Chapter 5
The LC-3
Instruction Set Architecture

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

- memory organization
  - address space -- how many 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:
  \[N = \text{negative}, \ Z = \text{zero}, \ P = \text{positive} \ (> 0)\]

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
Dataflow diagrams

Will show dataflow diagram with each instruction.

• illustrates *when* and *where* data moves to accomplish the desired operation

Components in data flow diagrams:

• Registers: each register can hold 16 bits in LC-3. Several
• Register File: contains eight 16-bit registers
• ALU: combinational (no storage). Two inputs, one output, functionality can be selected.
• SEXT: combinational. Sign extension from 5 to 16 bits.
• Memory: 216 words. Addressed by 16 bit MAR and MDR holds data

How components are implemented?

• Last third of the class
NOT (Register)

NOT

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

Assembly Ex:

NOT R3, R2

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

ADD

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

AND

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

this zero means “register mode”

Assembly Ex:
Add R3, R1, R3
ADD/AND (Immediate)

ADD
\[
\begin{array}{cccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 1 & | & Dst & | & Src1 & 1 & | & Imm5
\end{array}
\]

AND
\[
\begin{array}{cccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 1 & 0 & 1 & | & Dst & | & Src1 & 1 & | & Imm5
\end{array}
\]

Note: Immediate field is sign-extended.

*Assembly Ex: Add R3, R3, #1*
Using Operate Instructions
With only ADD, AND, NOT…

- How do we subtract?  Hint: Negate and add

- How do we OR?  Hint: Demorgan’s law

- 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)

Assembly Ex: LD R1, Label1
ST (PC-Relative)

<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>ST</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Src</td>
<td>PCoffset9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assembly Ex:
ST R1, Label2
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)

Assembly Ex:
LDI    R4, Adr

LDI 1010 | Dst | PCoffset9

Diagram of the LDI (Indirect) operation flow:
STI (Indirect)

Assembly Ex:
STI R4, Adr

STI 1 0 1 1 Src PCoffset9
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)

LDR 0 1 1 0  Dst  Base  offset6

Assembly Ex: LDR R4, R1, #1
STR (Base+Offset)

Assembly Ex:
STR R4, R1, #1
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 R1, Begin
LDR R3, R1, #0

We can use the destination register as a pointer.
LEA (Immediate)

Assembly Ex:
LEA R1, Lab1
### Example (with RTL)

<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 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 0 1 1 1 0</td>
<td>R2 ← R1 + 14 = x3102</td>
</tr>
</tbody>
</table>
| x30F8  | 0 0 1 1 0 1 0 1 1 1 1 1 1 1 0 1 1 | M[PC - 5] ← R2  
M[x30F4] ← x3102 |
| x30F9  | 0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 0 | R2 ← 0 |
| x30FA  | 0 0 0 1 0 1 0 1 0 0 1 0 1 0 0 1 0 1 | R2 ← R2 + 5 = 5 |
| x30FB  | 0 1 1 1 0 1 0 0 0 1 0 0 1 1 1 0 | M[R1+14] ← R2  
M[x3102] ← 5 |
| x30FC  | 1 0 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 | R3 ← M[M[x30F4]]  
R3 ← M[x3102]  
R3 ← 5 |
## Example (in assembly)

<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 0 1</td>
<td>LEA R1, Lab2</td>
</tr>
<tr>
<td>x30F7</td>
<td>0 0 0 1 0 1 0 0 0 1 1 0 1 1 1 0</td>
<td>ADD R2, R1, #14</td>
</tr>
<tr>
<td>x30F8</td>
<td>0 0 1 1 0 1 0 1 1 1 1 1 0 1 1 1</td>
<td>ST R2, Lab2</td>
</tr>
<tr>
<td>x30F9</td>
<td>0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0</td>
<td>AND R2, R2, #0</td>
</tr>
<tr>
<td>x30FA</td>
<td>0 0 0 1 0 1 0 0 1 0 1 0 0 1 0 1</td>
<td>ADD R2, R2, #5</td>
</tr>
<tr>
<td>x30FB</td>
<td>0 1 1 1 0 1 0 0 0 1 0 0 1 1 1 0</td>
<td>LDR R2, R1, #14</td>
</tr>
<tr>
<td>x30FC</td>
<td>1 0 1 0 0 1 1 1 1 1 1 1 0 1 1 1</td>
<td>LDI R2, Lab2</td>
</tr>
</tbody>
</table>
## LC3 Addressing Modes: Comparison

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOT</strong></td>
<td>NOT R2, R1</td>
<td>R2</td>
<td>R1</td>
</tr>
<tr>
<td><strong>ADD / AND</strong> (imm)</td>
<td>ADD R3, R2, #7</td>
<td>R3</td>
<td>R2, #7</td>
</tr>
<tr>
<td><strong>ADD /AND</strong></td>
<td>ADD R3, R2, R1</td>
<td>R3</td>
<td>R2, R1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LD</strong></td>
<td>LD R4, LABEL</td>
<td>R4</td>
<td>M[LABEL]</td>
</tr>
<tr>
<td><strong>ST</strong></td>
<td>ST R4, LABEL</td>
<td>M[LABEL]</td>
<td>R4</td>
</tr>
<tr>
<td><strong>LDI</strong></td>
<td>LDI R4, HERE</td>
<td>R4</td>
<td>M[M[HERE]]</td>
</tr>
<tr>
<td><strong>STI</strong></td>
<td>STI R4, HERE</td>
<td>M[M[HERE]]</td>
<td>R4</td>
</tr>
<tr>
<td><strong>LDR</strong></td>
<td>LDR R4, R2, #−5</td>
<td>R4</td>
<td>M[R2 − 5]</td>
</tr>
<tr>
<td><strong>STR</strong></td>
<td>STR R4, R2, #5</td>
<td>M[R2 + 5]</td>
<td>R4</td>
</tr>
<tr>
<td><strong>LEA</strong></td>
<td>LEA R4, TARGET</td>
<td>R4</td>
<td>address of TARGET</td>
</tr>
</tbody>
</table>
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: 
  \textcolor{red}{\textit{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 ← x3100
R3 ← 0
R2 ← 12

R2=0?

R4 ← M[R1]
R3 ← R3+R4
R1 ← R1+1
R2 ← R2-1

YES

NO
Sum of 4 integers

;Computes sum of integers
;R1: pointer, initialized to NUMS (x300C)
;R3: sum, initially cleared, accumulated here
;R2: down counter, initially holds number of numbers 4

.ORIG 0x3000

DONE ST R3, SUM ;added
HALT

NUMS .FILL 3
  .FILL -4
  .FILL 7
  .FILL 3

SUM .BLKW 1
.END

LEA R1,NUMS
AND R3,R3, #0
AND R2,R2, #0
ADD R2, R2, #4

LOOP BRnzp LOOP
LDR R4,R1,#0
ADD R3,R3,R4
ADD R1,R1,#1
ADD R2,R2,#-1
BRnzp LOOP
## Sample Program

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>11 10 00 10 11 11 11 11 11</td>
<td>R1 ← x3100 (PC+0xFF)</td>
</tr>
<tr>
<td>x3001</td>
<td>01 01 01 11 01 11 10 00 00 0</td>
<td>R3 ← 0</td>
</tr>
<tr>
<td>x3002</td>
<td>01 01 01 00 10 10 00 00 00</td>
<td>R2 ← 0</td>
</tr>
<tr>
<td>x3003</td>
<td>00 01 01 00 10 10 11 00</td>
<td>R2 ← 12</td>
</tr>
<tr>
<td>x3004</td>
<td>00 00 00 11 00 00 00 00 00 10 01</td>
<td>If Z, goto x300A (PC+5)</td>
</tr>
<tr>
<td>x3005</td>
<td>01 11 01 00 00 10 00 00 00</td>
<td>Load next value to R4</td>
</tr>
<tr>
<td>x3006</td>
<td>00 01 01 11 01 11 00 00 00 01</td>
<td>Add to R3</td>
</tr>
<tr>
<td>x3007</td>
<td>00 01 00 10 01 11 00 00 00 1</td>
<td>Increment R1 (pointer)</td>
</tr>
<tr>
<td>x3008</td>
<td>00 01 01 00 10 11 11 11 11</td>
<td>Decrement R2 (counter)</td>
</tr>
<tr>
<td>x3009</td>
<td>00 00 11 11 11 11 11 10 01 00</td>
<td>Goto x3004 (PC-6)</td>
</tr>
</tbody>
</table>
JMP (Register)

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

- Target address is the contents of a register.
- Allows any target address.
Calls a **service routine**, identified by 8-bit “trap vector.”

<table>
<thead>
<tr>
<th>vector</th>
<th>routine</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

Count = 0
(R2 = 0)

Ptr = 1st file character
(R3 = M[x3012])

Input char from keybd
(TRAP x23)

Load char from file
(R1 = M[R3])

Done?
(R1 ?= EOT)

Yes

Convert count to ASCII character
(R0 = x30, R0 = R2 + R0)

Print count
(TRAP x21)

Halt
(TRAP x25)

NO

Match?
(R1 ?= R0)

Yes

Incr Count
(R2 = R2 + 1)

Load next char from file
(R3 = R3 + 1, R1 = M[R3])
.ORIG x3000
AND  R2,R2,#0  ; R2 is counter, initialize to 0
LD   R3,PTR    ; R3 is pointer to characters
TRAP x23       ; R0 gets character input
LDR  R1,R3,#0  ; R1 gets the next character
;
; Test character for end of file
;
TEST  ADD  R4,R1,#-4  ; Test for EOT
    BRz  OUTPUT    ; If done, prepare the output
;
; Test character for match. If a match, increment count.
;
    NOT  R1,R1
    ADD  R1,R1,R0  ; If match, R1 = xFFFF
    NOT  R1,R1     ; If match, R1 = x0000
    BRnp GETCHAR   ; no match, do not increment
    ADD  R2,R2,#1
;
; Get next character from the file
;
GETCHAR ADD  R3,R3,#1  ; Increment the pointer
    LDR  R1,R3,#0  ; R1 gets the next character to test
    BRnzp TEST
;
; Output the count.
;
OUTPUT LD  R0,ASCII  ; Load the ASCII template
    ADD  R0,R0,R2  ; Convert binary to ASCII
    TRAP x21      ; ASCII code in R0 is displayed
    TRAP x25      ; Halt machine
;
; Storage for pointer and ASCII template
;
ASCIIFILL x0030
PTR .FILL x3015
.END
# Program (1 of 2)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>0101010010100000</td>
<td>R2 ← 0 (counter)</td>
</tr>
<tr>
<td>x3001</td>
<td>0010011000010000</td>
<td>R3 ← M[x3102] (ptr)</td>
</tr>
<tr>
<td>x3002</td>
<td>1111000000100011</td>
<td>Input to R0 (TRAP x23)</td>
</tr>
<tr>
<td>x3003</td>
<td>0110001011000000</td>
<td>R1 ← M[R3]</td>
</tr>
<tr>
<td>x3004</td>
<td>0001100000111110</td>
<td>R4 ← R1 – 4 (EOT)</td>
</tr>
<tr>
<td>x3005</td>
<td>0000010000001000</td>
<td>If Z, goto x300E</td>
</tr>
<tr>
<td>x3006</td>
<td>1001001001111111</td>
<td>R1 ← NOT R1</td>
</tr>
<tr>
<td>x3007</td>
<td>0001001001100001</td>
<td>R1 ← R1 + 1</td>
</tr>
<tr>
<td>X3008</td>
<td>0001001001000000</td>
<td>R1 ← R1 + R0</td>
</tr>
<tr>
<td>x3009</td>
<td>0000101000000001</td>
<td>If N or P, goto x300B</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>0001010010100001</td>
<td>R2 ← R2 + 1</td>
</tr>
<tr>
<td>x300B</td>
<td>0001011011100001</td>
<td>R3 ← R3 + 1</td>
</tr>
<tr>
<td>x300C</td>
<td>0110001011000000</td>
<td>R1 ← M[R3]</td>
</tr>
<tr>
<td>x300D</td>
<td>0000111111110110</td>
<td>Goto x3004</td>
</tr>
<tr>
<td>x300E</td>
<td>001000000000000100</td>
<td>R0 ← M[x3013]</td>
</tr>
<tr>
<td>x300F</td>
<td>000100000000000010</td>
<td>R0 ← R0 + R2</td>
</tr>
<tr>
<td>x3010</td>
<td>1111000000100001</td>
<td>Print R0 (TRAP x21)</td>
</tr>
<tr>
<td>x3011</td>
<td>1111000000100101</td>
<td>HALT (TRAP x25)</td>
</tr>
<tr>
<td>X3012</td>
<td><strong>Starting Address of File</strong></td>
<td></td>
</tr>
<tr>
<td>x3013</td>
<td>0000000001100000</td>
<td>ASCII x30 (‘0’)</td>
</tr>
</tbody>
</table>
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
More details later.
Multiplexer (MUX): selects data from multiple sources

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)
  ➢ ...
• Logic includes decoder for opcode, etc.