The LC-3 Computer

CS 350: Computer Organization & Assembler Language Programming

10/14: pp.9,15,16; 10/12: Solved

A. Why?

- We'll be writing machine and assembler programs using the LC-3.
- Instruction set architectures (and the LC-3 in particular) have different ways to specify operands.
- Data movement and calculation are two of the basic kinds of instructions.
- Control (branch and jump) instructions let us implement decisions and loops.
- Trap instructions let us access operating-system-level routines like I/O.

B. Outcomes

At the end of this lecture, you should know

- The basic architecture of the LC-3: word size, number of registers, datatypes, etc.
- How the `PC`-offset, Base-Offset, Indirect, and Immediate addressing modes work.
- How its data movement and calculation instructions work.
- How the branch and jump instructions work.
- How to use the `TRAP` instruction to read/write a character or halt the program.

C. The LC-3 Computer

- Patt & Patel book uses the Little Computer version 3
- **16-bit addresses** (64K memory locations), 16-bit word at each location
- **2's complement** integers
- **8 data registers** (named `R0` – `R7`; 3 bits to name a register)
  - Temporary storage. Register access takes 1 machine cycle;
  - Memory access generally takes > 1 cycle.
- **3 condition code bits** for tests (we'll see this later).

- **4-bit opcodes** (16 instructions). The opcode is always the leftmost 4 bits. There are three kinds of instructions:
  - **Data movement**: **Load** value into register or **store** value from register). [Remember, it's never "load into memory" or "store into register".]
  - **Calculation/Data Operation** (`ADD`, e.g).
  - **Control/Branch/Jump**: By default, execution proceeds sequentially through memory. These instructions change the `PC` so that the next instruction can be somewhere else. Used for decisions and loops.
- The LC-3 has 5 **addressing modes** (5 ways to specify an operand)
  - **Immediate** (contained in instruction)
  - **Register** (number 000, 001, …, 110, 111)
  - Three ways to specify memory locations: **Base-offset**, **PC-offset**, and **Indirect**
  - Base-offset and `PC`-offset are also known as Base-relative and `PC`-relative.
- Not every LC-3 instruction supports every addressing mode.
  - Instruction set does not have an "orthogonal" design.
- Compare the LC-3 with the Simple Decimal Machine from last time.
  - They differ in address size and addressability (word size), radix, number of registers, condition code (SDC doesn't have one), and number of opcodes.
  - The biggest difference is that the SDC uses **Absolute addresses** (the whole address is written out as part of the instruction).
  - We can't have absolute addresses on the LC-3 because we have 16-bit addresses and 16-bit instructions.

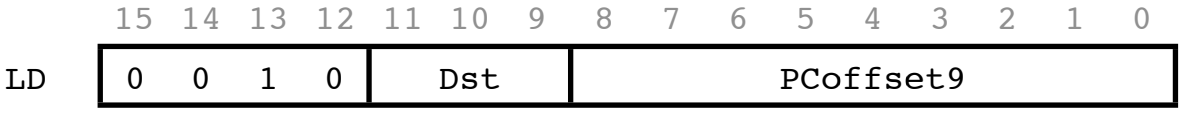
D. Data Movement Using PC-Relative Addressing Mode

- The 3 instructions that use `PC`-relative addressing all have the basic format: 4 bits of opcode, 3 bits of register number, and 9 bits of `PC` offset ($-256 \leq$ `PC` offset ≤ 255) to specify one memory operand.
- The effective address of the operand = `PC` + sign-extended 9-bit offset

- The PC was incremented as part of the FETCH phase of the current instruction cycle, so at when we reach the EVALUATE ADDRESS phase, the PC **already points to the next instruction**.
- So a PC offset of 0 means the next instruction, a PC offset of 1 means the instruction after that, etc.
- A PC offset of –1 means **this instruction**, a PC offset of –2 means the instruction before this one, etc.

Load instruction (LD)

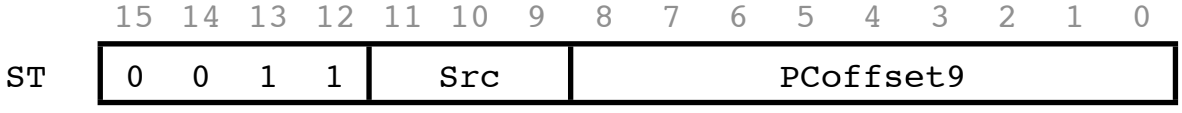
- Loads a register with the value of the memory location at the specified address.
- Has a destination register; uses PC-relative addressing with 9-bit offset
  - *Destination register* ← M[PC + *offset*]



- **Example 1**: Instruction at x2FFF = 0010 011 111111000, with M[x2FF8] = x4A30
  - R3 ← M[x3000 – 8] = M[x2FF8] = x4A30

Store instruction (ST)

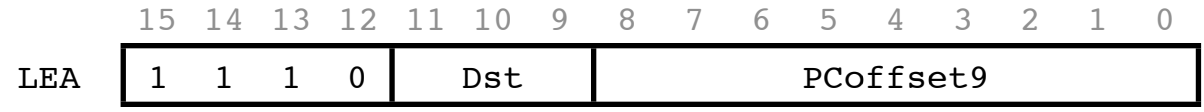
- Opposite direction of Load: Store value of a source register into memory.
  - M[PC+*offset*] ← *Source register*



- **Example 2**: Instruction at x2FFF = 0011 011 111111000
  - M[x3000 – 8] = M[x2FF8] ← R3

Load Effective Address (LEA)

- Load a register with the **address** of the memory operand (not the value stored at the address). Similar to Load but doesn't actually access memory.
  - *Destination register* ← PC+*offset* (not M[PC+*offset*])



- **Example 3**: Instruction at `x2FFF` = `1110 011 111111000`
  - `R3` ← `x3000 - 8` = `x2FF8`

Simple Assembler Formats

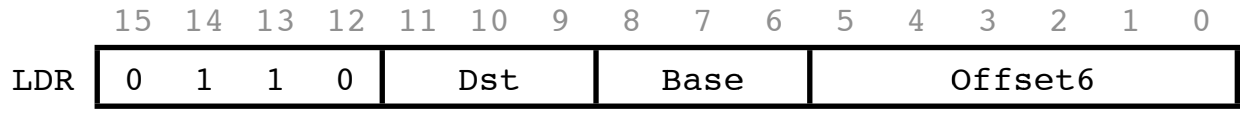
- It's tedious to write programs in binary, so we'll be using a more mnemonic technique called **Assembler Language**.
- The simplest way to write the `PC`-offset instructions uses the mnemonic op code (`LD`, `ST`, or `LEA`), the register number (written `R0`, `R1`, …, or `R7`), comma, and the `PC` offset as a signed integer constant or as a hex constant.
- For the three examples above,
  - The `LD` instruction `0010 011 111111000` is written `LD R3, -8`
  - The `ST` instruction `0011 011 111111000` is written `ST R3, -8`
  - The `LEA` instruction `1110 011 111111000` is written `LEA R3, -8`
  - In general, we can write constants in decimal or hex, but the assembler won't take `x1F8` as a representation of `-8` (it wants to read `x1F8` as a 12-bit string).

E. Base-Offset Addressing Mode

- With `PC`-offset addressing, we can only reference locations +255 or –256 from the `PC`.
- In Base-Offset addressing, the base register contains a 16-bit address. (It points to a location.) It can be any address, so we're not limited to locations close to the instruction.
  - We add 6 bits of sign-extended offset to the address in the base register
  - The effective address = Base register + offset

LDR (Load Using Base Register)

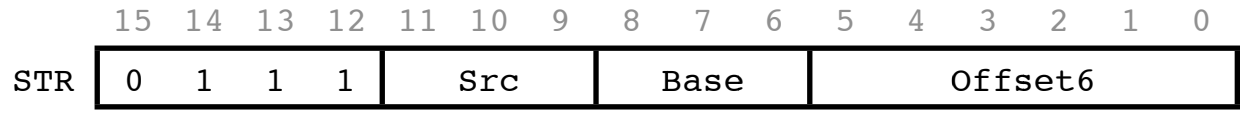
- *Destination register* ← `M`[*Base register* + *offset*]



- **Example 4**: Instruction `0110 011 111 111000`, `R7 = x320C`, `M[x3204] = 38`
  - (Because this is a base-offset instruction, the `PC` doesn't matter.)
  - `R3` ← `M[R7 – 8] = M[x320C – 8] = M[x3204] = 38`
  - The simple assembler format is `LDR R3, R7, -8`

STR (Store using base Register)

- `M`[*Base register* + *offset*] ← *Source Register*



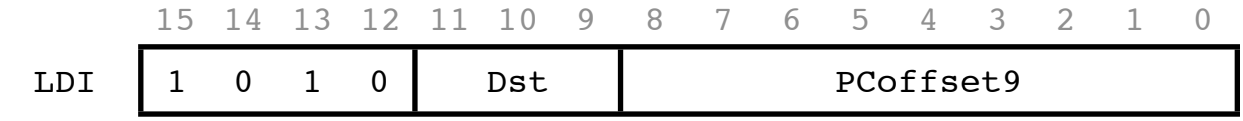
- **Example 4**: Instruction `0111 011 111 111000, R7 = x320C, R3 = 18`.
  - (Because this is a base-offset instruction, the `PC` doesn't matter.)
  - `M[R7 – 8] = M[x320C – 8] = M[x3204]` ← `R3 = 18`
  - The simple assembler format is `STR R3, R7, -8`

F. Indirect Addressing Mode

- The third addressing mode also uses a pointer, which again lets us access any location in memory.
- This time, the address is stored in memory: Effective address = `M[PC` + offset]
- Compare with `PC`-relative addressing: Effective address = `PC` + offset.

Load Indirect (LDI)

- *Destination Register* ← `M[M[PC` + *offset*]]

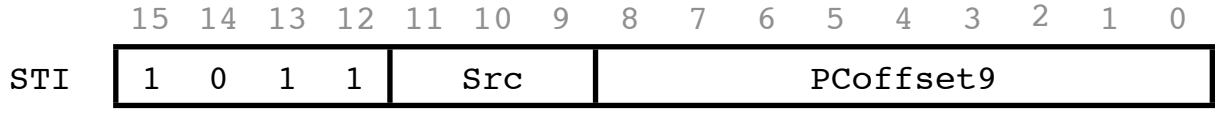


- **Example 5**: Instruction at `x2FFF = 0010 011 111111000, M[x2FF8] = x4A30, M[x4A30] = 15`
  - `R3` ← `M[M[x3000 – 8]] = M[M[x2FF8]] = M[x4A30] = 15`

- The simple assembler format is `LDI R3, -8`

Store Indirect (STI)

- `M[M[PC` + *offset*`]]` ← *Source Register*



- **Example 6**: Instruction at `x2FFF = 0011 011 111111000, M[x2FF8] = x4A30`
  - `M[M[x3000 - 8]] = M[M[x2FF8]] = M[x4A30]` ← `R3 = 24`
  - The simple assembler format is `STI R3, -8`

G. A Larger Example

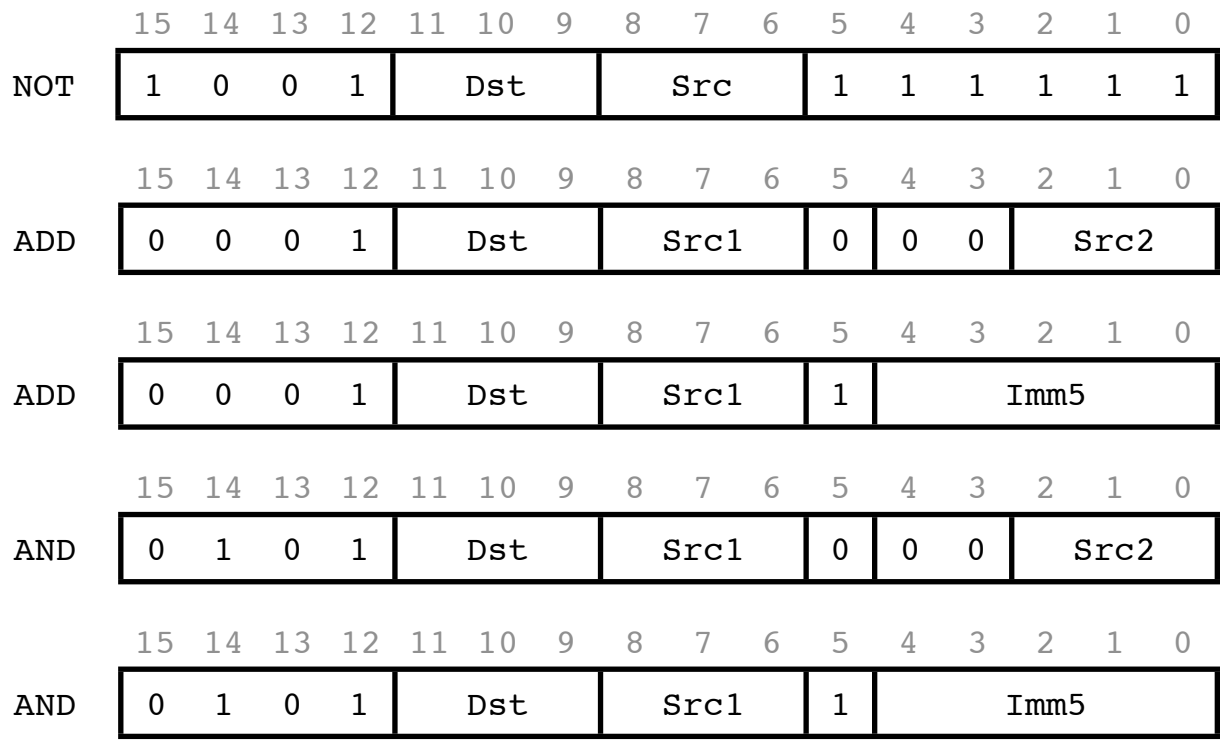
- **Example 7**:

Addr	Contents	Asm	Action
x3000	0010 110 000000001	LD R6, 1	R6 ← M[PC+1] =M[x3001+1]= M[x3002]=0011 0110 0000 0000
x3001	0010 011 111111111	LD R3, -1	R3 ← M[PC-1] = M[x3002-1] = M[x3001]=0010 0111 1111 1111
x3002	0011 011 000000000	ST R3, 0	M[PC+0] = M[x3003+0] ← R3
x3003	0011 100 000000000	ST R4, 0	(Gets overwritten with 0010 0111 1111 1111)
x3004	0011 110 011111111	ST R6, 255 or ST R6,xFF	M[PC+255] = M[x3005+xFF] = M[x3104] ← R6
x3005	1110 000 111111110	LEA R0, -2	R0 ← PC+(-2) = x3006-2 = x3004

H. Calculation/Data Operation Instructions

- On the LC-3, these instructions do not reference memory.
- For `NOT`, the destination (`Dst` below) and source operands (`Src`) are registers.
- For `ADD` and `AND`, the destination and the left operand (`Src1`) are registers.
  - The right operand is either a register or a value that's hard-wired into the instruction as a immediate value (5-bit signed number): –16 to +15
  - Bit 5 is used as a flag = "Do we have an immediate argument?"

- So ADD immediate is a great way to increment or decrement a register by a small positive or negative value but if you want a large value, you need to do something else.



NOT, AND, ADD Examples

- Below, the simple assembler version of the instruction is shown after the binary representation.
- **Example 8**: R3 ← NOT R2 = NOT 30 = -31
  - 1001 011 010 111111 NOT R3, R2
  - Note NOT(*nbr*) = -(*nbr*) - 1 in 2's complement.
- **Example 9**: R1 ← R0 AND x0000 = 0 (typical way to set a register to zero).
  - 0101 001 000 1 00000 AND R1, R0, 0
  - The simple assembler format specifies a constant for the right-hand (immediate) operand; the assembler sets bit 5 = 1 accordingly.
- **Example 10**: R0 ← R2+R3
  - 0001 000 010 000 011 ADD R0, R2, R3

  - This simple assembler format specifies a register as the right-hand operand. Again, bit 5 is set (= 0), this time to indicate a register operand. (Plus it sets the unused bits 3 and 4 to 0.)

- **Example 11**: `R7 ← R7+R7`
  - `0001 111 111 000 111  ADD R7, R7, R7`
- **Example 12**: `R0 ← R2+3`
  - `0001 000 010 1 00011  ADD R0, R2, 3`
- **Example 13**: `R1 ← R1+15 = 15`
  - `0001 001 001 1 01111  ADD R1, R1, 15`
- **Example 14**: `R6 ← R6 AND 0` (sets `R6` to 0). This is like Example 9 but uses the same register as the left-hand operand.)
  - `0101 110 110 1 00000  AND R6, R6, 0`
- **Example 15**: `R2 ← R1 ADD 0` (in effect, we copy `R1` to `R2`)
  - `0001 010 001 1 00000  ADD R2, R1, 0`
- **Example 16**: `R2 ← R1 AND R1` (another way to copy `R1` to `R2`)
  - `0101 010 001 000 001  AND R2, R1, R1`

Instructions We Don't Have

- We don't have separate instructions for:
- **Subtraction**
  - For `X – small constant`, use `ADD` immediate of negative of constant.
  - More generally, for `X – Y`, use `X + NOT Y + 1`
- **Logical OR:**
  - For `X OR Y`, use `NOT (NOT X AND NOT Y)`
  - For any particular bit position, if you want to set `X`[k] ← `X`[k] `OR Y`[k] and you know `X`[k] is 0, then `0 OR Y`[k] equals `0 + Y`[k], so you can select `Y`[k] with the mask with 1 at position k and add the result to `X`: `X ← X +` (`Y & 2`^{k}).

- **Setting a register to zero**
  - Use *register* ← *register* `AND` 0…0.
- **Copying one register to another:**
  - Don't try `LDR`: It means "Load using Base Register," not "Load from a register."
  - I know of three ways to copy one register value to another register:
    - *Destination register* ← *Source register* `ADD 0`
    - *Destination register* ← *Source register* `AND` *Source register*
    - *Destination register* ← *Source register* `AND` 16 `1` bits
    - The immediate field needs to be `11111` (i.e., `-1`); before the `AND` is done, the 5 bits `1....` are sign-extended to 16 bits.

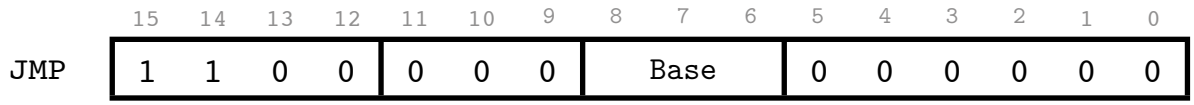
Example 17 (Another larger example) [Bit of reformatting 10/14]

Addr	*Instruction*	*Asm*	*Action*
x30F6	1110 001 111111101	LEA R1, -3	R1 ← PC-3 = x30F7-3 = x30F4
x30F7	0001 010 001 1 01110	ADD R2,R1,14	R2 ← R1+14 = x30F4+14 = x3102
x30F8	0011 010 111111011	ST R2, -5	M[PC-5] ← R2 M[x30F4] ← x3102
x30F9	0101 010 010 1 00000	AND R2, R2, 0	R2 ← R2 AND 0 (= 0)
x30FA	0001 010 010 1 00101	ADD R2, R2, 5	R2 ← R2+5 = 0+5 = 5
x30FB	0111 010 001 001110	STR R2, R1, 14	M[R1+14] ← R2 M[x3102] ← 5
x30FC	1010 011 111110111	LDI R3, -9	R3 ← M[M[PC-9]] = M[M[x30FD-9]] = M[M[x30F4]] = M[x3102] = 5

I. Control Instructions

- Control instructions alter the sequence of instructions being executed and let us go to other instructions.
  - They work by changing the PC during the **EXECUTE INSTRUCTION** phase of instruction cycle.
  - Compare with the PC change done for every instruction during the **FETCH INSTRUCTION** phase of the instruction cycle.

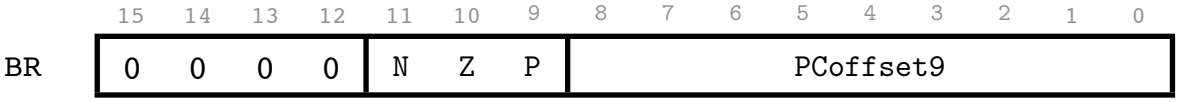
- **Short and Long-distance go-to**
  - On LC-3, the `BR` (branch) instruction specifies the target address using PC-offset so you can only do a short-distance jump with one.
  - The `JMP` (jump) instruction specifies the target using a base register, so you can jump anywhere ("long-distance" jump).
- **Conditional and Unconditional go-to**
  - On LC-3, `JMP` is unconditional; `BR` is conditional (though "always" and "never" are possible conditions).
- **Jump instruction**: The value in the base register is used as the new PC value; i.e., we go to the address indicated by the base register. Before the copy, the `PC` points to the next instruction. After we copy the new address to the `PC`, the next fetch instruction will get the instruction at this new address.
  - `PC` ← *Base Register*



J. Branch instruction

- The Branch instruction `BR` is a short-distance conditional go-to. The target of the go-to is calculated using PC-offset addressing (so about ± 255 addresses).
- The branch instruction `BR` contains a **mask** field, which it combines with the current **condition code** to decide whether or not to do a go to.
- **Condition code**: There is a three-bit condition code register `CC`; its value is `100`, `010`, or `001`. The three bits are named `N`, `Z`, and `P` (left-to-right), short for `Negative`, `Zero`, and `Positive`, and exactly one of `N`, `Z`, and `P` is one at any given time. "`CC` is `N`" or "`N = 1`" means `CC = 100`, etc.

- The condition code is set automatically. On boot-up, `Z = 1`. Every time we do a load or calculation instruction (`LD`, `LDI`, `LDR`, `LEA`, `NOT`, `ADD`, or `AND`), the LC-3 checks the value being copied to the destination register and sets `N`, `Z`, or `P` accordingly.*
- **Mask**: The `BR` instruction contains a three-bit mask; its bits correspond to the `NZP` bits of the condition code. To execute a `BR` instruction, the `CC`'s three `NZP` bits are bitwise ANDed with the instruction's three `NZP` mask bits. If the result ≠ `000`, the `PC` is set to `PC` + *offset*, so that we'll go the instruction at that address. (If the result is `000`, the PC is not changed and execution continues with the next instruction.)
  - If (`CC & IR`[11:9]) ≠ `000` then `PC` ← `PC` + *offset* [10/12]



- **Versions of Branches:** We can control the kind of branch we want to do by how we set the branch mask. E.g., if the mask is `011`, then we will branch if the `CC` is `010` or `001` (but not `100`). So we branch if the `CC` is `Z` or `P`, hence "branch if ≥ 0".
- Note if the mask is `111`, we always jump; if the mask is `000`, we never jump.
- There are mnemonic codes for the 8 possible three-bit masks. In general, you concatenate `BR` with some combination of `N`, `Z`, or `P` (in that order).
- In the table below, "`N`" means "**negative**" not "not," so `BRNZ` means "branch if negative or zero." To get branch on not zero, you use `BRNP` ("branch on negative or positive). The mnemonic "`NOP`" means "no operation" — no goto is done. For unconditional branch you can use `BR` or `BRNZP` (your choice).

---

* Technical note (won't make sense until we see the `TRAP` and subroutine call instructions): `HALT` sets `CC` to `P`; the other `TRAP`s set the `CC` by loading the return address into `R7` (addresses `x8000`, ..., `xFFFF` are treated as negative).

Mask	*Mnemonic*	*Branch Condition*
000	NOP	Never
100	BRN	< 0
010	BRZ	= 0
001	BRP	> 0
110	BRNZ	≤ 0
101	BRNP	≠ 0
011	BRZP	≥ 0
111	BR or BRNZP	Unconditional

K. Examples of Using Branch Instructions

If-then Statement

- For an **`if-then`** statement, we'll do a test and either fall into the true branch code or jump around the true branch code, so the test we do will actually be the opposite of the **`if`** condition.
- **Example 18**: Say we want `if R1 > 0 then` …., where the code should begin at `x3015` and the true branch contains $42_{10} = 2A_{16}$ instructions.
  - Below, the if test and branch take up two instructions (at x3015 and x3016), so the true branch code is at x3017 – x3040 (since x3040 = x3017 + 2A – 1). Thus, the `BR` at `x3016` needs a PC offset of `x3041 – x3017 = x2A`.

Addr	*Value*	*Asm*	*Action/Comment*
x3015	0001 010 001 1 00000	ADD R1,R1,0	Test R1
x3016	0000 110 000101010	BRNZ x2A	if R1 <= 0, go to x3041
x3017	...		(42 instructions for
...	...		then branch)
x3041	...		(Code after if-then)

While Loop

- A **`while`** loop is like an **`if-then`** statement except that at the end of the loop body, we branch back up to the top of the loop (the **`while`** test).
- **Example 19**: Say we want **`while`** `R1 > 0 { ... }`, our code should begin at `x3015` and the loop body takes 42 instructions.
  - The first instruction below (`R1 ← R1`, at `x3015`) just sets the condition code according to the value of `R1`. (If `R1` hasn't been initialized, we could just LD it instead.)
  - The branch at `x3016` goes to `x3042`, the first instruction after the loop, so it needs an offset of `x3042 - x3017` = $2B_{16} = 43_{10}$.
  - The branch at `x3041` has to go back to `x3016`, so it needs an offset of `x3042 - x3016` = $-2C_{16} = -44_{10}$. (It turns out the LC-3 assembler requires the minus sign **after the x** for a negative hex constant, hence `x-2C` for `-2C`.)
  - This code assumes that the loop body ends with the condition code set according to the value in `R1`. That way, since the unconditional branch to the top of the loop doesn't change the condition code, the `BRNZ` test relates to `R1`.
  - If the loop body doesn't end with the condition code set according to `R1`, the unconditional branch can go to `x3015`, which makes us test `R1`.

Addr	*Value*	*Asm*	*Action/Comment*
x3015	0001 001 001 1 00000	ADD R1,R1,0	Test R1
x3016	0000 110 000101011	BRNZ x2B	Top: if R1 ≤ 0, go to x3042
x3017 ...	... ...		(Loop body's 42 instr)
x3041	0000 111 111010010	BR x-2C	go to x3016 (top of loop)
x3042	...		(Code after loop)

If-Else Statement

- For an **`if-else`** statement, we need two branch instructions.
  - If the test fails, `BR` around the true branch to the beginning of the false branch.

- If the test succeeds, fall into the true branch code; at the end of the true branch, BR around the code for the false branch.

- **Example 20**: Say we want **`if`** `R1 > 0` **`then`** …. **`else`** …, where the code should begin at `x3015`, the true branch ends at `x3040`, and the false branch ends at `x3050`. Note the branch at `x3016` needs PC offset `x3042 – x3017 = x2B`; the branch at `x3041` needs offset `x3051 – x3042 = xF`.

Addr	*Value*	*Asm*	*Action/Comment*
x3015	0001 001 001 1 00000	ADD R1,R1,0	Test R1
x3016	0000 110 000101011	BRNZ x2B	if R1 <= 0, skip true br.
x3017 ...	... ...		(true branch; 42 instr)
x3041	0000 111 000001111	BR 15	skip false branch
x3042 ...	... ...		(false branch; 16 instr)
x3051			(Code after if-else)

Implementing A Loop that Executes N times:

- To do something *N* times, it's easiest to use a loop with a counter = *N*, *N* – 1, *N* – 2, …, 1. The counter will be contained in some temporary register, and the loop body will end with a decrement of the counter register.

```
counter = N
while (counter ≥ 0) {
    … Loop Body …
    --counter;
}
… Code after the loop …
```

- **Example 21**: Say we want **`for`** `(R2 = N; R2 >= 0; R2--) {` ... `}`, our code should begin at `x3015` and the loop body takes 42 instructions.

  - The code is similar to that for Example 19 except for the addition of the decrement of `R1` after the rest of the loop body.

Addr	*Value*	*Asm*	*Action/Comment*
x3015	0001 001 001 1 00000	ADD R1,R1,0	Test R1
x3016	0000 110 000101100	BRNZ x2C	Top: if R1 ≤ 0, go to x3043
x3017 ...	... ...		(Loop body's 42 instr)
x3041	0001 001 001 1 11111	ADD R1,R1,-1	R1--
x3042	0000 111 111010011	BR x-2D	go to x3016 (top of loop)
x3043	...		(Code after loop)

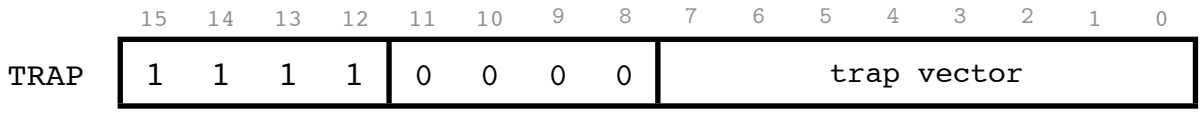
- There are a couple of ways to have an up-going counter 0, 1, …, *N*–1. (Neither are particularly elegant.)
  - Maintain the up-going counter in parallel with a down-going counter *N*, *N*–1, … 0. A sketch of the C code would be

    ```
    for (R1 = N, R2 = 0; R1 >= 0; R1--, R2++) { ... }
    ```

  - Have the up-going counter go through –*N*, –*N*+1, …, –1.

L. TRAP Instruction: Call the Operating System

- The TRAP instruction calls a **service routine** (an operating system routine). It's like calling a subroutine that's owned by the operating system. Control jumps to some code to handle the trap, and when that code finishes, control jumps to the instruction after the TRAP instruction (**Exception**: We don't return after the HALT trap or any illegal TRAP.)
- Service routines are identified by an 8-bit **trap vector**. The hardware uses the trap vector to figure out where the OS code for that particular service is: The location of the code to handle trap *T* is at memory location *T*. (The address of the code to handle TRAP x20 is in M[x20], etc.) The table of TRAP handler addresses (x0000 – x00FF) is called the TRAP table.



- The pseudocode for executing the TRAP instruction is below. (We'll discuss supervisor mode vs user mode execution when we look at interrupts.) [10/14]

- `R7` ← `PC` // Save location to return to (see below)
  `PC` ← `M`[*trap vector*] // Look up location of `TRAP` code; go there
  Set execution mode to "system"[†]

- The `R7` ← `PC` saves the location after the `TRAP` instruction. At the end of the `TRAP`-handling code, there's a `JMP R7` instruction that jumps ("**through**" `R7`) back to the user's code.
- The simple assembler instruction is `TRAP` *n* where *n* is the trap code, typically a hex number:
  - `x20`: `GETC`: Input a character from keyboard into the rightmost byte of `R0` (and clear the leftmost byte).
  - `x21`: `OUT`: Output the character in the rightmost byte of `R0` to the monitor (and ignore the leftmost byte).
  - `x22`: `PUTS`: Display the null-terminated string pointed to by `R0`.
  - `x23`: `IN`: Like `GETC` but prints a message before reading the character.
  - `x25`: `HALT`: Halt execution. (Clears the CPU's *running* flag.)
- **A note on TRAPs vs subroutines**: When we start writing subroutines, we'll use `R7` the same way that `TRAP` does to return to the calling routine. If our subroutine code calls a `TRAP`, it will overwrite the `R7` value we need to return to our caller. We'll have to save our `R7` before the `TRAP` and restore it later.

Example: Read and Echo a Character

- Here's code to prompt for input with "> ", read one character, print it back out, and halt. `GETC` doesn't echo its input, so we print the character read in so that the user sees it after pressing the key. [table added 10/14]

Addr	*Value*	*Asm*	*Action/Comment*
x3000	1110 000 000000100	LEA R0,4	Pt R0 to prompt string
x3001	1111 0000 0010 0010	TRAP x22	PUTS (print prompt)
x3002	1111 0000 0010 0000	TRAP x20	GETC (read char into R0)
x3003	1111 0000 0010 0001	TRAP x21	OUT (print char in R0)

[†] System/user execution mode will be discussed later.

Addr	*Value*	*Asm*	*Action/Comment*
x3004	1111 0000 0010 0101	TRAP x25	HALT
x3005	0000 0000 0011 1110		Prompt: '>' = x3E
x3006	0000 0000 0010 0000		' '
x3007	0000 0000 0000 0000		end of prompt

The LC-3 Computer, Part 1

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Instruction set architectures (and the LC-3 in particular) have different ways to specify operands, each with its advantages and disadvantages.
- Data movement and calculation are two of the basic kinds of instructions.
- Control instructions let us implement decisions and loops.
- Trap instructions let us access operating-system-level routines like I/O.

B. Outcomes

After this activity, you should be able to:

- Be able to hand-execute basic data movement and calculation instructions for the LC-3.
- Be able to distinguish between `PC`-relative, base offset, and indirect addressing.
- Write LC-3 branch instructions to implement a loop or decision.
- Use the `TRAP` instruction to read/write a character or halt the program.

C. Problems

1. What is the range of offset values we can use with `LD` *reg*, *offset*? What about `ST` and `LEA`?
2. What is the range of immediate values we can use in `ADD` and `AND`?

For Problems 3–6, fill out the missing table entries below. (Problem 6 is the entry for address `x3000`, Problem 7 for `x3001`, etc.) Assume execution starts at `x3000` and that all registers contain unknown values.

Addr	*Instruction*	*Asm*	*Action*
`x3000`	`1110 010 001001111`	`LEA ???`	`R2 ← PC + x4F = x?....?`

Addr	*Instruction*	*Asm*	*Action*
x3001	0010 101 111111111	LD ???	R5 ← M[PC - 1] = x2??? [3 digits missing]
x3002	0011 100 000001100	ST ???	?...?
x3003	1110 011 ?????????	LEA ???	R3 ← PC - 8 = x3004 - 8 = x2FFC

For Problems 7 – 12, repeat the previous problem using the table below. Again, assume execution starts at x3000 and that all registers contain unknown values.

Addr	*Value*	*Asm*	*Action*
x3000	1010 000 000111111	LDI ???	R0 ← M[M[PC+x3F]] = M[M[x3001+x3F]] = M[M[???]] = M[???] = ???
x3001	1011 000 000111111	STI ???	M[M[PC+x3F]] = M[M[x3002+x3F]] = M[M[???]] = M[???] ← R0 = ???
x3002	0010 001 000111101	LD ???	R1 ← M[PC+x3D] = M[???+x3D] = M[???] = ???
x3003	0110 010 001 000000	LDR ???	R2 ← M[R1+0] = M[???] = ???
x3004	0001 001 001 1 00001	ADD ???	R1 ← R1+1 = ???+1 = ???
x3005	0111 010 001 000000	STR ???	M[R1] = M[???] ← R2 = ???
...			
x3040	x4000		
x3041	x4002		
x4000	x00AB		
x4001	x3210		
x4002	xABCD		

For Problems 13 – 23, find the corresponding description (a) – (j). You might find multiple instructions described the same way, and you might find some descriptions don't have a corresponding instruction.

13. ADD R_1 R_2 1 00000
14. AND R_1 R_2 1 00000
15. ADD R_1 R_2 1 00001
16. AND R_1 R_2 1 00001
17. ADD R_1 R_2 1 11111
18. AND R_1 R_2 1 11111
19. ADD R_1 R_2 000 R_2
20. AND R_1 R_2 000 R_2
21. ADD R_1 R_2 000 000
22. AND R_1 R_2 000 111
23. NOT R_1 R_2 11111

a. $R_1 \leftarrow R_2[0]$
b. $R_1 \leftarrow -R_2$
c. $R_1 \leftarrow -R_2 - 1$
d. $R_1 \leftarrow 0$
e. $R_1 \leftarrow 2 * R_2$
f. $R_1 \leftarrow R_2$
g. $R_1 \leftarrow R_2 + \mathrm{R0}$
h. $R_1 \leftarrow R_2$ & R7
i. $R_1 \leftarrow R_2 - 1$
j. $R_1 \leftarrow R_2 + 1$

In Problems 24 – 27, the number of question marks doesn't necessarily indicate anything about the length of the answer.

24. Fill in the missing pieces of the (silly) instruction sequence. What does it do? (It depends on what's in R1.)

Addr	*Value*	*Asm*	*Action/Comment*
x3000	0101 011 001 1 11111	AND R3,R1,-1	???
x3001	0000 011 000000111	BRZP ???	If ??? then ???
x3002	0000 010 000000010	BRN ???	else ???
x3003	...	...	(Do we ever reach this instruction?)

25. Implement "`if R0 < 0 then R0 ← 0 else M[x30AC] ← R0`" by filling in the missing pieces of the instructions.

Addr	*Value*	*Asm*	*Action/Comment*
x4000	0001 000 000 ??????	ADD R0,R0,???	Test value of R0
x4001	0000 011 ?????????	BRZP ???	If R0 ≥ 0, go to false br.
x4002	0101 000 ?????????	AND R0,R0,0	. R0 ← 0
x4003	0000 ??? 000000001	BR??? 1	Skip over false branch
x4004	0011 000 ?????????	ST R0,???	. M[x40AC] ← R0
x4005	...		(code after if-else)

26. Implement "`if R7 = 1 then go to x5000`". (`R1` is a temporary register.)

Addr	*Value*	*Asm*	*Action/Comment*
x8000	0001 001 ?????????	ADD R1,R7,-1	R1 ← R7 – 1
x8001	0000 ??? 000000011	BR?? ???	If R7 ≠ 1, go to end if
x8002	0010 001 000000001	LD R1,???	. R1 ← Target location
x8003	1100 000 001 00000	JMP R1	. Jump to target
x8004	x5000		Location of target
x8005	...		(code after if-then)

26. Implement "if `R0 ≤ 0` then go to the location pointed to by `x3150`; else if `R0 = 1` go to `x3160`; else go to `x3165`". (`R1` is a temporary register.)

Addr	*Value*	*Asm*	*Action/Comment*
x3100	0001 000 ?????	ADD R0,???,0	Test R0
x3101	0000 001 ???	BRP ???	if R0 ≤ 0 then
x3102	0010 000 ???	LD R0,???	. Get loc pt'd to by x3150
x3103	1100 000 000 00000	JMP R0	. Jump to that location
x3104	0001 001 000 ????	ADD R1,R0,-1	else R1 ← R0 - 1
x3105	0000 ?????	BRZ ???	. if R0 = 1 go to x3160
x3016	000 111 ?????	BR ???	. else go to x3165
...			(x49 words)
x3150	(address to jump to)		

Solution to Activity 10

1. –256 through +255

2. –16 through 15

For Problems 3 – 6 and 7 – 10, the added information is shown **in bold**.

Addr	*Instruction*	*Asm*	*Action*
x3000	1110 010 001001111	LEA **R2,x4F**	R2 ← PC + x4F = **x3001+x4F = x3050**
x3001	0010 101 111111111	LD **R5,-1**	R5 ← M[PC – 1] = **M[x3002-1]** = **M[x3001] = 0010 101 111111111** = **2BFF**
x3002	0011 100 000001100	ST **R4,12**	**M[PC+12] = M[x3003+xC]** = **M[x300F] ← R4**
x3003	1110 011 111111000	LEA **R3,-8**	R3 ← PC – 8 = x3004 – 8 = x2FFC

Addr	*Value*	*Asm*	*Action*
x3000	1010 000 000111111	LDI **R0,x3F**	R0 ← M[M[PC+x3F]] = M[M[x3001+x3F]] = M[M[**x3040**]] = M[**x4000**] = **x00AB**
x3001	1011 000 000111111	STI **R0,x3F**	M[M[PC+x3F]] = M[M[x3002+x3F]] = M[M[**x3041**]] = M[**x4002**] ← R0 = **x00AB**
x3002	0010 001 000111101	LD **R1,x3D**	R1 ← M[PC+x3D] = M[**x3003**+x3D] = M[**x3040**] = **x4000**
x3003	0110 010 001 000000	LDR **R2,R1,0**	R2 ← M[R1+0] = M[**x4000**] = **x00AB**
x3004	0001 001 001 1 00001	ADD **R1,R1,1**	R1 ← R1+1 = **x4000**+1 = **x4001**
x3005	0111 010 001 000000	STR **R2,R1,0**	M[R1] = M[**x4001**] ← R2 = **x00AB**
...			
x3040	x4000		
x3041	x4002		
...			
x4000	x00AB		
x4001	x3210		**becomes x00AB**
x4002	xABCD		**becomes x00AB**

Instruction goes with ——————→	*Action*
13. ADD R_1 R_2 1 00000	f. $R_1 \leftarrow R_2$
14. AND R_1 R_2 1 00000	d. $R_1 \leftarrow 0$
15. ADD R_1 R_2 1 00001	j. $R_1 \leftarrow R_2 + 1$
16. AND R_1 R_2 1 00001	a. $R_1 \leftarrow R_2[0]$
17. ADD R_1 R_2 1 11111	i. $R_1 \leftarrow R_2 - 1$
18. AND R_1 R_2 1 11111	f. $R_1 \leftarrow R_2$
19. ADD R_1 R_2 000 R_2	e. $R_1 \leftarrow 2 * R_2$
20. AND R_1 R_2 000 R_2	f. $R_1 \leftarrow R_2$
21. ADD R_1 R_2 000 000	g. $R_1 \leftarrow R_2 + \text{R0}$
22. AND R_1 R_2 000 111	h. $R_1 \leftarrow R_2$ AND R7
23. NOT R_1 R_2 11111	c. $R_1 \leftarrow -R_2 - 1$

24. If R1 ≥ 0 then go to x3009 else go to x3004.

Addr	*Value*	*Asm*	*Action/Comment*
x3000	0101 011 001 1 11111	AND R3,R2,-1	R3 ← R1 & -1 = R1
x3001	0000 011 000000111	BRNP 7	if R1 ≠ 0 then go to x3009
x3002	0000 100 000000010	BRN 2	else (R1 < 0) go to x3004
x3003	...	...	(We don't get here)

25. (Implement if R0 < 0 then R0 ← 0 else M[x30AC] ← R0)

Addr	*Value*	*Asm*	*Action/Comment*
x4000	0001 000 000 1 00000	ADD R0,R0,0	Test value of R0
x4001	0000 011 000000010	BRZP 2	If R0 ≥ 0, go to else br.
x4002	0101 000 000 1 00000	AND R0,R0,0	. R0 ← 0
x4003	0000 111 000000001	BR 1	Skip over else branch
x4004	0011 000 0 1010 0111	ST R0,xA7	. M[x40AC] ← R0
x4005	...		(code after if-else)

26. Implement "if R7 = 1 then go to x5000". (R1 is a temporary register.)

Addr	*Value*	*Asm*	*Action/Comment*
x8000	0001 001 111 1 11111	ADD R1,R7,-1	R1 ← R7 – 1
x8001	0000 101 000000011	BRNP 3	If R7 ≠ 1, go to end if
x8002	0010 001 000000001	LD R1,1	. R1 ← Target location
x8003	1100 000 001 00000	JMP R1	. Jump to target
x8004	x5000		Location of target
x8005	...		(code after if-then)

27. Implement "if R0 ≤ 0 then go to the location pointed to by x3150; else if R0 = 1 go to x3160; else go to x3165". (R1 is a temporary register.)

Addr	*Value*	*Asm*	*Action/Comment*
x3100	0001 000 000 1 00000	ADD R0,R0,0	Test R0
x3101	0000 001 000000010	BRP 2	if R0 ≤ 0 then
x3102	0010 000 001001101	LD R0,x4D	. Get loc pt'd to by x3150
x3103	1100 000 000 00000	JMP R0	. Jump to that location
x3104	0001 001 000 1 11111	ADD R1,R0,-1	else R1 ← R0 - 1
x3105	0000 010 001011010	BRZ x5A	. if R0 = 1 go to x3160
x3016	0000 111 001011110	BR x5E	. else go to x3165
...			(x49 words)
x3150	(address to jump to)		

