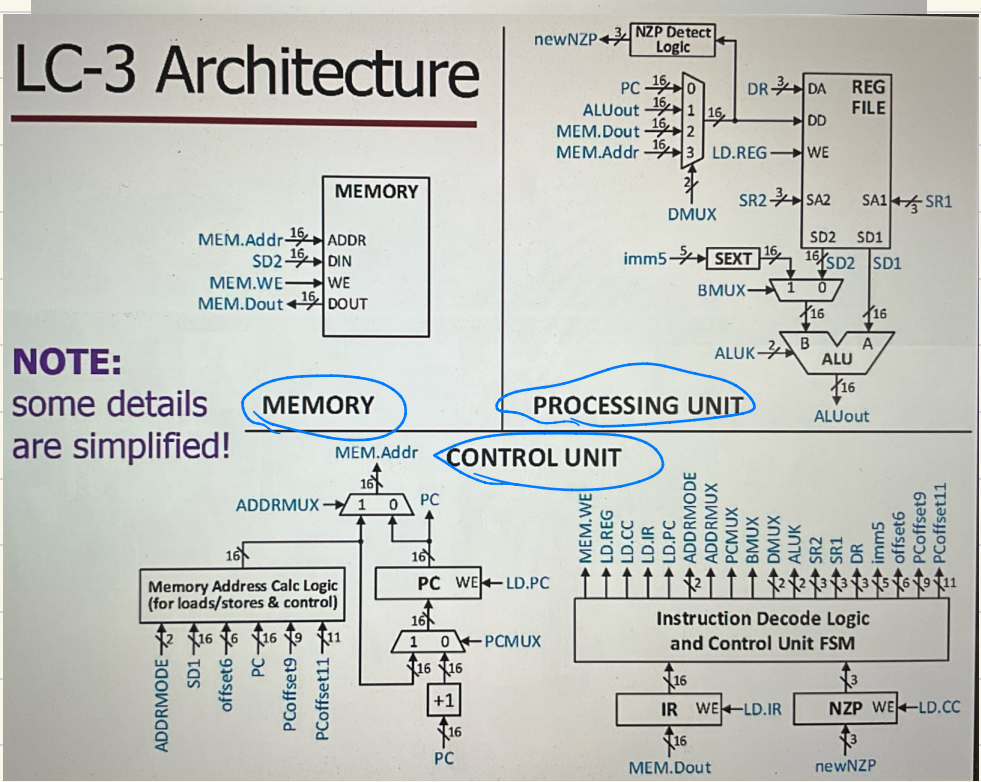
LC-3 ISA and Processor Overview

Instruction Set Architecture

- Specifies all information needed to program in machine language (binary)
  - Instruction bitwidth (number of bits)
  - Instruction opcodes and operations they represent
  - Other instruction fields and how they are used
  - Format (datatype) of the data the instruction operates on (unsigned binary vs. 2's-complement, etc.)
  - Any other effects of instructions (setting flags, etc.)
  - Number and size of registers, and how they can be used by instructions
  - Memory address space (number of addressable locations) and its addressability (bits per location)

LC-3 Architecture



LC-3 Instruction Set Architecture

OPCODE				DEST REG			SRC REG 1			MODE	5-bit CONSTANT				
0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

OPCODE				DEST REG			SRC REG 1			MODE			SRC REG 2		
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Instructions are 16 bits wide, divided into several **fields** of information
  - Bits 15 through 12 are the 4-bit **opcode**
    - The opcode for the current instruction is in **IR**[15:12]
  - Remaining bits are divided up into various other fields (the exact set of fields depends in part on the opcode)

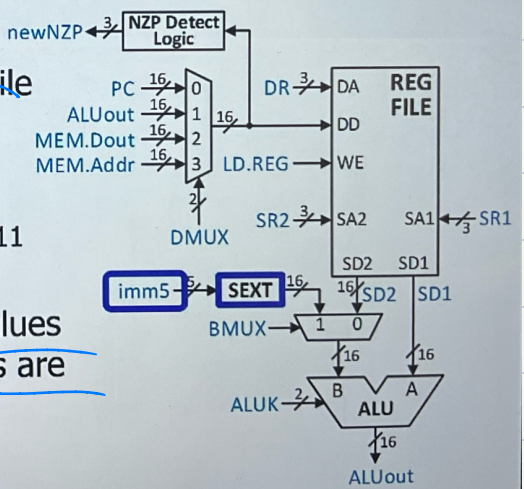
(Opcode is used to identify the types of instructions)

Decoded in the IR

LC-3 Processing Unit

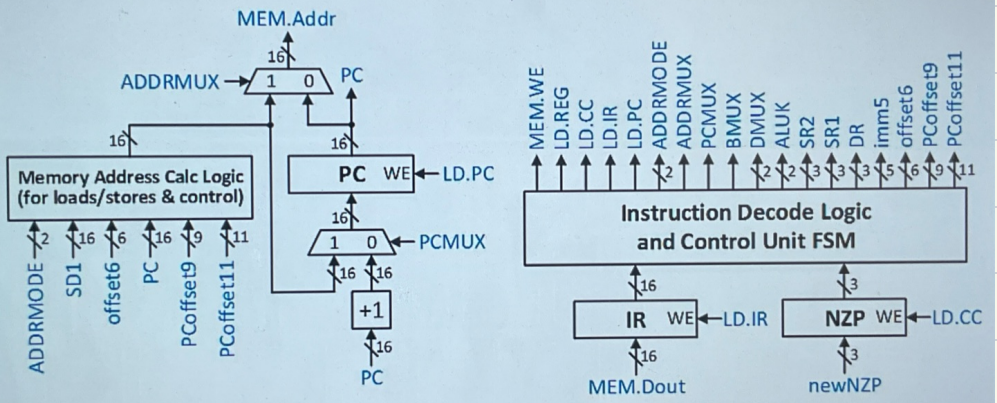
OPCODE				DEST REG			SRC REG 1			MODE	5-bit CONSTANT				
0	0	0	1	0	0	0	0	1	1	1	0	0	0	0	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Eight general-purpose registers in the register file
  - Referenced using a 3-bit register address
  - LC-3 assembly language: R3 is register w/address 011
  - Data word size is 16 bits
- Constant (immediate) values encoded into instructions are 2's-complement format
  - Sign-extended to 16 bits before processing



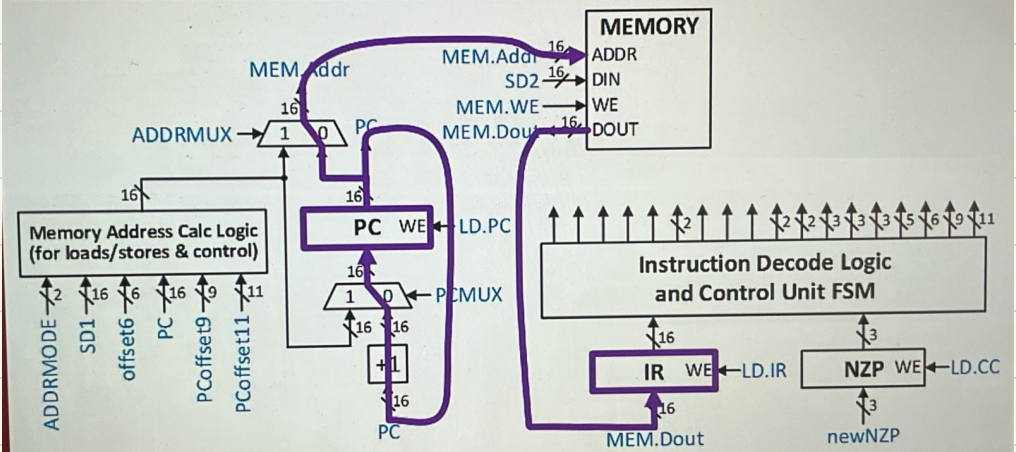
- The address in the register file is 3-bit long : 8 registers
- The data word size is 16-bit long
- The imm5 is the input signal of any constant value, they do not need to be acquired from the memory. The SEXT logic is to sign-extend the constant to 16-bit long

LC-3 Control Unit



- Control Unit coordinates and controls processor resources to execute a program

LC-3 Instruction FETCH

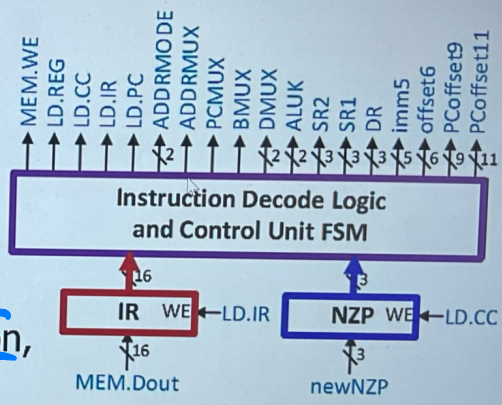


- Read from memory using **PC** as address
- Place value read from memory (the next instruction that will be executed) into the **IR**
- Increment the **PC**

(☆: The PC will not be incremented until the instruction in IR has already been executed)

LC-3 Control Signals

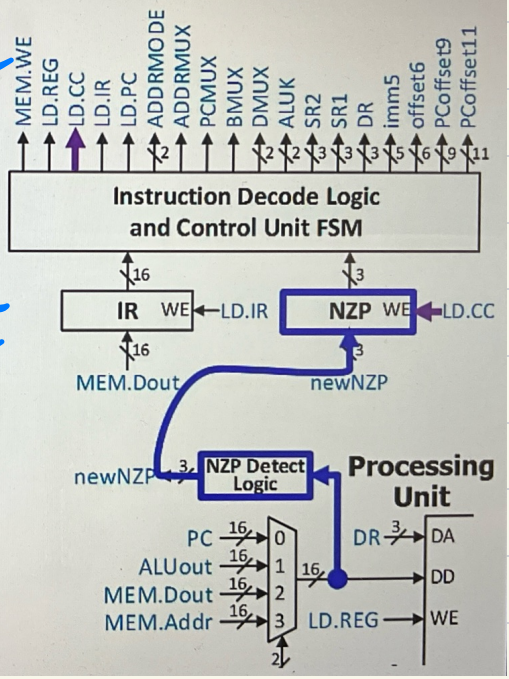
- Finite State Machine and instruction decode logic generates signals to control other resources
- Affected by the binary instruction in the **IR**
- If current instruction is a conditional Control instruction, output is also affected by **condition codes** (**CCs**)



(change the PC only if the conditions indicated in the current instructions are true)

LC-3 Condition Codes

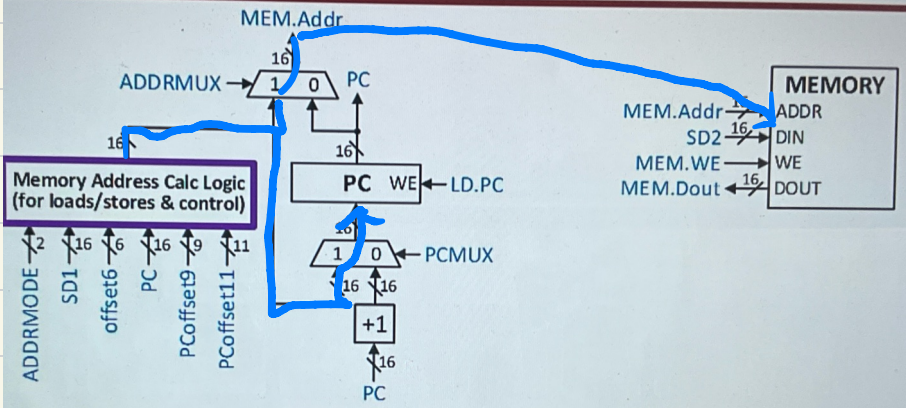
- Condition codes (**CCs**) are stored in a special register
- Indicate information about the most recent result in the Processing Unit
  - **N**egative, **Z**ero, or **P**positive
  - Three 1-bit flags; exactly one of these equals 1 at any time!
- Control Unit causes CCs to update as part of executing some instruction types
  - Operate instructions
  - Load instructions



(CC is what that allows us to create conditional branches that let the processor execute different parts of a program)

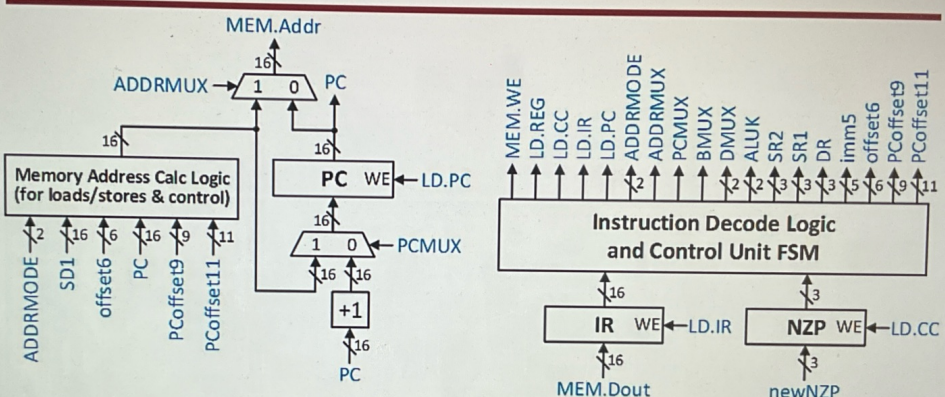
Tells CPU how to proceed based on the result

LC-3 Memory Address Calculation



- Other logic calculates memory addresses
  - Addresses used for Data Movement (loads/stores)
  - New PC value for Control instructions
    - Remember – the PC holds a memory address…
  - These calculations are specified by the ISA

LC-3 Memory Address Calculation

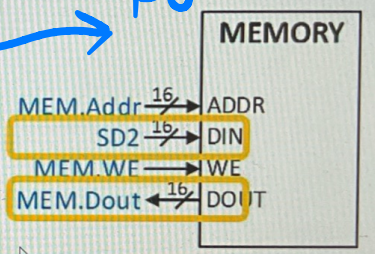


- The address calculation method depends on the **memory addressing mode** of the instruction
  - Add an offset to a value from the register file
  - Add one of two possible offsets to the PC
- Offset values are decoded from the instruction (in IR)

LC-3 Memory

- Holds both instructions and program data
  - Memory's DOUT (data out) output goes to both the **IR** in the Control Unit and to the Processing Unit
- Only Data Movement **store** instructions write to memory
  - Memory's DIN (data in) input comes from the Register File
- 16-bit memory addresses (range: 0x0000 – 0xFFFF)
- 16-bit memory data word size
  - Same as size of instruction and register data word size

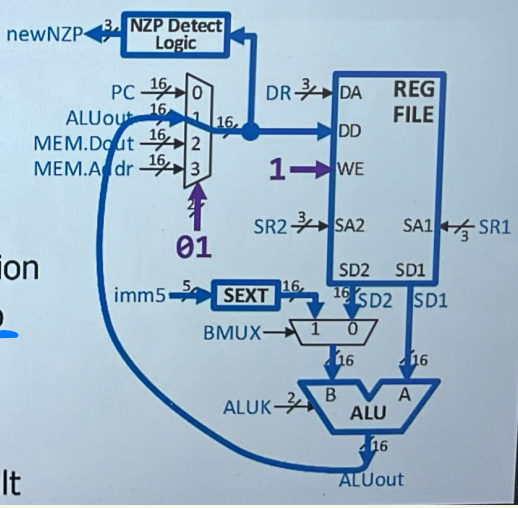
Register File



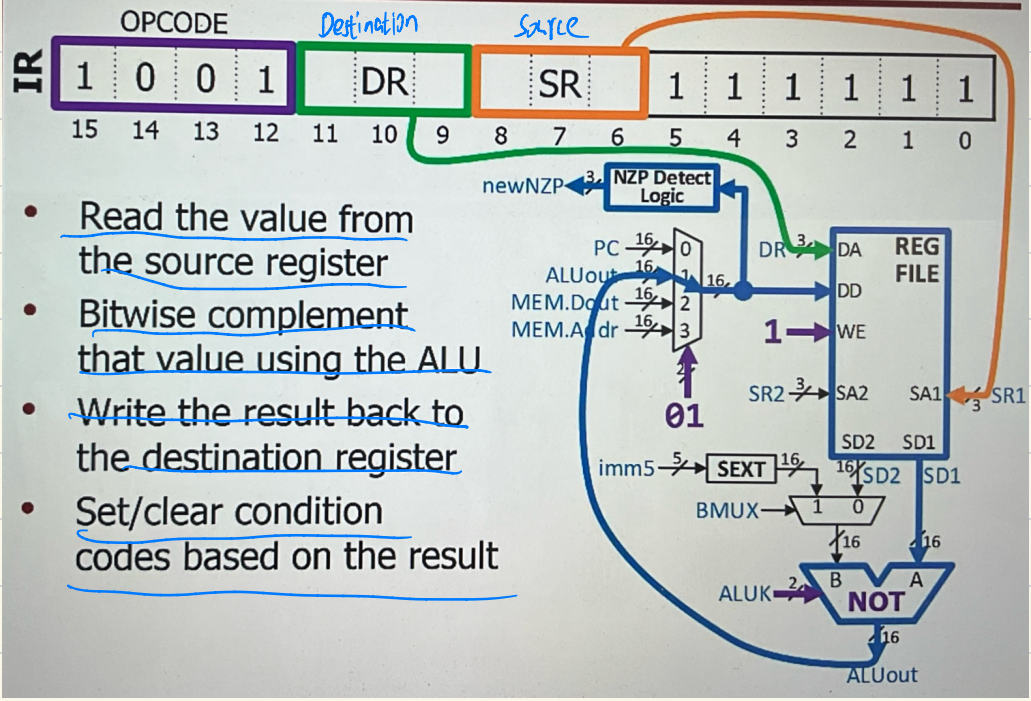
LC-3 Operate Instruction

Operate Instructions

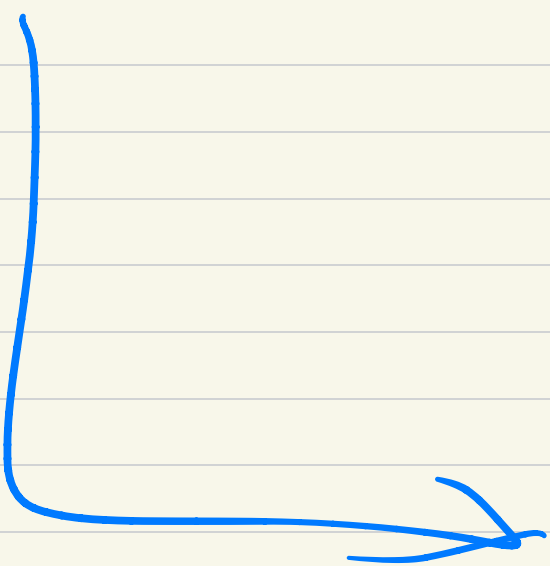
- Perform computations using the Processing Unit
- At least one source operand is read from the register file
- May have 2nd operand
  - Another register value
  - A constant encoded into the instruction itself
- Performs an ALU operation
- Writes the result back to the destination register in the register file
- Sets/clears condition codes based on the result

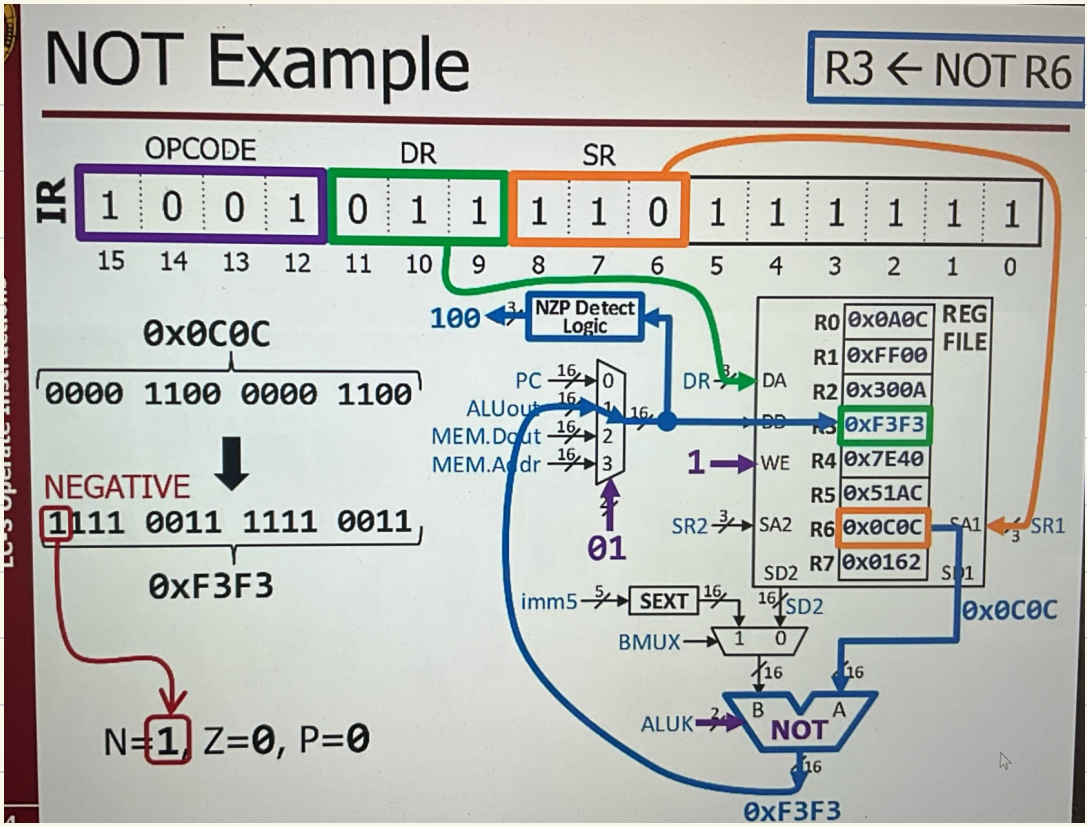


NOT Instruction



- Read the value from the source register
- Bitwise complement that value using the ALU
- Write the result back to the destination register
- Set/clear condition codes based on the result



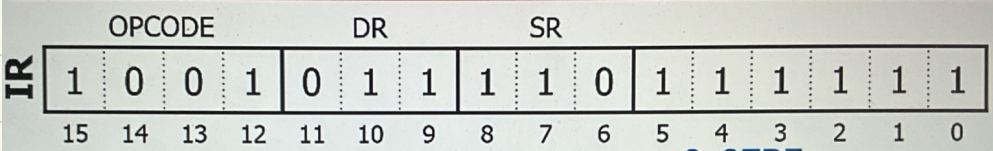


1. Opcode → NOT, sets several variables in Processing Unit accordingly

2. Source Register's data goes to ALU, after not operation, store in Destination Register, overwritting the previous value there.

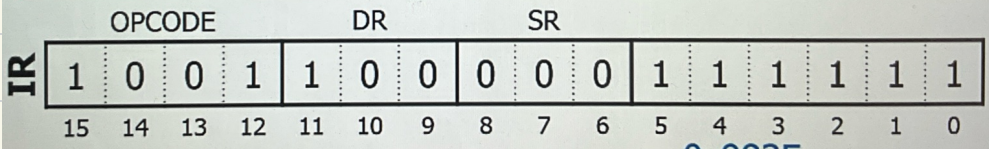
3. The result is negative → NZP detect logic for later operations

Example NOT Instructions



0x97BF

- Machine Language: 1001 0111 1011 1111
- LC-3 Assembly Language: NOT R3, R6
- Register Transfer Language: R3 ← NOT R6



0x983F

- Machine Language: 1001 1000 0011 1111
- LC-3 Assembly Language: NOT R4, R0
- Register Transfer Language: R4 ← NOT R0

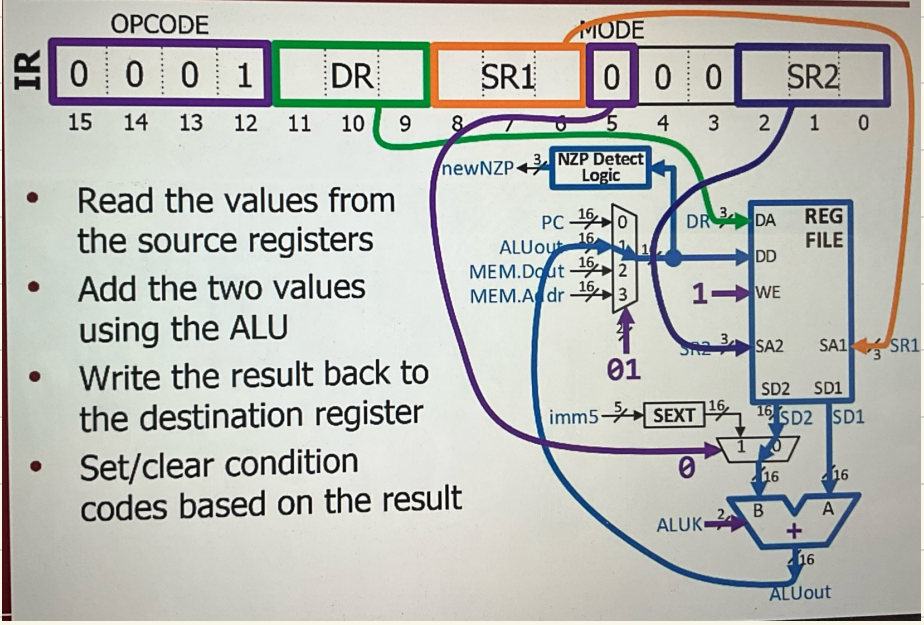
(☆ Register Transfer Language is not used in machines, it is for human understanding)

Assembly Language Syntax

Operation Type Destination Source

#1

ADD Instruction

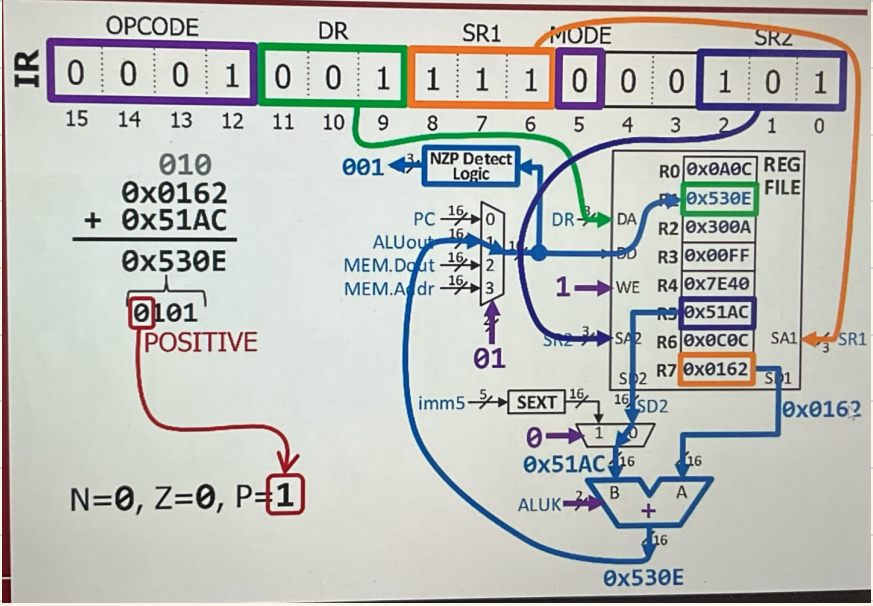


- Read the values from the source registers
- Add the two values using the ALU
- Write the result back to the destination register
- Set/clear condition codes based on the result

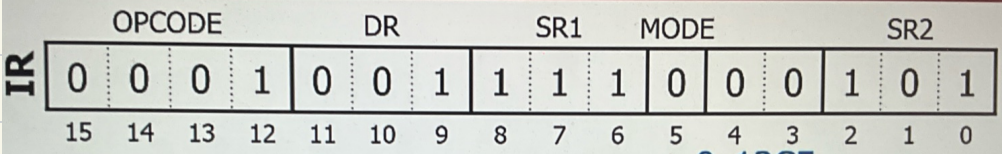
(☆ Mode is 0; Two source registers)

ADD Example

R1 ← R7 + R5

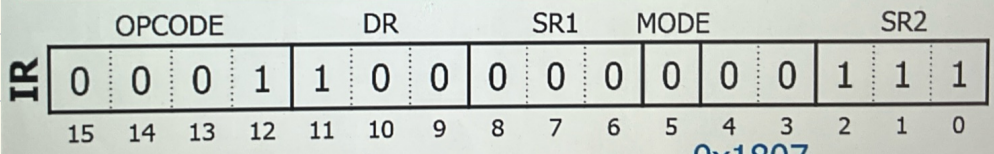


Example ADD Instructions



0x13C5

- Machine Language: 0001 0011 1100 0101
- Register Transfer Language: R1 ← R7 + R5
- LC-3 Assembly Language: ADD R1, R7, R5

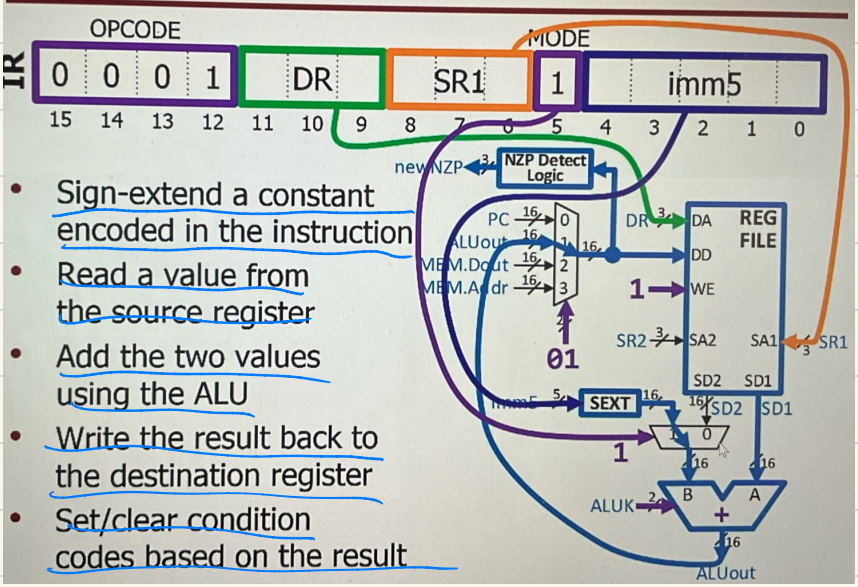


0x1807

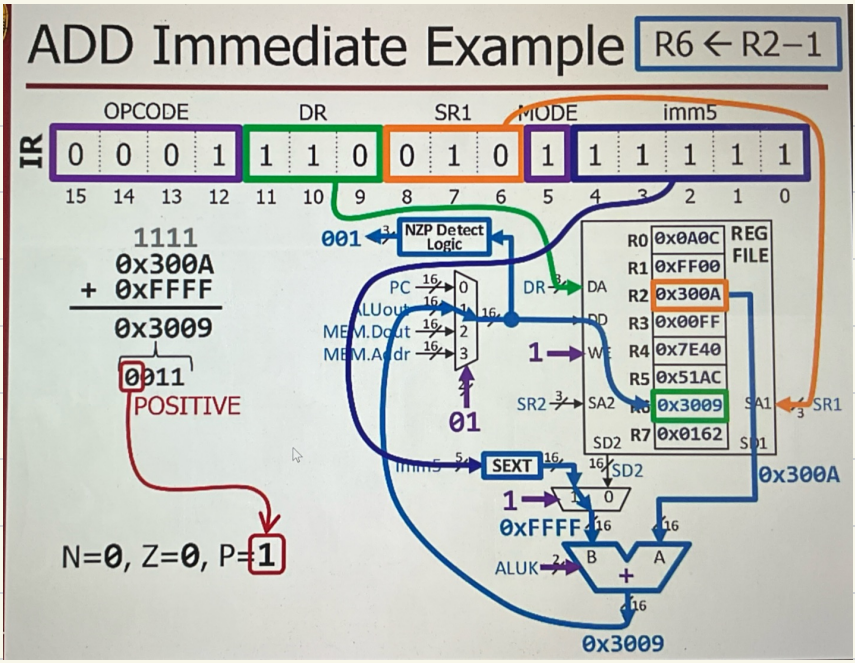
- Machine Language: 0001 1000 0000 0111
- Register Transfer Language: R4 ← R0 + R7
- LC-3 Assembly Language: ADD R4, R0, R7

#2

ADD Immediate Instruction



- Sign-extend a constant encoded in the instruction
- Read a value from the source register
- Add the two values using the ALU
- Write the result back to the destination register
- Set/clear condition codes based on the result



(Opcode → Mode → IMM → SR1 → Execute → DR → N2P)

Example ADD Imm. Instructions

OPCODE DR SR1 MODE imm5

IR: 0 0 0 1 | 1 1 0 | 0 1 0 | 1 | 1 1 1 1 1

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

0x1CBF

- Machine Language: 0001 1100 1011 1111
- Register Transfer Language: R6 ← R2 – 1
- LC-3 Assembly Language: ADD R6, R2, #-1

OPCODE DR SR1 MODE imm5

IR: 0 0 0 1 | 0 0 0 | 1 0 0 | 1 | 0 1 1 1 1

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

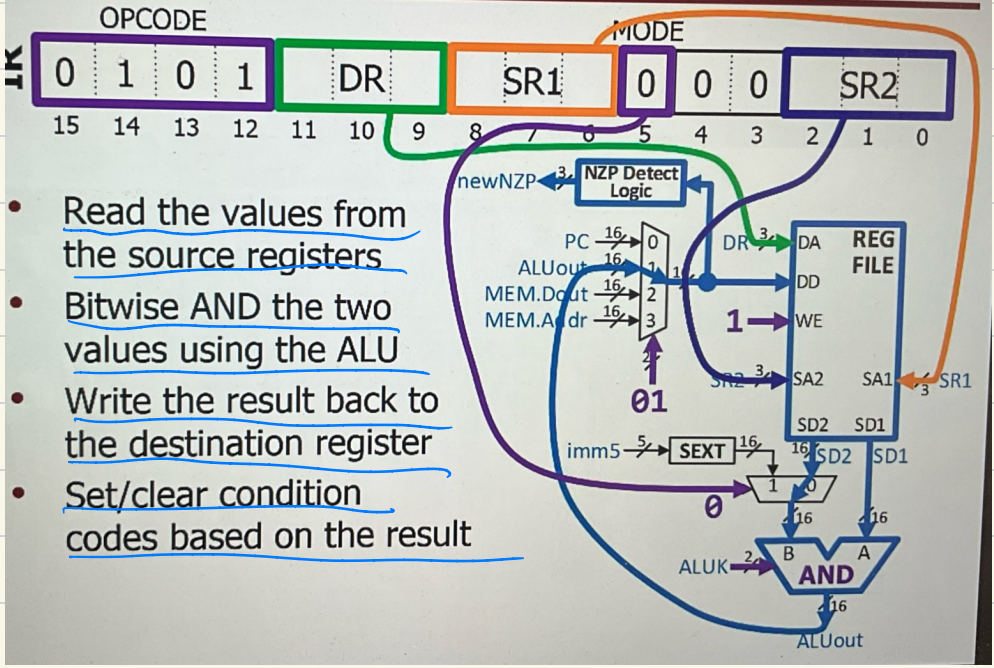
0x112F

- Machine Language: 0001 0001 0010 1111
- Register Transfer Language: R0 ← R4 + 15
- LC-3 Assembly Language: ADD R0, R4, #15

Indicate in decimal

#1

AND Instruction

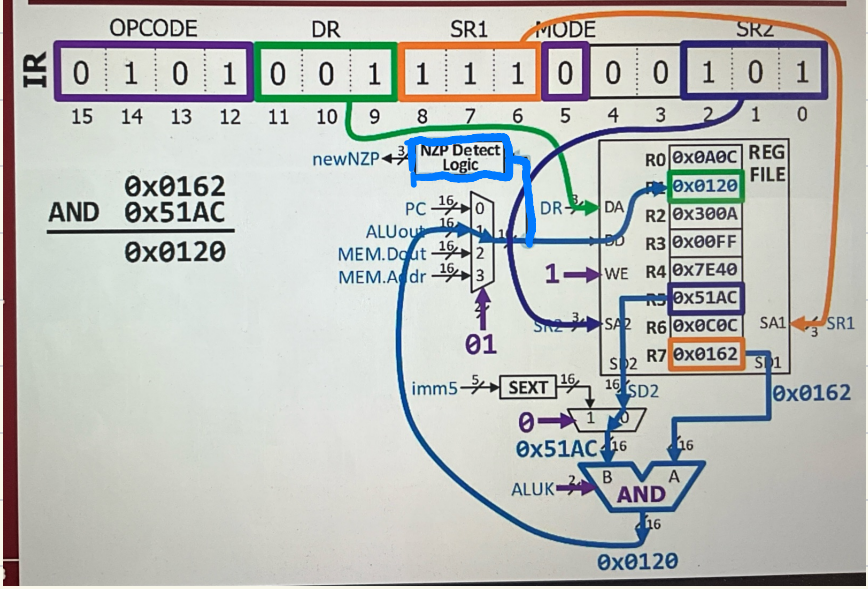


- Read the values from the source registers
- Bitwise AND the two values using the ALU
- Write the result back to the destination register
- Set/clear condition codes based on the result

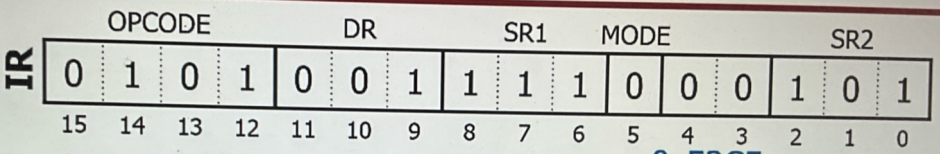
(★ Basically identical to Add)

AND Example

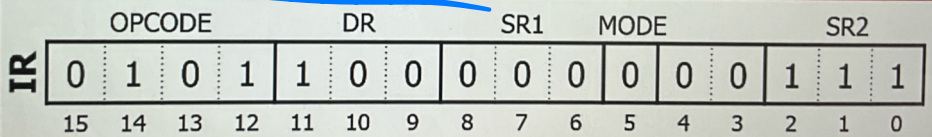
R1 ← R7 AND R5



Example AND Instructions



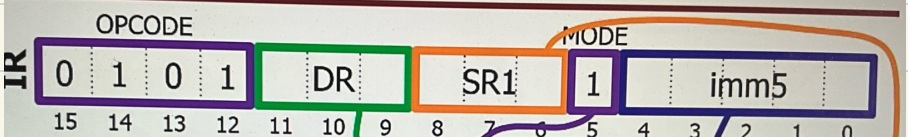
- Machine Language: 0101 0011 1100 0101 (0x53C5)
- Register Transfer Language: R1 ← R7 AND R5
- LC-3 Assembly Language: AND R1, R7, R5



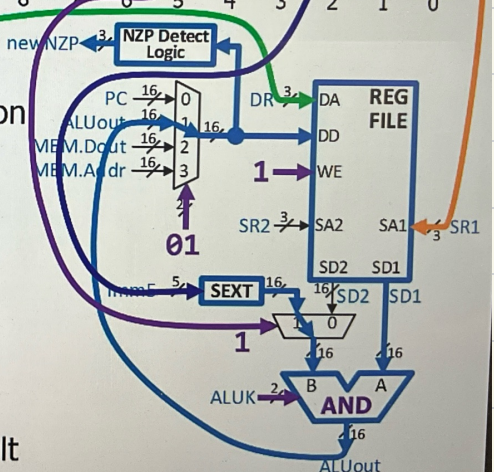
- Machine Language: 0101 1000 0000 0111 (0x5807)
- Register Transfer Language: R4 ← R0 AND R7
- LC-3 Assembly Language: AND R4, R0, R7

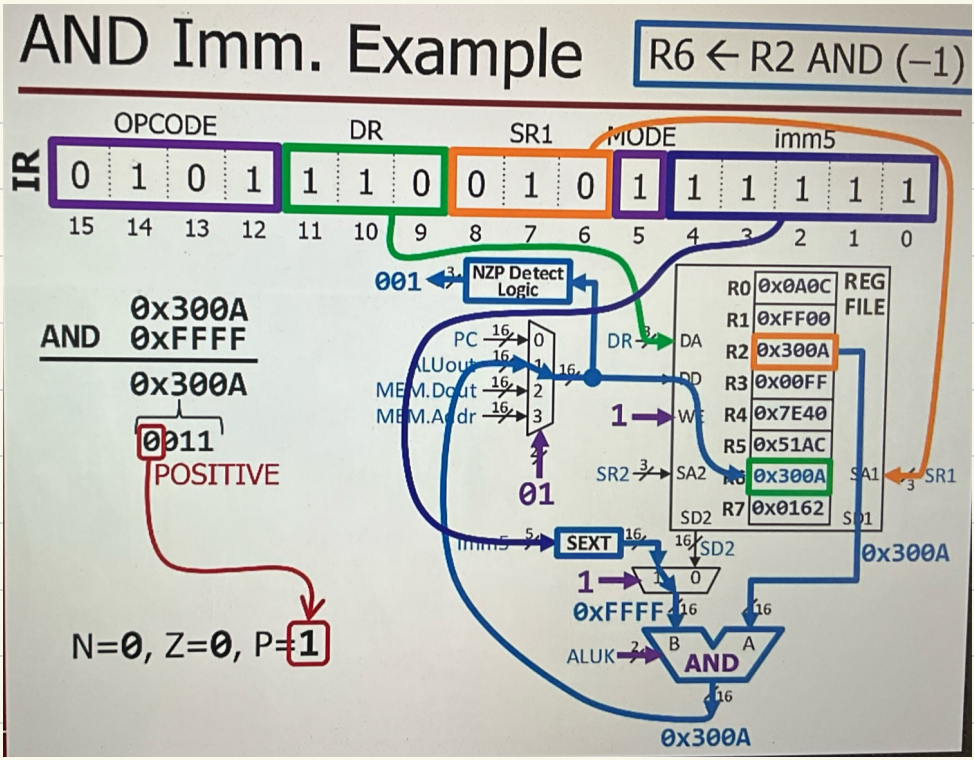
#2

AND Immediate Instruction



- Sign-extend a constant encoded in the instruction
- Read a value from the source register
- Bitwise AND the two values using the ALU
- Write the result back to the destination register
- Set/clear condition codes based on the result



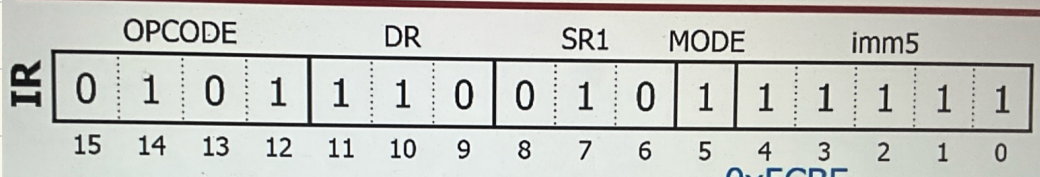


(☆ When AND 2 numbers, the result is the upper operand)

↓

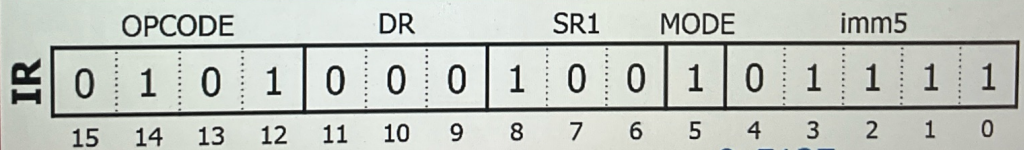
Basically "copying" the value to another location

Example AND Imm. Instructions



0x5CBF

- Machine Language: 0101 1100 1011 1111
- Register Transfer Language: R6 ← R2 AND (-1)
- LC-3 Assembly Language: AND R6, R2, #-1



0x512F

- Machine Language: 0101 0001 0010 1111
- Register Transfer Language: R0 ← R4 AND 15
- LC-3 Assembly Language: AND R0, R4, #15

Simple Programs with LC-3

Operate Instructions

Assembly Language: It is close to machine language, but it has several features that make it easier to use (Higher Level of Abstraction)

The assembler translates assembly language into machine language that can be executed by the processor (Low Level)

Example: Sum Register Values

sum.asm → file extension

```
; This program adds four values
; in the register file and puts
; the result into register R0
;    R0 ← ((R1 + R2) + R3) + R4

.ORIG x0200

ADD R0, R1, R2   ; add two values
ADD R0, R0, R3   ; then another
ADD R0, R0, R4   ; then the last

.END
```

comments explain program code to people

instructions tell the processor what to do

Directives give information to the assembler

sum.asm

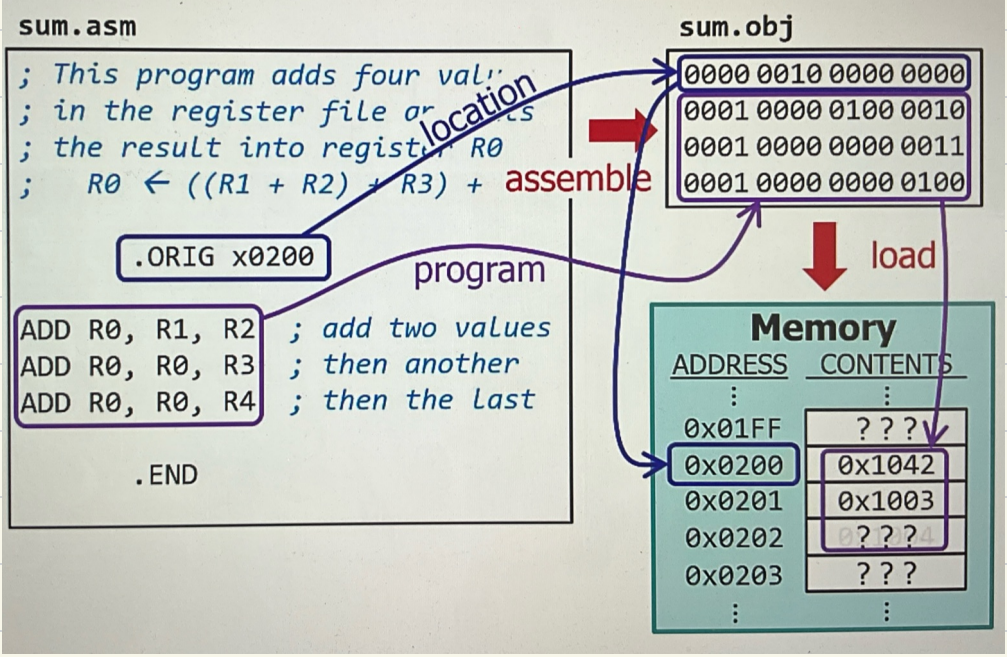
```
; This program adds four values
; in the register file and puts
; the result into register R0
;    R0 ← ((R1 + R2) + R3) + R4

.ORIG x0200

ADD R0, R1, R2   ; add two values
ADD R0, R0, R3   ; then another
ADD R0, R0, R4   ; then the last

.END
```

start this program at the indicated memory address

indicates **(to assembler)** end of the program

Assembler directives are not instructions!

Example: Sum Register Values



Essential Programming Skills

- These small tasks are building blocks that are important to many programs!
  - Clear a register ①
  - Initialize a register to a specific value ②
  - Copy the value in one register to another register ③
  - Increment or decrement the value in a register ④
  - Negate the value in a register ⑤
  - Subtract the value in one register from another ⑥
  - Mask certain bits in a register ⑦

Register Initialization

- Initialize to zero ① clear Register

```
AND R1, R1, #0    ; R1 ← 0
```

- Initialize to a specific (small) value
  - The LC-3 does not have an instruction that copies an immediate value to a register...

```
AND R3, R3, #0    ; R3 ← 0   clear first
ADD R3, R3, #7    ; R3 ← 7   Add constant
```

  - Only works for values in range [-16,15] because the immediate is a sign-extended 5-bit number (**imm5**)
    - Larger values can be loaded from memory

Register-Register Copy (3)

- It is often useful to copy a value from one register to another
  - Most ISAs have a "MOVE" instruction…
- Can do this several ways…
  - Add zero to the source register

```
ADD R1, R2, #0    ; R1 ← R2
```

  - AND the source register with itself

```
AND R1, R2, R2    ; R1 ← R2
```

  - AND the source register with all 1s (xFFFF)

```
AND R1, R2, #-1   ; R1 ← R2
```

Increment or Decrement (4)

- Use the ADD with Immediate to add/subtract small numbers to/from a value in a register

(+1)
```
ADD R7, R7, #1    ; increment R7
```

(-1)
```
ADD R6, R6, #-1   ; decrement R6
```

- The '#' symbol indicates that the number is given in decimal; 'x' indicates hex, 'b' indicates binary
  - Can represent 12_{10} as #12, xC, or b1100

Negation and Subtraction (5) (6)

- Remember how to negate a 2's-complement number: flip all the bits (bitwise NOT) and add 1

negation

```
NOT R6, R6          ; negate step 1
ADD R6, R6, #1      ; negate step 2
```

- The LC-3 ISA does not have a subtract instruction, so we add the negation

subtraction

```
NOT R0, R4          ; R0 ← -R4
ADD R0, R0, #1
ADD R0, R0, R3      ; R0 ← R3-R4
```

2's complement negation

Add the value we are subtracting from

Masking (7)

- Sometimes we need to know the value of a particular bit within a word
- We can determine if a number is odd or even by "masking" all bits except bit 0

```
; mask R5 except bit 0; 16-bit result
; equals 1 if R5 was odd, 0 if even
AND R5, R5, #1      ; mask = b0000…0001
```

- Or we may want to clear only a particular bit

```
AND R2, R2, #-3     ; mask = b1111…1101
```

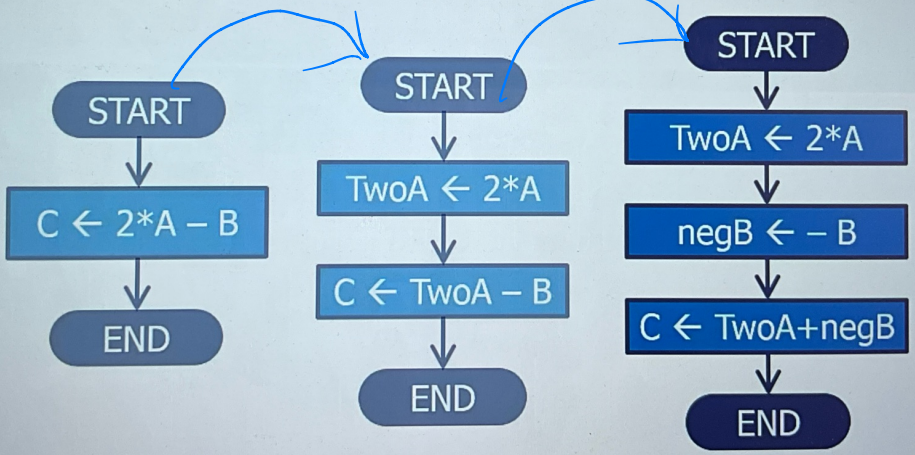
- Useful with Control instructions that let the program do different things based on the result

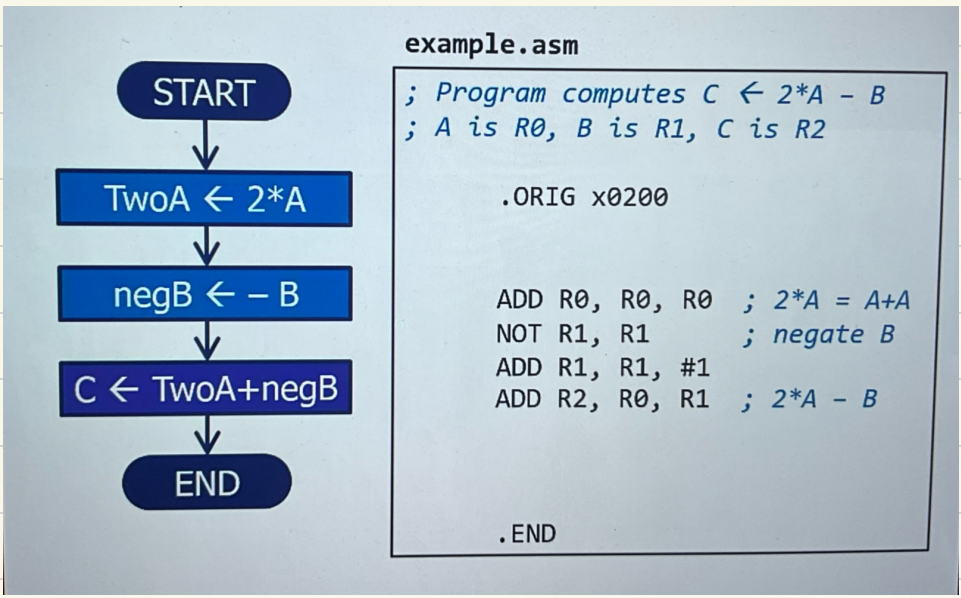
And Last Bit, 1 is odd
0 is even
(not affecting other bits)

Clear the bit position 1 to 0

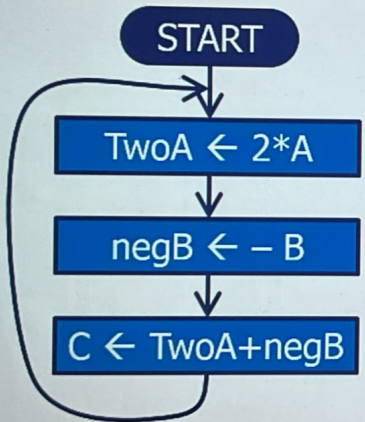
Example Program: C ← 2*A – B

- Refine the problem by decomposing it into a sequence of smaller (easier) steps





Example Program: C ← 2*A – B



example.asm

```
; Program computes C ← 2*A - B
; A is R0, B is R1, C is R2

        .ORIG x0200

START
        ADD R0, R0, R0   ; 2*A = A+A
        NOT R1, R1       ; negate B
        ADD R1, R1, #1
        ADD R2, R0, R1   ; 2*A - B

        BR START    ; repeat forever

        .END
```

(Makes the program not wandering off, and repeat the program forever for better testing)