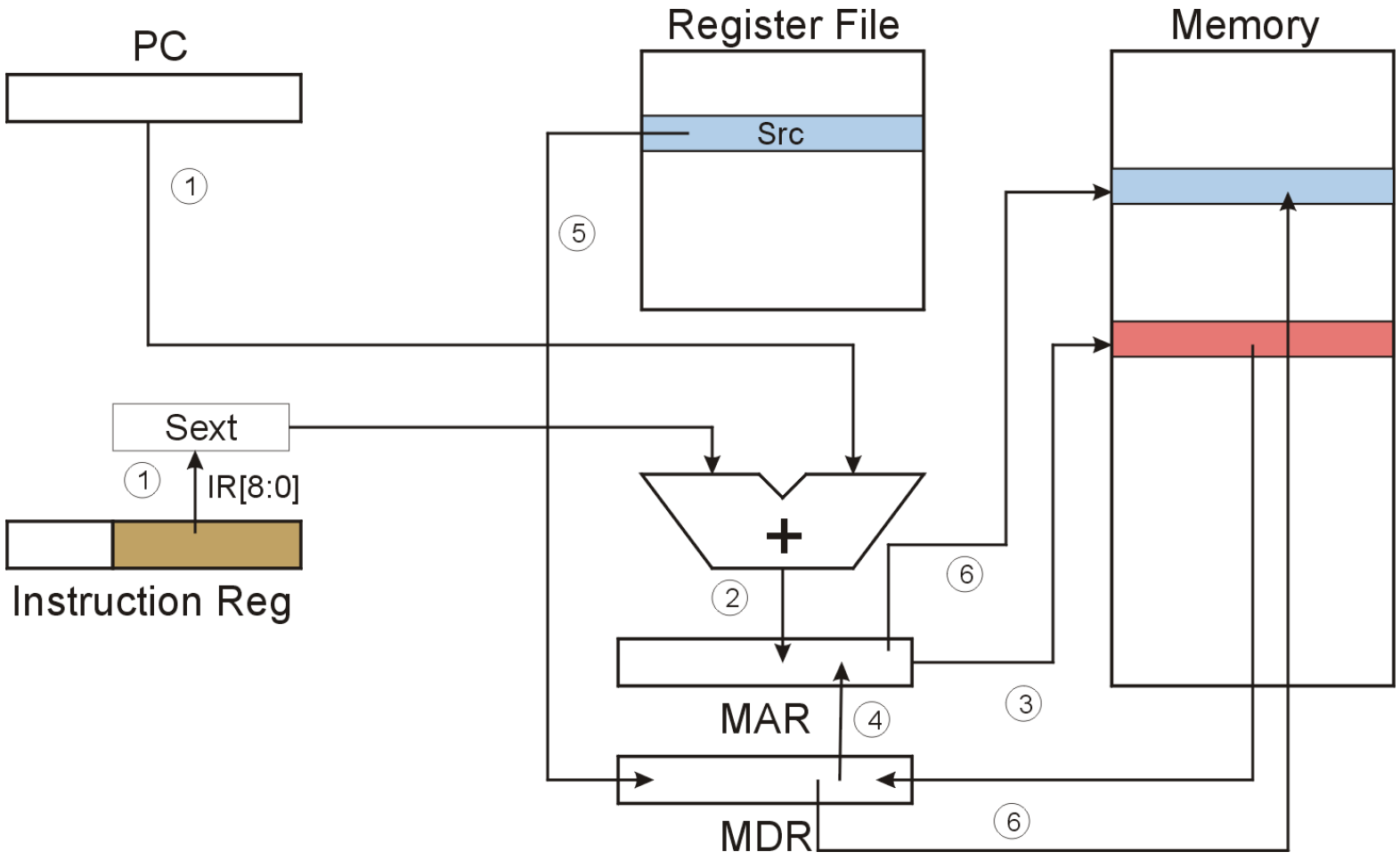
- First address is generated from PC and IR (just like PC-relative addressing), then content of that address is used as target for load/store.

LDI (Indirect)





STI (Indirect)



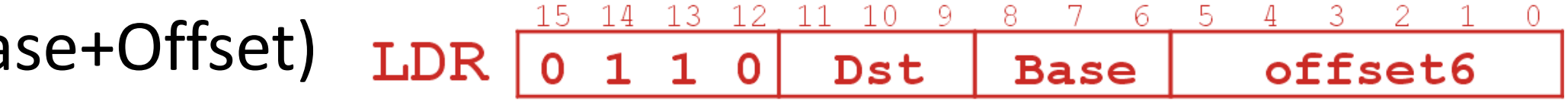
Base + Offset Addressing Mode

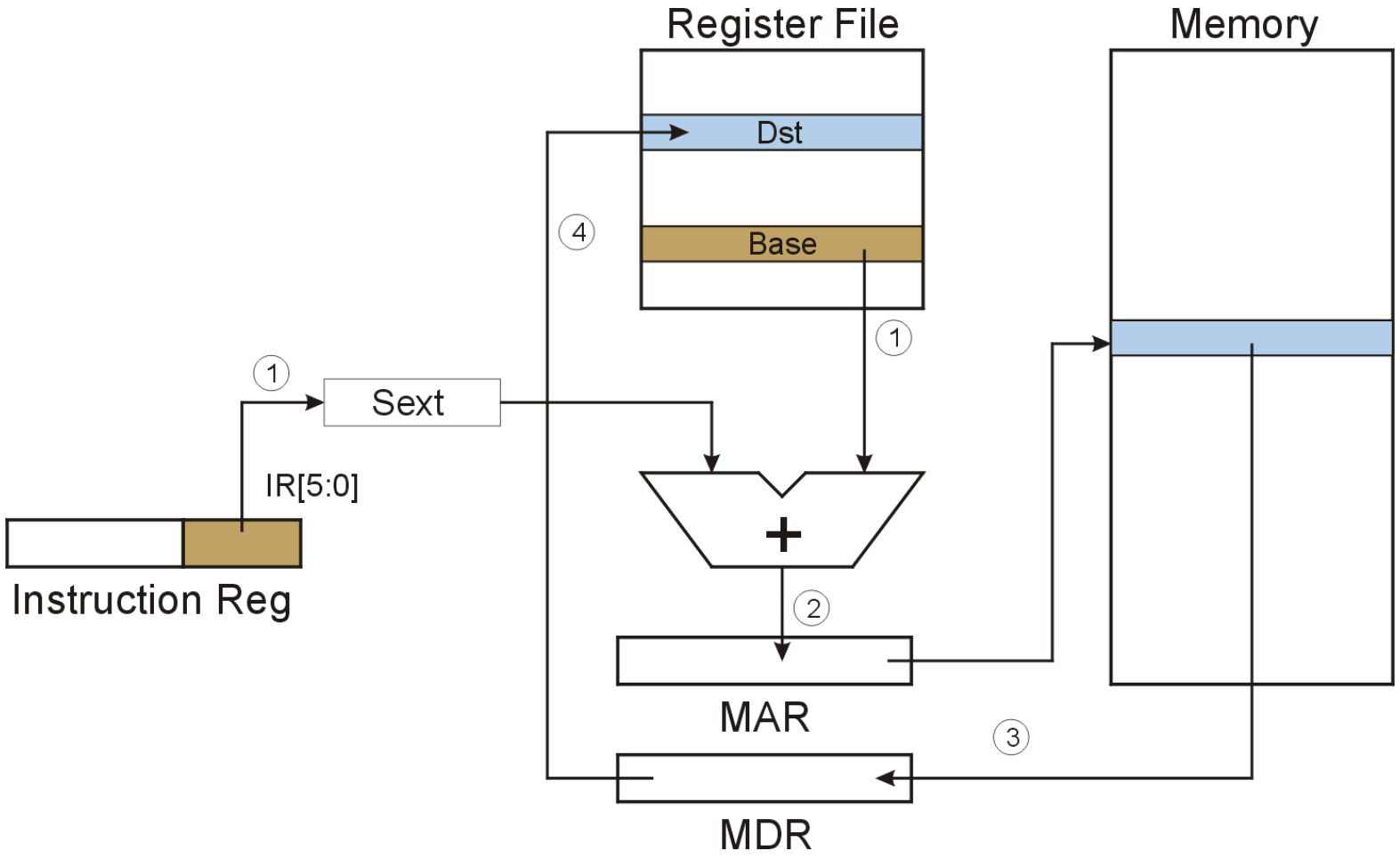
- With PC-relative mode, can only address data within 256 words of the instruction.
  - What about the rest of memory?

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

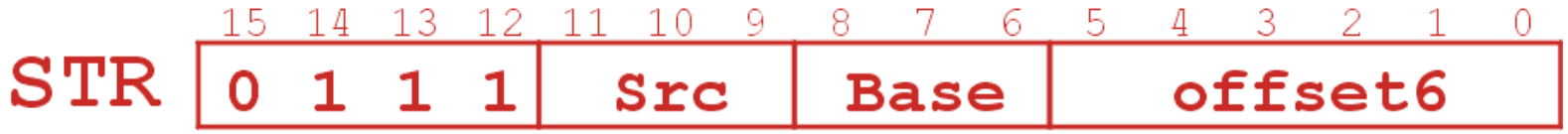
- 4 bits for opcode, 3 for src/dest register, 3 bits for *base* register -- remaining 6 bits are used as a *signed offset*.
  - Offset is *sign-extended* before adding to base register.

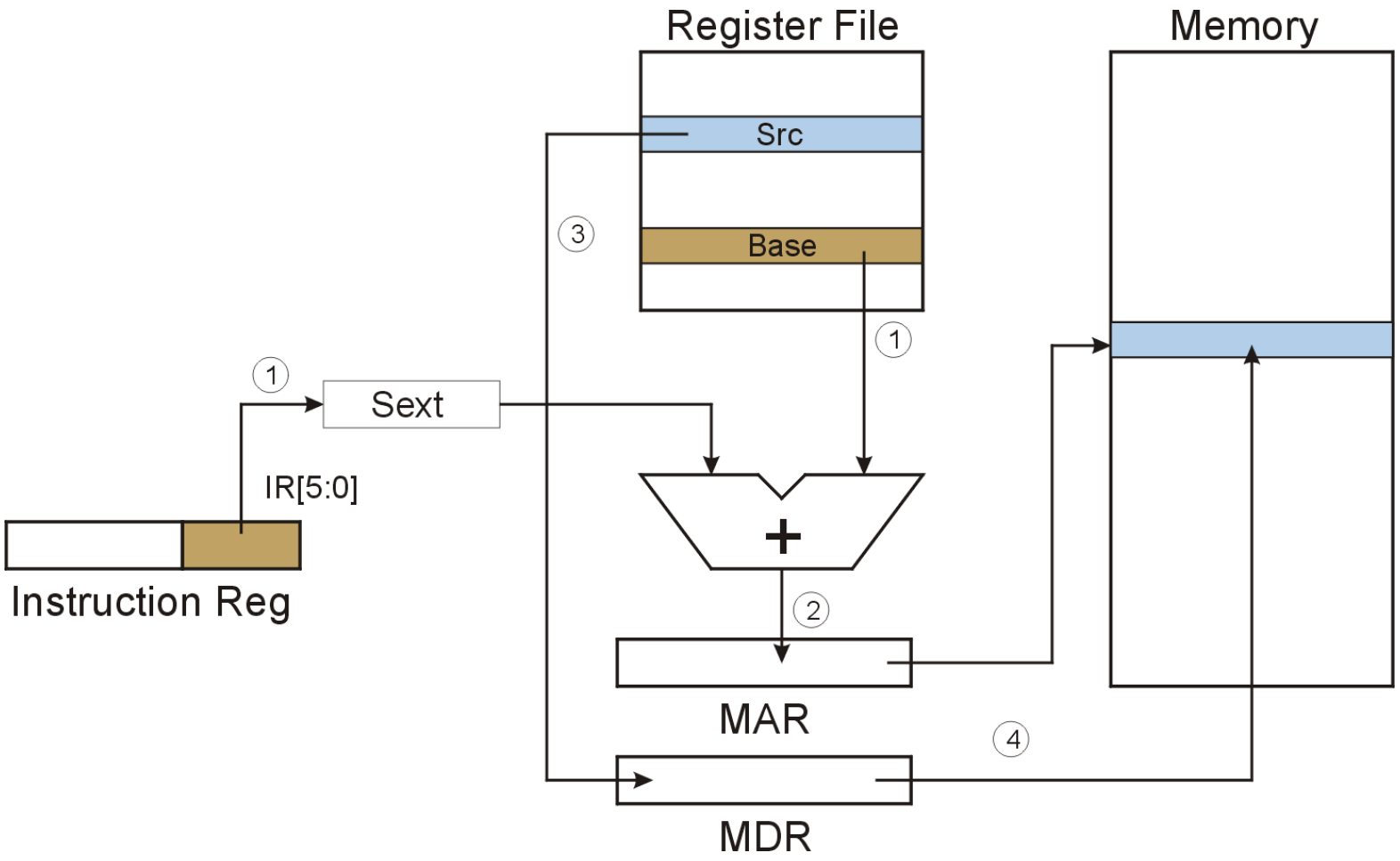
LDR (Base+Offset)





STR (Base+Offset)





Load Effective Address

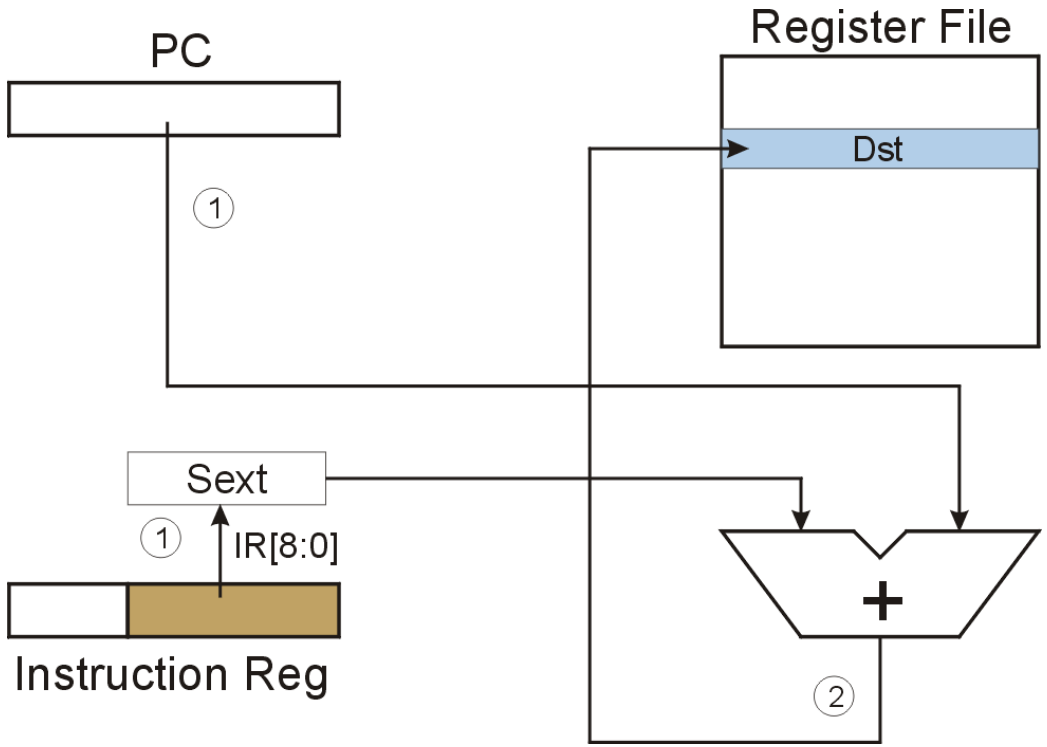
- Computes address like PC-relative (PC plus signed offset) and stores the result into a register.

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

LEA (I

LEA


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


PC
Register File
Dst
1
Sext
1
IR[8:0]
+
2
Instruction Reg

Control Instructions

- Used to alter the sequence of instructions (by changing the Program Counter)
- conditional branch, unconditional jump, subroutine call (sometimes
- called *function*), TRAP, and RTI (Return from Trap or Interrupt).
- 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

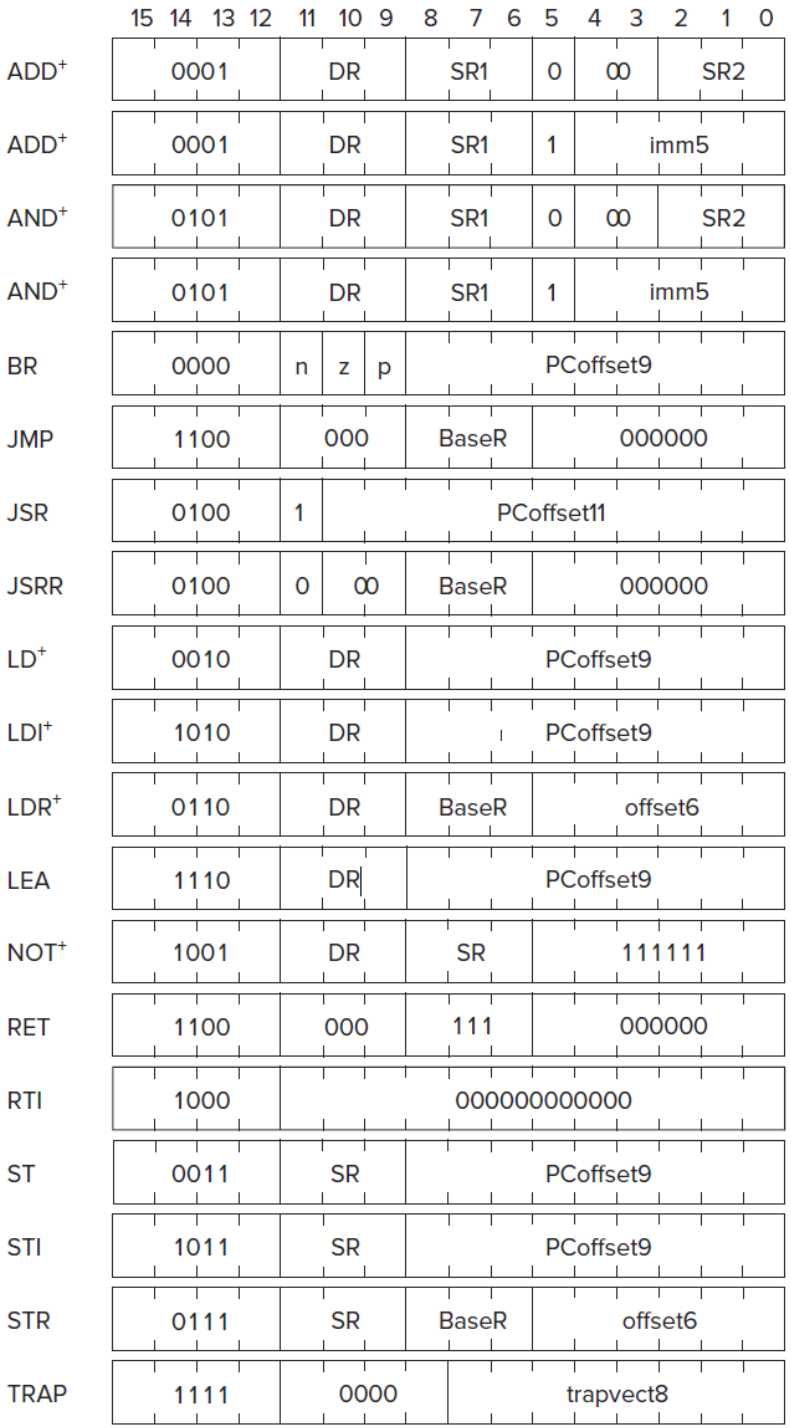
- LC-3 has three condition code registers:
  N -- negative
  Z -- zero
  P -- positive (greater than zero)

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

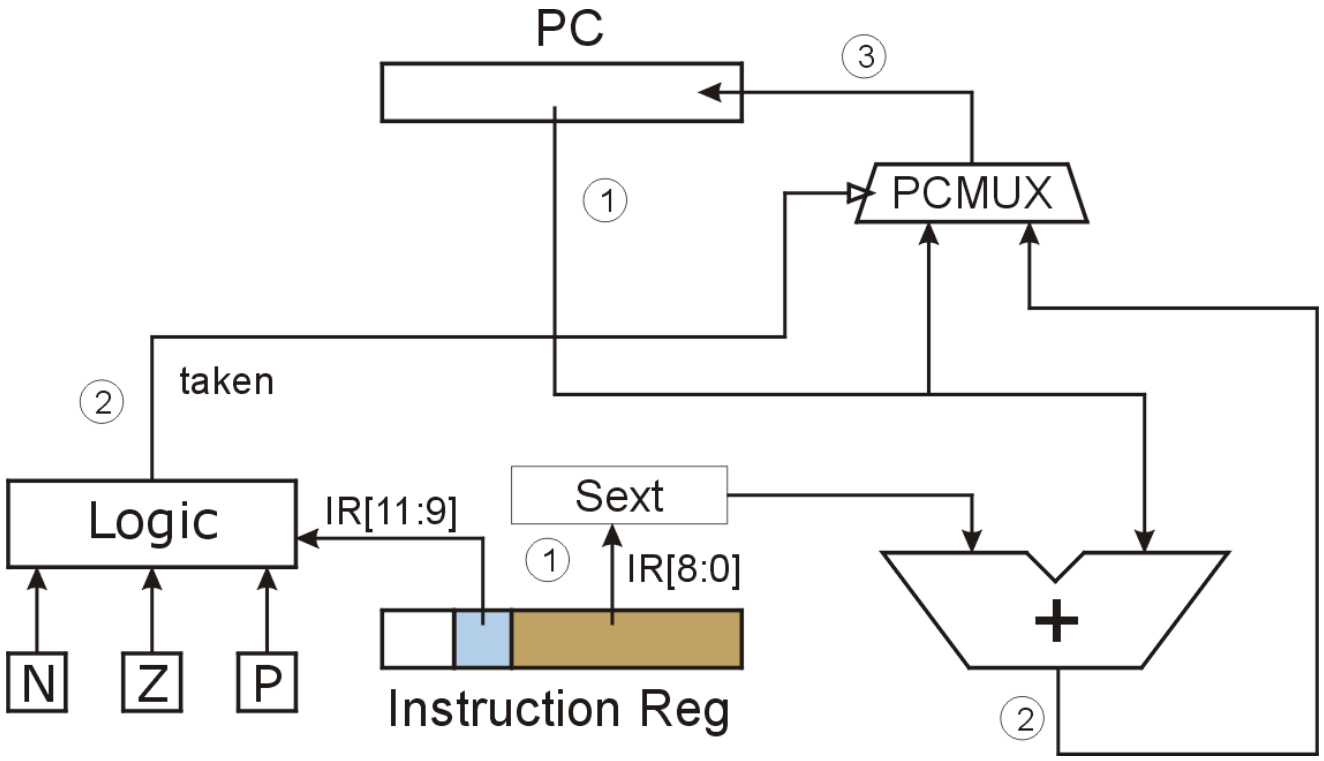
- Exactly one will be set at all times
  - Based on the last instruction that altered a register

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.


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

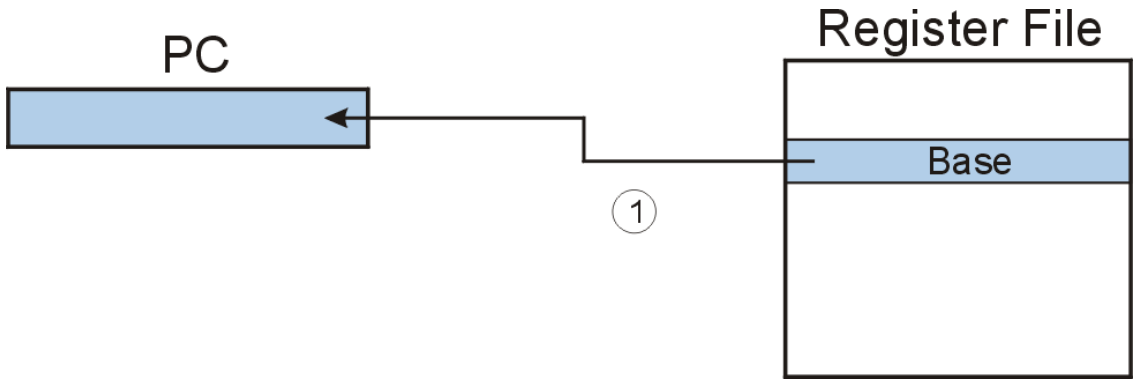
BR (PC-Relative)



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

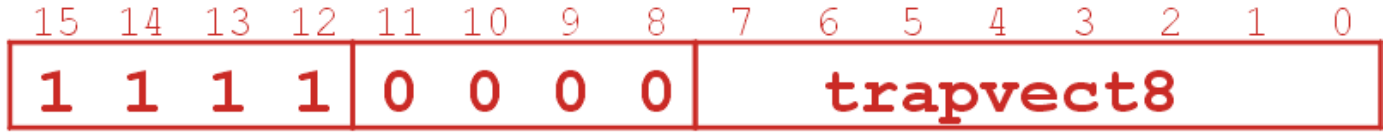
JMP (Register)

- Jump is an unconditional branch -- *always* taken.
  - Target address is the contents of a register.
  - Allows any target address.



TRAP

TRAP



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

vector	*routine*
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.)