The LC-3 Computer, Part 1

CS 350: Computer Organization & Assembly Language Programming

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.
- The data movement and calculation instructions are two of the basic kinds of instructions.

B. Outcomes

- At the end of today, you should:
- Know the basic architecture of the LC-3: word size, number of registers, data types supported.
- Know how the LC-3's PC-offset, Base-Offset, Indirect, and Immediate addressing modes work.
- Know how the LC-3's data movement and calculation instructions work.

C. The LC-3 Computer

- Text 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). [Note: 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**, where the address is written as part of the instruction.
  - 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 ($-256 \le$ PC offset ≤ 255) to specify one memory operand.
- The effective address of the operand = PC + sign-extended 9-bit offset

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

Load instruction (LD)

- Mnemonic code `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*]

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

- **Example**: 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*

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

- **Example**: 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*])

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

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

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.)
  - To this address we add 6 bits of sign-extended offset.
  - Effective address = Base register + offset

LDR (Load using base Register)

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

- **Example**: Instruction `0110 011 111 111000`; `R7` = `x320C`; `M`[`x3204`] = `38`
  - `R3` ← `M`[`R7 – 8`] = `M`[`x320C – 8`] = `M`[`x3204`] = `38`

STR (Store using base Register)

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

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STR	0	1	1	1	Src			Base			Offset6					

- **Example**: Instruction `0111 011 111 111000; R7 = x320C; R3 = 18`.
  - `M[R7 – 8] = M[x320C – 8] = M[x3204] ← R3 = 18`

F. Indirect Addressing Mode

- The third addressing mode also uses a pointer, so we can access any location using a pointer.
- 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*]]

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

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

Store Indirect (STI)

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

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

- **Example**: Instruction at `x2FFF = 0011 011 111111000; M[x2FF8] = x4A30`
    - `M[M[x3000 – 8]] = M[M[x2FF8]] = M[x4A30] ← R3 = 24`

G. A Larger Example

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

H. Calculation/Data Operation Instructions

- On the LC-3, these instructions do not reference memory.
- For **NOT**, source and destination operands are registers.
- For **ADD** and **AND**, left-hand-source and destination operands are always registers.
    - Right-hand operand is either a register or a value that's hard-wired into the instruction ("immediate" mode).
    - Only 5 bits of immediate value (sign-extended): –16 to +15
- So **ADD** immediate is a great way to increment or decrement a register by 1, but to increment a register by ≥ 16 or < –16, you need multiple/a different **ADD** instruction.

	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

	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		

	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			0	0	0	Src2		

	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				

NOT, AND, ADD Examples

- `R1` ← `R0 AND x0000 = 0` (typical way to set a register to zero).
  - `0101 001 000 1 00000      ; AND R1,R0,0`
- `R0` ← `R2+R3`
  - `0001 000 010 000 011      ; ADD R0,R2,R3`
- `R7` ← `R7+R7` (the registers don't have to be distinct).
  - `0001 111 111 000 111      ; ADD R7,R7,R7`
- `R0` ← `R2+3`
  - `0001 000 010 1 00011      ; ADD R0,R2,3`
- `R1` ← `R1+15 = 15`
  - `0001 001 001 1 01111      ; ADD R1,R1,15`
- `R2` ← `R1 AND R1 = 15 AND 15 = 15` (a way to copy a register to another register)
  - `0101 010 001 000 001      ; AND R2,R1,R1`

- `R3` ← `NOT R2 = NOT 30 = -31`
    - `1001 011 010 111111        ; NOT R3,R2`
    - Note `NOT`(*nbr*) = `-`(*nbr*) `- 1` in 2's complement.

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)`
    - **Setting a register to zero**
        - Use *register* ← *register* `AND` 0's
    - **Copying one register to another:**
        - Don't try `LDR`: It means "Load using Base Register," not "Load from a register."
        - There are three ways to copy a register to another
            - *Destination register* ← *Source register* + 0
            - *Destination register* ← *Source register* `AND` 16 `1`'s
                - The immediate field needs to be `11111` (i.e., `-1`).
            - *Destination register* ← *Source register* `AND` *Source register*

I. Another Larger Example

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

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.

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.

C. Questions

1. What range of decimal values can a (9-bit signed) PC offset have?

For Questions 2–5, fill out the missing table entries below. Assume execution starts at `x3000` and that all registers contain unknown values. The table contains three kinds of instructions:

`LEA`: *Destination register* ← `PC+`*offset*

`LD`: *Destination register* ← `M`[`PC` + *offset*]

`ST`: `M`[`PC+`*offset*] ← *Source register*

Addr	*OpC*	*Instruction*	*Comments*
`x3000`	`LEA`	`1110 010 001001111`	`R2` ← `PC + x4F = x?....?`
`x3001`	`LD`	`0010 101 111111111`	`R5` ← `M[PC – 1] = x2???` [3 digits missing]
`x3002`	`ST`	`0011 100 000001100`	`?...?`
`x3003`	`LEA`	`1110 011 ?????????`	`R3` ← `PC – 8 = x3004 – 8 = x2FFC`

6. Say at memory address P, we want to do `R1 ← PC + x1000`. Can we do this with a LD or LEA instruction?

For Questions 7 – 12, fill out the missing table entries below. (Question 7 (a) – (c) is at address x3000, etc.) Assume execution starts at x3000 and that all registers contain unknown values.

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

For Questions 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. $R_1 \leftarrow R_2 + \mathtt{R0}$

h. $R_1 \leftarrow R_2$ AND R7

i. $R_1 \leftarrow R_2 - 1$

j. $R_1 \leftarrow R_2 + 1$

Solution

1. +255 through -256
2. Since the current instruction is at `x3000`, the `PC` = `x3001`, so `R2` ← `PC` + `x4F` = `x3002`+`x4F` = `x3050`.
3. Since the current instruction is at `x3001`, the `PC` = `x3002`, so `R5` ← `M[PC-1]` = `M [x3002-1]` = `M[x3001]` = `0010 101 111111111` = `x2BFF`
4. `0011 100 000001100` translates to `ST R4` with offset `12`, so `M[PC+12]` = `M[x3003+xC]` = `M[x300F]` ← `R4`
5. We want an offset of `-8`, which is `-(000001000)` = `11111000`.
6. We can do `R1` ← `PC`+n where -256 ≤ n ≤ 255, but there isn't any obvious way to do n = `x1000`. Once we get an `ADD` instruction, we could try `R1` ← `PC` (via `LEA R1 000000000`) and then add `x1000` to that.

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

Instruction		***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.	$R_1 \leftarrow R_2 + \texttt{R0}$
22.	`AND` R_1 R_2 `000 111`	h.	$R_1 \leftarrow R_2$ `AND R7`
23.	`NOT` R_1 R_2 `11111`	c.	$R_1 \leftarrow -R_2 - 1$