LC-3
Instruction Set Architecture
(Ch5)
D = 0
WE = 1
clk edge

11101
16 8 4 1
29
-16
13
Instruction Set Architecture

ISA is all of the *programmer-visible* components and operations of the computer.

- memory organization
  - address space -- how may 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

The ISA provides all the information needed for someone to write a program in machine language (or translate from a high-level language to machine language).
Memory vs. 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 is 16 bits wide
  - how many bits to uniquely identify a register?
- other registers
  - not directly addressable, but used/effect by instructions
  - PC (program counter), condition codes
Instruction Set

Opcodes
- 15 opcodes
- **Operate** (Logical or Arithmetic) 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 (< 0), 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

NOT 1001

NOT R1, R1

Note: Src and Dst could be the same register.

Note: works only with registers.
ALU

Diagram:

- Num1
- Num2
- Num
- SUM
- AND
- S
Instructions

ADD/AND

ADD
0 0 0 1 Dst Src1 0 0 0 Src2

AND
0 1 0 1 Dst Src1 0 0 0 Src2

ADD Rf, R1, R2

This zero means "register mode"
Instructions

ADD

AND

Note: Immediate field is sign-extended.

AND R1, R1, 3

AND R1, R1, 0

-2^4 -16 - 15
Using Operate Instructions

With only ADD, AND, NOT...

- How do we subtract? $R_3 = R_1 - R_2$
  $R_2 = \overline{R_1} \oplus (R_2)$
  $R_3 = R_1 + R_2$
- How do we OR? $R_3 = R_1 \text{ or } R_2$

- How do we copy from one register to another? $R_3 = R_2$

- How do we initialize a register to zero? $R_1 = R_1 \text{ and } 0$
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*
Addressing Modes

• How memory is addressed.
• Different instructions use different addressing modes.
• Some instructions support more than one addressing mode.
LC-3 Addressing Modes

- PC-Relative
  - Address is a displacement from PC
- Indirect
  - Use PC-Relative to get address from memory
- Base plus Offset
  - Use contents of a register as base address and add offset to find address (most common for load/store architectures)

4 + 3 = 7
334 - 72
PC-Relative

The Problem:
We 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 only 9 bits available for address.
PC-Relative Addressing Mode

The Solution:
Use the 9 bits as a **signed offset** from the current PC.

9 bits allows the offset range to be:

\[-256 \leq \text{offset} \leq +255\]

We can now form any address \( X \), such that:

\[(PC - 256) \leq X \leq (PC + 255)\]

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

PC-Relative Addressing Mode

LD 0 0 1 0 Dst PC offset 9

Diagram showing the flow of data from PC to memory through various registers and file.
PC-Relative Addressing Mode

ST (Store Data)

ST 0 0 1 1 Src PCoffset9

Diagram shows the flow of data and instructions in the ST (Store Data) mode, including
- PC
- Register File
- Memory
- Instruction Reg
- Sext
- IR[8:0]
- MAR
- MDR
Indirect

The Problem:
With PC-relative mode, we can only address data within 256 words of the instruction.

– What about the rest of memory? How do we access it?
Indirect Addressing Mode

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.
Indirect Addressing Mode

Indirect Load Instruction (LDI)

1010  Dst  PCoffset9

Flow of Control:
1. Instruction Reg: IR[8:0]
2. Sext: R[8:0] + offset
3. MAR
4. MDR
5. Memory
6. Register File
Indirect Addressing Mode

STI

1 0 1 1  Src  PCoffset9

Diagram showing the flow of instructions and data in indirect addressing mode.
Base + Offset

Remember The Problem:
With PC-relative mode, can only address data within 256 words of the instruction.

- What about the rest of memory? How do we access it?
Solution #2:
– Use a register to generate a full 16-bit address.

4 bits for opcode, 3 bits for src/dest register, 3 bits for base register – the remaining 6 bits are used as a signed offset.

– Offset is sign-extended before adding to base register.
Base + Offset Addressing Mode

LDR

0 1 1 0  Dst  Base  offset 6

PC-relative

Instruction Reg

Register File

Memory

IR[5:0]

Sext

+/

MAR

MDR

CMPE-012/L
Base + Offset Addressing Mode

STR

0 1 1 1  Src  Base  offset6

Instruction Reg

IR[5:0]

Sext

Register File

Src

Base

Memory

MDR

MAR

CMPE-012/L
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 1110 \( \text{PCoffset9} \)

Labels

foo

LEA, foo

LEA, R1, 44
### Example Code

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</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>LEA</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>ADD</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 | ST  | M[PC - 5] ← R2  
M[x30F4] ← x3102 |
| x30F9   | 0 1 0 1 0 1 0 0 1 0 1 0 1 0 0 0 0 0 | AND | R2 ← 0 |
| x30FA   | 0 0 0 1 0 1 0 0 1 0 1 0 1 0 0 1 0 1 | ADD | R2 ← R2 + 5 = 5 |
| x30FB   | 0 1 1 1 0 1 0 0 0 1 0 0 1 1 1 1 0 | STR | M[R1+14] ← R2  
M[x3102] ← 5 |
| x30FC   | 1 0 1 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1 | LDI | R3 ← M[M[x30F4]]  
R3 ← M[x3102]  
R3 ← 5 |
Instructions

Control Instructions

Used to alter the sequence of instructions. This is done by changing the PC.

\[ x = 0 \]

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" Keyboard / terminal
- routine will return control to the next instruction (after TRAP) when finished
Condition Codes

LC-3 has three condition code bits:

- 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 is used
  – target address is made by adding signed offset (IR[8:0]) to current PC.
If the branch is not taken, the next sequential instruction is executed.

**Labels**

- Note: PC has already been incremented by FETCH stage.
- Note: Target must be within 256 words of BR instruction.
BR (PC-Relative)

BRn
BRz

ADD R1, R1, 0
R1 = R1 + 0
Example: Using a Branch

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

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

Flowchart:

1. Check $R2 = 0$?
   - Yes: $R2 \leftarrow R2 - 1$
   - No: $R4 \leftarrow M[R1]$, $R3 \leftarrow R3 + R4$, $R1 \leftarrow R1 + 1$
## Example: Using a Branch

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>×3000</td>
<td>1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1</td>
<td>LEA</td>
<td>R1 ← ×3100 (PC+0xFF)</td>
</tr>
<tr>
<td>×3001</td>
<td>0 1 0 1 0 1 1 0 1 1 1 1 0 0 0 0</td>
<td>AND</td>
<td>R3 ← 0</td>
</tr>
<tr>
<td>×3002</td>
<td>0 1 0 1 0 1 0 0 1 0 1 0 1 0 0 0</td>
<td>AND</td>
<td>R2 ← 0</td>
</tr>
<tr>
<td>×3003</td>
<td>0 0 0 1 0 1 0 0 1 0 1 0 1 0 1 0</td>
<td>ADD</td>
<td>R2 ← 12</td>
</tr>
<tr>
<td>×3004</td>
<td>0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0</td>
<td>BRz</td>
<td>If Z, goto ×300A (PC+5)</td>
</tr>
<tr>
<td>×3005</td>
<td>0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0</td>
<td>LDR</td>
<td>Load next value to R4</td>
</tr>
<tr>
<td>×3006</td>
<td>0 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0</td>
<td>ADD</td>
<td>R3 ← R4 + R3</td>
</tr>
<tr>
<td>×3007</td>
<td>0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1</td>
<td>ADD</td>
<td>Increment R1 (pointer)</td>
</tr>
<tr>
<td>×3008</td>
<td>0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1</td>
<td>ADD</td>
<td>Decrement R2 (counter)</td>
</tr>
<tr>
<td>×3009</td>
<td>0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 0</td>
<td>BRnzp</td>
<td>Goto ×3004 (PC-6)</td>
</tr>
</tbody>
</table>

**opcode**
Instructions

**JMP**

Jump is an **unconditional branch** -- *always* taken.
- Target address is the contents of a register.
- Allows any target address.

```
JMP 1 1 0 0 0 0 0 0
    Base 0 0 0 0 0 0 0 0
```

Diagram:
- PC
- Register File: Base

Instructions

**TRAP**

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

```
1111100000 trapvect8
```

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**.
Another Example

Count the occurrences of a character in an array

- Program begins at location x3000
- Read character from keyboard
- Load each character from an array
  - An array is a sequence of memory locations
  - Starting address of array is stored in the memory location immediately after the program
- If array character equals input character, increment counter
- End of array is indicated by a special ASCII value: EOT (x04)
- At the end, print the number of characters and halt (let's assume there will be less than 10 occurrences of the character)
Flow Chart

Count = 0  
(R2 = 0)

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

Input char from keybd  
(TRAP x23)

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

Done?  
(R1 ≠ EOT)

YES

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

NO

Match?  
(R1 = R0)

YES

Incr Count  
(R2 = R2 + 1)

NO

Print count  
(TRAP x21)

HALT  
(TRAP x25)

Load next char from array  
(R3 = R3 + 1, R1 = M[R3])
n \geq \frac{32k}{R_2} \implies R_2 = R_1

R_2 - R_1 = 0
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000</td>
<td>0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0</td>
<td>AND</td>
<td>R2 ← 0 (counter)</td>
</tr>
<tr>
<td>0x3001</td>
<td>0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0</td>
<td>LD</td>
<td>R3 ← M[0x3012] (ptr)</td>
</tr>
<tr>
<td>0x3002</td>
<td>1 1 1 1 0 0 0 0 0 0 1 0 0 0 1 1</td>
<td>TRAP</td>
<td>Input to RO (TRAP x23)</td>
</tr>
<tr>
<td>0x3003</td>
<td>0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0</td>
<td>LDR</td>
<td>R1 ← M[R3]</td>
</tr>
<tr>
<td>0x3004</td>
<td>0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0</td>
<td>ADD</td>
<td>R4 ← R1 - 4 (EOT)</td>
</tr>
<tr>
<td>0x3005</td>
<td>0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0</td>
<td>BRz</td>
<td>If Z, goto 0x300E</td>
</tr>
<tr>
<td>0x3006</td>
<td>1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1</td>
<td>NOT</td>
<td>R1 ← NOT R1</td>
</tr>
<tr>
<td>0x3007</td>
<td>0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1</td>
<td>ADD</td>
<td>R1 ← R1 + 1</td>
</tr>
<tr>
<td>0x3008</td>
<td>0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 0</td>
<td>ADD</td>
<td>R1 ← R1 + RO</td>
</tr>
<tr>
<td>0x3009</td>
<td>0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1</td>
<td>BRnp</td>
<td>If N or P, goto 0x300B</td>
</tr>
</tbody>
</table>

*opcode*
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</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>ADD</td>
<td>R2 ← R2 + 1</td>
</tr>
<tr>
<td>x300B</td>
<td>0 0 0 1 0 1 1 0 1 1 1 1 0 0 0 0 1</td>
<td>ADD</td>
<td>R3 ← 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 0</td>
<td>LDR</td>
<td>R1 ← 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>BRnzp</td>
<td>Goto x3004</td>
</tr>
<tr>
<td>x300E</td>
<td>0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0</td>
<td>LD</td>
<td>RO ← M[x3013]</td>
</tr>
<tr>
<td>x300F</td>
<td>0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0</td>
<td>ADD</td>
<td>RO ← RO + R2</td>
</tr>
<tr>
<td>x3010</td>
<td>1 1 1 1 0 0 0 0 0 0 0 1 0 0 0 0 1</td>
<td>TRAP</td>
<td>Print RO (TRAP x21)</td>
</tr>
<tr>
<td>x3011</td>
<td>1 1 1 1 0 0 0 0 0 0 0 1 0 0 1 0 1</td>
<td>TRAP</td>
<td>HALT (TRAP x25)</td>
</tr>
<tr>
<td>x3012</td>
<td></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 0</td>
<td>Data</td>
<td>ASCII x30 (‘0’)</td>
</tr>
</tbody>
</table>

(opcode)
LC-3
Data Path

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
- There are 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
- There are 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.
Summary of ISA

• Instruction Set Architecture
• The ISA provides all the information needed for someone to write a program in machine language (or translate from a high-level language to machine language).