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 may 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/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:
  - $N = \text{negative } (< 0)$, $Z = \text{zero}$, $P = \text{positive } (> 0)$

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.

\[
\text{ADD } R_1 \, R_2 \, R_3 \quad \text{ADD } R_1 \, R_2 \, 6
\]

Will show dataflow diagram with each instruction.
  – illustrates when and where data moves to accomplish the desired operation
**NOT**

```
NOT  [1 0 0 1] Dst [1 1 1 1 1 1]
```

Note: works only with registers.

Note: Src and Dst could be the same register.
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
0 0 0 1 | Dst | Src1 | 1 | Imm5

AND
0 1 0 1 | Dst | Src1 | 1 | Imm5

Note: Immediate field is sign-extended.
Using Operate Instructions

With only ADD, AND, NOT...

\[ R_3 = R_1 - R_2 \]
- How do we subtract?
\[ R_2 = \overline{\text{NOT}(R_2)} \]
\[ R_2 = R_2 + 1 \]
\[ R_3 = R_1 + R_2 \]
- How do we OR?
\[ \overline{R_1 \cdot R_2} = R_1 + R_2 \]
- How do we copy from one register to another?
\[ R_3 = R_2 + O \]
- How do we initialize a register to zero?
\[ R_1 = R_1 \text{ AND } O \]
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)
Addressing Modes

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 ≤ offset ≤ +255

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

\[(PC - 256) ≤ X ≤ (PC + 255)\]

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

**ST** (Store Data)

<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>

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

Steps:
1. **PC**
2. **IR[8:0]**
3. **PCoffset9**
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

LDI

1 0 1 0

Dst

PCoffset9

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC

Register File

Memory

Sext

IR[8:0]

Instruction Reg

PC
Indirect Addressing Mode

STI

1 0 1 1

Src

PCoffset9

PC

Register File

Memory

Instruction Reg

Sext

IR[8:0]
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

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

Prof. Maxwell James Dunne – Fall 2015

CMPE-012/L
Base + Offset Addressing Mode

**STR**

<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:

1. Instruction Reg
2. Sext
3. Register File
4. Memory

IR[5:0] → Sext → Register File → MAR → MDR → Memory
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

Instructions
## 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 1 0 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 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 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 0 1 0 1 0 1 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 0 1 1 1 0 0 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 1</td>
<td>LDI</td>
<td>R3 ← M[M[x30F4]], R3 ← M[x3102], R3 ← 5</td>
</tr>
</tbody>
</table>

*opcode*
Instructions

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.

\[ \text{foo} = 0 \]
\[ \text{API, foo, foo} \neq 0 \]
Branch Instruction

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   0 0 0 0 0 n z p   PCoffset9
```

Diagram:
- Logic
- IR[11:9]
- Sext
- IR[8:0]
- PC
- Taken
- PCMUX
- Instruction Reg
- +

PC offset calculation:
1. IR[11:9]
2. Sext
3. IR[8:0]
4. Addition
5. Taken
Example: Using a Branch

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

\[
\begin{align*}
&1 \leftarrow 1 \\
&2 \leftarrow 3 \\
&3 \leftarrow 6 \\
&4 \leftarrow 10 \\
&5 \leftarrow 15 \\
&6 \leftarrow \ldots \\
&12 \leftarrow \ldots \\
\end{align*}
\]

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

R2=0?

\[
\begin{align*}
R4 &\leftarrow M[R1] \\
R3 &\leftarrow R3+R4 \\
R1 &\leftarrow R1+1 \\
R2 &\leftarrow R2-1
\end{align*}
\]

CMPE-012/L
### 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 ← x3100 (PC+0xFF)</td>
</tr>
<tr>
<td>x3001</td>
<td>0 1 0 1 0 1 1 0 1 1 1 1 1 0 0 0</td>
<td>AND</td>
<td>R3 ← 0</td>
</tr>
<tr>
<td>x3002</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</td>
</tr>
<tr>
<td>x3003</td>
<td>0 0 0 1 0 1 0 0 1 0 1 0 1 1 1 0</td>
<td>ADD</td>
<td>R2 ← 12</td>
</tr>
<tr>
<td>x3004</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 x300A (PC+5)</td>
</tr>
<tr>
<td>x3005</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>x3006</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>x3007</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>x3008</td>
<td>0 0 0 1 0 1 0 0 0 1 0 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 0 1 0</td>
<td>BRnzp</td>
<td>Goto x3004 (PC-6)</td>
</tr>
</tbody>
</table>
Instructions

JMP

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

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

![Diagram of JMP instruction]

**JMP**

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

---

Maxwell James Dunne – Fall 2015

CMPE-012/L
TRAP

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)
# 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>( \times 3000 )</td>
<td>0 1 0 1 1 0 0 0 1 0 1 0 0 0 0 0</td>
<td>AND</td>
<td>R2 ( \leftarrow 0 ) (counter)</td>
</tr>
<tr>
<td>( \times 3001 )</td>
<td>0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0</td>
<td>LD</td>
<td>R3 ( \leftarrow M[\times3012] ) (ptr)</td>
</tr>
<tr>
<td>( \times 3002 )</td>
<td>1 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 ( \times 23 ))</td>
</tr>
<tr>
<td>( \times 3003 )</td>
<td>0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 0</td>
<td>LDR</td>
<td>R1 ( \leftarrow M[R3] )</td>
</tr>
<tr>
<td>( \times 3004 )</td>
<td>0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0</td>
<td>ADD</td>
<td>R4 ( \leftarrow R1 - 4 ) (EOT)</td>
</tr>
<tr>
<td>( \times 3005 )</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 ( \times300E )</td>
</tr>
<tr>
<td>( \times 3006 )</td>
<td>1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1</td>
<td>NOT</td>
<td>R1 ( \leftarrow \text{NOT } R1 )</td>
</tr>
<tr>
<td>( \times 3007 )</td>
<td>0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1</td>
<td>ADD</td>
<td>R1 ( \leftarrow R1 + 1 )</td>
</tr>
<tr>
<td>( \times 3008 )</td>
<td>0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0</td>
<td>ADD</td>
<td>R1 ( \leftarrow R1 + RO )</td>
</tr>
<tr>
<td>( \times 3009 )</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 ( \times300B )</td>
</tr>
</tbody>
</table>

**opcode**

\( 0 \times 300B \)
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>×300A</td>
<td>0 0 0 1 1 0 0 1 0 1 1 0 0 0 0 1</td>
<td>ADD</td>
<td>R2 ← R2 + 1</td>
</tr>
<tr>
<td>×300B</td>
<td>0 0 0 1 1 1 0 1 1 1 1 0 0 0 0 1</td>
<td>ADD</td>
<td>R3 ← R3 + 1</td>
</tr>
<tr>
<td>×300C</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>×300D</td>
<td>0 0 0 0 1 1 1 1 1 1 1 1 0 1 1 0</td>
<td>BRnzp</td>
<td>Goto ×3004</td>
</tr>
<tr>
<td>×300E</td>
<td>0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0</td>
<td>LD</td>
<td>RO ← M[×3013]</td>
</tr>
<tr>
<td>×300F</td>
<td>0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0</td>
<td>ADD</td>
<td>RO ← RO + R2</td>
</tr>
<tr>
<td>×3010</td>
<td>1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1</td>
<td>TRAP</td>
<td>Print RO (TRAP ×21)</td>
</tr>
<tr>
<td>×3011</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 ×25)</td>
</tr>
<tr>
<td>×3012</td>
<td></td>
<td>Starting Address of File</td>
<td></td>
</tr>
<tr>
<td>×3013</td>
<td>0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0</td>
<td>Data</td>
<td>ASCII ×30 (‘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).