LC-3

Instruction Set Architecture

(Ch5)
Instruction Set Architecture

ISA is 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

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/effecteded 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.
Instructions

**NOT**

<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>1</td>
<td>0</td>
<td>0</td>
<td>1</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>

**Note:** Src and Dst could be the same register.

**Note:** works only with registers.

Register File

ALU

1

2
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

This zero means "register mode"
ADD/AND

**ADD**

<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>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>Dst</td>
<td>Src1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**AND**

<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>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>Dst</td>
<td>Src1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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*
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)
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:

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

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 PCoffset9

Diagram:
- PC
- Register File
- Memory
- Instruction Reg
- Sext
- IR[8:0]
- MAR
- MDR

Steps:
1. Instruction Reg
2. Sext
3. IR[8:0]
4. Dst
PC-Relative Addressing Mode

ST (Store Data)

ST 0 0 1 1 Src PCoffset9

Diagram of the instruction execution flow:
1. IR[8:0] is input to Sext.
2. Sext's output (PC offset) is added to the contents of the register file.
3. The result is written to memory.

PC, Instruction Reg, Register File, Memory, MAR, MDR are components of the system.
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?
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

**LDI**

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

**LDI** 1 0 1 0 **Dst** **PC**\ offset9

- **PC**
- **Register File**
- **Memory**
- **Instruction Reg**
- **Sext**
- **IR[8:0]**
- **MAR**
- **MDR**

1. PC
2. IR[8:0] + Sext
3. MAR
4. MDR
5. Memory
6. Register File

Indirect Addressing Mode

STI

1 0 1 1  Src  PCoffset9
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

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

Instruction Reg

IR[5:0]

Sext

Register File

Dst

Base

Memory

IR[5:0] + Base

MAR

MDR

[Diagram showing the LDR instruction flow]
Base + Offset Addressing Mode

STR

0 1 1 1  Src  Base  offset6

Register File

Memory

Instruction Reg

IR[5:0]

Sext

MAR

MDR
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 Dst PCoffset9
# 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 1 0</td>
<td>ADD</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>ST</td>
<td>M[PC - 5] ← R2 M[x30F4] ← x3102</td>
</tr>
<tr>
<td>x30F9</td>
<td>0 1 0 1 0 1 0 1 0 1 1 0 0 0 0 0</td>
<td>AND</td>
<td>R2 ← 0</td>
</tr>
<tr>
<td>x30FA</td>
<td>0 0 0 1 0 1 0 1 0 1 0 0 1 0 0 1 0 1</td>
<td>ADD</td>
<td>R2 ← R2 + 5 = 5</td>
</tr>
<tr>
<td>x30FB</td>
<td>0 1 1 1 0 1 0 0 0 1 0 1 0 0 1 1 1 0</td>
<td>STR</td>
<td>M[R1+14] ← R2 M[x3102] ← 5</td>
</tr>
<tr>
<td>x30FC</td>
<td>1 0 1 0 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1</td>
<td>LDI</td>
<td>R3 ← M[M[x30F4]] R3 ← M[x3102] R3 ← 5</td>
</tr>
</tbody>
</table>

*opcode*
Control Instructions

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

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) 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.

– Note: PC has already been incremented by FETCH stage.

– Note: Target must be within 256 words of BR instruction.
BR (PC-Relative)

BR

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

\[
\begin{array}{cccccccccccccc}
0 & 0 & 0 & 0 & n & z & p & PC_{offset9} \\
\end{array}
\]

Instructions
Example: Using a Branch

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

\[
\begin{align*}
R1 & \leftarrow x3100 \\
R3 & \leftarrow 0 \\
R2 & \leftarrow 12 \\
\end{align*}
\]

- **R2 = 0?**
  - **NO**
    - \( R4 \leftarrow M[R1] \)
    - \( R3 \leftarrow R3 + R4 \)
    - \( R1 \leftarrow R1 + 1 \)
    - \( R2 \leftarrow R2 - 1 \)
  - **YES**
### 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>x3000</td>
<td>1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1</td>
<td>LEA</td>
<td>$R1 \leftarrow x3100 \ (PC+0xFF)$</td>
</tr>
<tr>
<td>x3001</td>
<td>0 1 0 1 0 1 1 0 1 1 1 1 1 1 0 0</td>
<td>AND</td>
<td>$R3 \leftarrow 0$</td>
</tr>
<tr>
<td>x3002</td>
<td>0 1 0 1 0 1 0 0 1 0 1 0 1 0 0 0</td>
<td>AND</td>
<td>$R2 \leftarrow 0$</td>
</tr>
<tr>
<td>x3003</td>
<td>0 0 0 1 0 1 0 0 1 0 1 0 1 1 0 0</td>
<td>ADD</td>
<td>$R2 \leftarrow 12$</td>
</tr>
<tr>
<td>x3004</td>
<td>0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0</td>
<td>BRz</td>
<td>If Z, goto x300A \ (PC+5)</td>
</tr>
<tr>
<td>x3005</td>
<td>0 1 1 0 1 0 0 0 0 1 0 0 0 1 0 0 1</td>
<td>LDR</td>
<td>Load next value to R4</td>
</tr>
<tr>
<td>x3006</td>
<td>0 0 0 1 0 1 1 0 1 1 0 0 0 0 1 0 0</td>
<td>ADD</td>
<td>$R3 \leftarrow R4 + R3$</td>
</tr>
<tr>
<td>x3007</td>
<td>0 0 0 1 0 0 1 0 0 1 1 1 0 0 0 0 1</td>
<td>ADD</td>
<td>Increment R1 (pointer)</td>
</tr>
<tr>
<td>x3008</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>x3009</td>
<td>0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 0</td>
<td>BRnzp</td>
<td>Goto x3004 \ (PC-6)</td>
</tr>
</tbody>
</table>

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

**TRAP**

![TRAP Diagram]  

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
  (lets assume there will be less than 10 occurrences of the character)
Flow Chart

1. **Count = 0**  
   (R2 = 0)
2. **Ptr = 1st character of array**  
   (R3 = M[x3012])
3. **Input char from keybd**  
   (TRAP x23)
4. **Load char from array**  
   (R1 = M[R3])
5. **Done?**  
   (R1 ?= EOT)
   - **NO**
     - **Match?**  
       (R1 ?= R0)
       - **NO**
         - **Incr Count**  
           (R2 = R2 + 1)
         - **Load next char from array**  
           (R3 = R3 + 1, R1 = M[R3])
       - **YES**
         - **Convert count to ASCII character**  
           (R0 = x30, R0 = R2 + R0)
         - **Print count**  
           (TRAP x21)
9. **HALT**  
   (TRAP x25)

- **YES**
### Program (page 1 of 2)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>0 1 0 1 1 0 0 1 0 1 0 1 0 0 0 0</td>
<td>AND</td>
<td>R2 ← 0 (counter)</td>
</tr>
<tr>
<td>x3001</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[x3012] (ptr)</td>
</tr>
<tr>
<td>x3002</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 R0 (TRAP x23)</td>
</tr>
<tr>
<td>x3003</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>x3004</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>x3005</td>
<td>0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0</td>
<td>BRz</td>
<td>If Z, goto x300E</td>
</tr>
<tr>
<td>x3006</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>x3007</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>x3008</td>
<td>0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0</td>
<td>ADD</td>
<td>R1 ← R1 + R0</td>
</tr>
<tr>
<td>x3009</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 x300B</td>
</tr>
</tbody>
</table>
# Program (page 2 of 2)

<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 0 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 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 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 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 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 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 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1</td>
<td>TRAP</td>
<td>HALT (TRAP x25)</td>
</tr>
<tr>
<td>x3012</td>
<td>Starting Address of File</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3013</td>
<td>0 0 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)
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).