Chapter 5
The LC-3
I speak
Spanish to God,
Italian to women,
French to men, and
German to my horse.

Charles V

Source: https://upload.wikimedia.org/wikipedia/commons/thumb/3/3e/Elderly_Karl_V.jpg/220px-Elderly_Karl_V.jpg
Instruction Set Architecture

**ISA = All of the programmer-visible state and operations of the computer**

- **Programmer-visible state**
  - memory organization
    - address space -- how many locations can be addressed?
    - addressability -- 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**

1. Someone that wants to write a program in machine language (or translate from a high-level language to machine language) (i.e., for engineers in software)

2. Someone that wants to build a machine for the ISA (i.e., for engineers in hardware)
LC-3 Overview: Programmer-visible State

- Memory Organization
  - address space: $2^{16}$ locations (16-bit addresses)
  - addressability: 16 bits

- Register Set
  - 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: Operations

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)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOT: \[1\ 0\ 0\ 1\ \text{Dst} \ \\
\ 1\ 1\ 1\ 1\ 1\ 1\ \text{Src} \]

Note: \(\text{Src}\) and \(\text{Dst}\) could be the same register.
ADD/AND (Register)

ADD

\[ \begin{array}{cccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 1 & \text{Dst} & \text{Src1} & 0 & 0 & 0 & \text{Src2}
\end{array} \]

AND

\[ \begin{array}{cccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 1 & 0 & 1 & \text{Dst} & \text{Src1} & 0 & 0 & 0 & \text{Src2}
\end{array} \]

this zero means “register mode”
ADD/AND (Immediate)

Note: Immediate field is sign-extended.
Using Operate Instructions
With only ADD, AND, NOT…

• How do we subtract?

• How do we OR?

• 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: \(-256 \leq \text{offset} \leq +255\)

Can form any address \(X\), such that: \(\text{PC} - 256 \leq X \leq \text{PC} + 255\)

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

```
LD  0 0 1 0 | Dst | PCoffset9
```

Diagram showing the execution of the LD (PC-Relative) instruction.
ST (PC-Relative)

ST 0 0 1 1 | Src | PC[9:offset9]
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)

LDI 1 0 1 0 Dst PC offset 9
STI (Indirect)

STI 1 0 1 1 | Src | PC offset 9

Diagram of the STI instruction flow, showing the interaction between the PC, register file, memory, and instruction register.
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 bits for src (store)/dest (load) 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)

LDR  0  1  1  0  Dst  Base  offset6

Diagram showing the operation of the LDR (Base+Offset) instruction with labels and connections.
STR (Base + Offset)

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Src</td>
<td>Base</td>
<td>offset6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Diagram:
- Instruction Reg
- Sext
- IR[5:0]
- Register File
  - Src
  - Base
- Memory
- MAR
- MDR

Steps:
1. Sext
2. Base + Offset
3. SRC
4. Store
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 (Immediate)

LEA 1 1 1 0 Dst PCoffset9

Diagram:
- PC
- Instruction Reg
- Sext
- IR[8:0]
- +
- Register File
- Dst
**Example**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x30F6</td>
<td>1 1 1 0 0 0 1 1 1 1 1 1 1 1 0 1</td>
<td>R1 ← PC − 3 = x30F4</td>
</tr>
<tr>
<td>x30F7</td>
<td>0 0 0 1 0 1 0 0 0 1 1 1 0 1 1 1 0</td>
<td>R2 ← R1 + 14 = x3102</td>
</tr>
<tr>
<td>x30F8</td>
<td>0 0 1 1 0 1 0 1 1 1 1 1 1 0 1 1</td>
<td>M[PC − 5] ← R2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M[x30F4] ← x3102</td>
</tr>
<tr>
<td>x30F9</td>
<td>0 1 0 1 0 1 0 0 1 1 0 0 0 0 0 0</td>
<td>R2 ← 0</td>
</tr>
<tr>
<td>x30FA</td>
<td>0 0 0 1 0 1 0 0 1 0 1 1 0 1 0 1</td>
<td>R2 ← R2 + 5 = 5</td>
</tr>
<tr>
<td>x30FB</td>
<td>0 1 1 1 0 1 0 0 1 1 0 0 1 1 1 0 0</td>
<td>M[R1+14] ← R2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M[x3102] ← 5</td>
</tr>
<tr>
<td>x30FC</td>
<td>1 0 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1</td>
<td>R3 ← M[M[x30F4]]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R3 ← M[x3102]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R3 ← 5</td>
</tr>
</tbody>
</table>

opcode
Control Instructions

Used to alter the sequence of instructions (by changing the Program Counter)

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.
What happens if bits [11:9] are all zero? All one?
Using Branch Instructions

Compute sum of 12 integers.
Numbers start at location x3100. Program starts at location x3000.

R1 $\leftarrow$ x3100
R3 $\leftarrow$ 0
R2 $\leftarrow$ 12

R2=0?

R4 $\leftarrow$ M[R1]
R3 $\leftarrow$ R3+R4
R1 $\leftarrow$ R1+1
R2 $\leftarrow$ R2-1

NO

YES
## Sample Program

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>1 1 1 0</td>
<td>( R1 \leftarrow x3100 ) (PC+0xFF)</td>
</tr>
<tr>
<td>x3001</td>
<td>0 1 0 1</td>
<td>( R3 \leftarrow 0 )</td>
</tr>
<tr>
<td>x3002</td>
<td>0 1 0 1</td>
<td>( R2 \leftarrow 0 )</td>
</tr>
<tr>
<td>x3003</td>
<td>0 0 0 1</td>
<td>( R2 \leftarrow 12 )</td>
</tr>
<tr>
<td>x3004</td>
<td>0 0 0 0</td>
<td>If Z, goto x300A (PC+5)</td>
</tr>
<tr>
<td>x3005</td>
<td>0 1 1 0</td>
<td>Load next value to R4</td>
</tr>
<tr>
<td>x3006</td>
<td>0 0 0 1</td>
<td>Add to R3</td>
</tr>
<tr>
<td>x3007</td>
<td>0 0 0 1</td>
<td>Increment R1 (pointer)</td>
</tr>
<tr>
<td>x3008</td>
<td>0 0 0 1</td>
<td>Decrement R2 (counter)</td>
</tr>
<tr>
<td>x3009</td>
<td>0 0 0 0</td>
<td>Goto x3004 (PC-6)</td>
</tr>
</tbody>
</table>

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
JMP (Register)

Jump is an unconditional branch -- *always* taken.

- Target address is the contents of a register.
- Allows any target address.

```
JMP
11000000 | Base | 00000000
```

PC

Register File

Base
Calls a **service routine**, identified by 8-bit “trap vector.”

<table>
<thead>
<tr>
<th><strong>vector</strong></th>
<th><strong>routine</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>x23</td>
<td>input a character from the keyboard</td>
</tr>
<tr>
<td>x21</td>
<td>output a character to the monitor</td>
</tr>
<tr>
<td>x25</td>
<td>halt the program</td>
</tr>
</tbody>
</table>

When routine is done, PC is set to the instruction following TRAP. (We’ll talk about how this works later.)
Another Example

Count the occurrences of a character in a file

• Program begins at location x3000
• Read character from keyboard
• Load each character from a “file”
  ➢ File is a sequence of memory locations
  ➢ Starting address of file is stored in the memory location immediately after the program
• If file character equals input character, increment counter
• End of file is indicated by a special ASCII value: EOT (x04)
• At the end, print the number of characters and halt
  (assume there will be less than 10 occurrences of the character)

A special character used to indicate the end of a sequence is often called a sentinel.
  • Useful when you don’t know ahead of time how many times to execute a loop.
Flow Chart

1. **Count = 0**
   - (R2 = 0)

2. **Ptr = 1st file character**
   - (R3 = M[x3012])

3. **Input char from keybd**
   - (TRAP x23)

4. **Load char from file**
   - (R1 = M[R3])

5. **Done?**
   - (R1 ?= EOT)
   - **YES**
     - **Convert count to ASCII character**
       - (R0 = x30, R0 = R2 + R0)
   - **NO**

6. **Match?**
   - (R1 ?= R0)
   - **YES**
     - **Incr Count**
       - (R2 = R2 + 1)
   - **NO**

7. **Load next char from file**
   - (R3 = R3 + 1, R1 = M[R3])

8. **Print count**
   - (TRAP x21)

9. **HALT**
   - (TRAP x25)
## Program (1 of 2)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>0 1 0 1</td>
<td>0 1 0 0 1 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>x3001</td>
<td>0 0 1 0</td>
<td>0 1 1 0 0 0 0 0 1 0 0 0</td>
</tr>
<tr>
<td>x3002</td>
<td>1 1 1 1</td>
<td>0 0 0 0 0 0 0 1 0 0 0 1 1</td>
</tr>
<tr>
<td>x3003</td>
<td>0 1 1 0</td>
<td>0 0 1 0 1 1 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>x3004</td>
<td>0 0 0 1</td>
<td>1 0 0 0 0 1 1 1 1 1 1 0 0</td>
</tr>
<tr>
<td>x3005</td>
<td>0 0 0 0</td>
<td>0 1 0 0 0 0 0 0 1 0 0 0</td>
</tr>
<tr>
<td>x3006</td>
<td>1 0 0 1</td>
<td>0 0 1 0 0 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>x3007</td>
<td>0 0 0 1</td>
<td>0 0 1 0 0 1 1 0 0 0 0 1</td>
</tr>
<tr>
<td>x3008</td>
<td>0 0 0 1</td>
<td>0 0 1 0 0 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>x3009</td>
<td>0 0 0 0</td>
<td>1 0 1 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
## Program (2 of 2)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x300A</td>
<td>0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1</td>
<td>$R2 \leftarrow R2 + 1$</td>
</tr>
<tr>
<td>x300B</td>
<td>0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1</td>
<td>$R3 \leftarrow R3 + 1$</td>
</tr>
<tr>
<td>x300C</td>
<td>0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0</td>
<td>$R1 \leftarrow M[R3]$</td>
</tr>
<tr>
<td>x300D</td>
<td>0 0 0 0 1 1 1 1 1 1 1 1 0 1 1 0</td>
<td>Goto x3004</td>
</tr>
<tr>
<td>x300E</td>
<td>0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0</td>
<td>$R0 \leftarrow M[x3013]$</td>
</tr>
<tr>
<td>x300F</td>
<td>0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0</td>
<td>$R0 \leftarrow R0 + R2$</td>
</tr>
<tr>
<td>x3010</td>
<td>1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1</td>
<td>Print R0 (TRAP x21)</td>
</tr>
<tr>
<td>x3011</td>
<td>1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1</td>
<td>HALT (TRAP x25)</td>
</tr>
<tr>
<td>X3012</td>
<td>Starting Address of File</td>
<td></td>
</tr>
<tr>
<td>x3013</td>
<td>0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0</td>
<td>ASCII x30 (‘0’)</td>
</tr>
</tbody>
</table>
LC-3
Data Path Revisited

Filled arrow
= info to be processed.
Unfilled arrow
= control signal.
LC-3
Data Path
Revisited

Filled arrow
= info to be processed.
Unfilled arrow
= control signal.
LC-3 Data Path Revisited

Filled arrow = info to be processed.
Unfilled arrow = control signal.
LC-3
Data Path Revisited

Filled arrow = info to be processed. 
Unfilled arrow = control signal.
Data Path Components

Global bus

- special set of wires that carry a 16-bit signal to many components
- inputs to the bus are “tri-state devices,” that only place a signal on the bus when they are enabled
- only one (16-bit) signal should be enabled at any time
  - control unit decides which signal “drives” the bus
- any number of components can read the bus
  - register only captures bus data if it is write-enabled by the control unit

Memory

- Control and data registers for memory and I/O devices
- memory: MAR, MDR (also control signal for read/write)
Data Path Components

ALU

- Accepts inputs from register file and from sign-extended bits from IR (immediate field).
- Output goes to bus.
  - used by condition code logic, register file, memory

Register File

- Two read addresses (SR1, SR2), one write address (DR)
- Input from bus
  - result of ALU operation or memory read
- Two 16-bit outputs
  - used by ALU, PC, memory address
  - data for store instructions passes through ALU
Data Path Components

**PC and PCMUX**

- Three inputs to PC, controlled by PCMUX
  1. PC+1 – FETCH stage
  2. Address adder – BR, JMP
  3. bus – TRAP (discussed later)

**MAR and MARMUX**

- Two inputs to MAR, controlled by MARMUX
  1. Address adder – LD/ST, LDR/STR
  2. Zero-extended IR[7:0] -- TRAP (discussed later)
Data Path Components

Condition Code Logic

- Looks at value on bus and generates N, Z, P signals
- Registers set only when control unit enables them (LD.CC)
  - only certain instructions set the codes
    (ADD, AND, NOT, LD, LDI, LDR, LEA)

Control Unit – Finite State Machine

- On each machine cycle, changes control signals for next phase of instruction processing
  - who drives the bus? (GatePC, GateALU, …)
  - which registers are write enabled? (LD.IR, LD.REG, …)
  - which operation should ALU perform? (ALUK)
  - which multiplexor inputs go to output? (SR2MUX, …)
- Logic includes decoder for opcode, etc.
• Instruction Set Architecture (ISA)
  • A software/hardware interface
  • specifies all of the *programmer-visible state and operations of the computer*
    ✓ state
      – memory
      – register set
    ✓ operation
      – opcodes
      – data types
      – addressing modes
• Three main instruction categories
  ✓ Operate instructions: add, and, not, etc
  ✓ Data movement instructions
    – load
    – store
  ✓ Control instructions: branch, jump, trap, etc