LC-3 Architecture

(Ch4’ish material)
CISC vs. RISC

CISC: Complex Instruction Set Computer
Lots of instructions of variable size, very memory optimal, typically less registers.

RISC: Reduced Instruction Set Computer; Less instructions, all of a fixed size, more registers, optimized for speed. Usually called a “Load/Store” architecture.
What is “Modern”

For embedded applications and for workstations there exist a wide variety of CISC and RISC and CISCy RISC and RISCy CISC. Most current PCs use the best of both worlds to achieve optimal performance.
LC-3 Architecture

- Very RISC, only 15 instructions
- 16-bit data and address
- 8 general purpose registers (GPR)
- Program Counter (PC)
- Instruction Register (IR)
- Condition Code Register (CC)
- Process Status Register (PSR)
Instruction Fetch / Execute Cycle

In addition to input & output a program also:

- Evaluates arithmetic & logical functions to determine values to assign to variable.
- Determines the order of execution of the statements in the program.

In assembly this distinction is captured in the notion of **Arithmetic, logical, and control** instructions.
Instruction Fetch / Execute Cycle

\[ a = b + c \quad a \leq c \]

**Arithmetic** and **logical** instructions evaluate variables and assign new values to variables.

**Control instructions** test or compare values of a variable and makes decisions about what instruction is to be executed next.

**Program Counter (PC)**

Basically the address at which the current executing instruction exists, or the next instruction.
Instruction Fetch / Execute Cycle

\[ c = a + b \]

1. load rega, 10
2. load regb, 20
3. add regc, rega, regb
4. beq regc, regd, 8
5. store regd, rege
6. store regc, regd
7. load regb, 15
8. load rega, 30

*Note: This is just pseudo assembly code
Instruction Fetch / Execute Cycle

\[ a = b + c, \text{PC} \]
\[ c = 2 \cdot a, \checkmark \]

The CPU begins the execution of an instruction by supplying the value of the PC to the memory & initiating a read operation (fetch).

The CPU “decodes” the instruction by identifying the opcode and the operands.

PC increments automatically unless a control instruction is used.
Instruction Fetch / Execute Cycle

For example:

PC → ADD A, B, C

ADD A, A, D

- CPU fetches instruction
- Decodes it and sees it is an “add” operation, needs to get values for the variables “B” & “C”
- Gets the variable “B” from a register or memory
- Does the same for variable “C”
- Does the “add” operation and stores the result in location register for variable “A”

\[ a = b + c \]
\[ a = c + d \]
Instruction Fetch / Execute Cycle

**Branch** – like a goto instruction, next instruction to be fetched & executed is an instruction other than the next in memory.

*Note: This is almost real LC-3 assembly*

```
ADD A, B, C
BRn fred
ADD A, D, 3
ADD A, D, 4
```

If A is negative then next instruction to be executed is at “fred”, which is just an address.
Breaking down an instruction

MAL has an 8-bit opcode. Variables a, b, and c each require an address to specify their memory location. If addresses are 32-bit then this is 104 bits, not a 32 bit instruction. How?

Memory accesses take more time than arithmetic and logical operations. Why? No memory fetches from slow, external parts.

If we have a 32-bit memory interface this operation would take 7 memory accesses. 4 to fetch the instruction (104 bits), one for each of the operands B and C and one more to store the result C.

\[
\begin{align*}
\text{ADD} & \quad a, b, c \\
\text{add} & \quad a \quad b \quad c \\
\text{Opcode} & \\
\text{Destination register} & \\
\text{Source registers/immediate} & \\
\end{align*}
\]

\[a = b + c\]
\[a = 5\]
The Stored Program Computer

1943: ENIAC
- Presper Eckert and John Mauchly -- first general electronic computer. (or was it John V. Atanasoff in 1939?)
- Hard-wired program -- settings of dials and switches.

1944: Beginnings of EDVAC
- among other improvements, includes program stored in memory

1945: John von Neumann
- wrote a report on the stored program concept, known as the First Draft of a Report on EDVAC
First Draft of a Report on EDVAC

The basic structure proposed in the draft became known as the “von Neumann machine” (or model).

This machine/model had five main components:

- a **memory**, containing instructions and data
- a **processing unit**, for performing arithmetic and logical operations
- a **control unit**, for interpreting instructions
- and **input** and **output** to get data into and out of the system.
Von Neumann Model*

* A slightly modified version of Von Neumann’s original diagram
Locality of reference

16-bit

We need techniques to reduce the instruction size. From observation of programs we see that a small and predictable set of variables tend to be referenced much more often than other variables.

Basically, locality is an indication that memory is not referenced randomly.

Odd MEM1, MEM2, MEM3
2^k x m array of stored bits:
- Address
  - unique (k-bit) identifier of location
- Contents
  - m-bit value stored in location

Basic Operations:
- LOAD
  - read a value from a memory location
- STORE
  - write a value to a memory location
Interface to Memory

How does the processing unit get data to/from memory?

**MAR**: Memory Address Register

**MDR**: Memory Data Register

To **LOAD** a location (A):

1. Write the address (A) into the MAR.
2. Send a “read” signal to the memory.
3. Read the data from MDR.

To **STORE** a value (X) to a location (A):

1. Write the data (X) to the MDR.
2. Write the address (A) into the MAR.
3. Send a “write” signal to the memory.
R/W: 0/1

2.2

A: 0/1
\[
\begin{array}{c|c|c}
\text{SR} & Q & X \\
\hline
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & L \\
1 & 1 & 1 \\
\end{array}
\]

\[D\text{ and } \overline{WE} = 0\]

\[\overline{D}\text{ and } WE = 0\]

\[
\begin{array}{c|c|c|c|c|c|c}
\text{WE} & \text{D} & \overline{Q} & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]
\[ S = 0 \]
\[ R = 1 \]
\[ S = 1 \]
\[ R = 0 \]

\[ \begin{array}{c}
S = 0 \\
R = 1 \\
S = 1 \\
R = 1
\end{array} \]

\[ \begin{array}{c}
1 \\
0 \\
1 \\
0
\end{array} \]

\[ \text{Nond} \]
\[ \begin{array}{ccc}
q & b & F \\
0 & 0 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0
\end{array} \]

\[ \overline{Q} \rightarrow R \]

\[ \overline{Q} \rightarrow \overline{Q} \]

\[ \overline{Q} \rightarrow 0 \]

\[ \overline{Q} \rightarrow 1 \]

\[ Q \rightarrow 0 \]

\[ Q \rightarrow 1 \]
Von Neumann Model

Processing Unit

Functional Units
- ALU = Arithmetic and Logic Unit
- could have many functional units, some of them special-purpose (multiply, square root, ...)
- LC-3 performs ADD, AND, NOT

Registers
- Small, temporary storage
- Operands and results of functional units
- LC-3 has eight registers (R0, ..., R7), each 16 bits wide

Word Size
- number of bits normally processed by ALU in one instruction
- also width of registers
- LC-3 is 16 bits

Input and Output

Devices for getting data into and out of computer memory

- LC-3 supports keyboard (input) and monitor (output)
- keyboard: data register (KBDR) and status register (KBSR)
- monitor: data register (DDR) and status register (DSR)

Some devices provide both input and output
- disk, network

The program that controls access to a device is usually called a driver.
Control Unit

Controls the execution of the program

Instruction Register (IR) contains the current instruction.

Program Counter (PC) contains the address of the next instruction to be executed.

Control unit:
- reads an instruction from memory
  - the instruction’s address is in the PC
- interprets the instruction, generating signals that tell the other components what to do
  - an instruction may take many machine cycles to complete
Instructions

The instruction is the fundamental unit of work.

Specifies two things:

- **opcode**: operation to be performed
- **operands**: data/locations to be used for operation

```
add r1, r2, r3
not
```
An instruction is encoded as a sequence of bits. *(Like data)*

– Often, but not always, instructions have a fixed length, such as 16 or 32 bits. (RISC vs. CISC)

– Control unit interprets instruction: generates sequence of control signals to carry out operation.

– Operation is either executed completely, or not at all.

A computer’s instructions and their formats is known as its *Instruction Set Architecture (ISA).*
Ex: LC-3 ADD Instruction

LC-3 has 16-bit instructions.
- Each instruction has a four-bit opcode, bits [15:12].
- LC-3 has 8 registers (R0-R7) for temp. storage.
- Sources and destination of ADD are registers.

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| ADD| Dst| Src1| 0 | 0 | 0 |Src2|
```

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

“Add the contents of R2 to the contents of R6, and store the result in R6.”
Ex: LC-3 LDR Instruction

Load instruction -- reads data from memory

Base + offset mode:
- add offset to base register - result is memory address
- load from memory address into destination register

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

```
0 1 1 0 0 1 0 0 1 1 0 0 0 1 1 0
```

“Add the value 6 to the contents of R3 to form a memory address. Load the contents of that memory location to R2.”
Instruction Processing

- Fetch instruction from memory
- Decode instruction
- Evaluate address
- Fetch operands from memory
- Execute operation
- Store result
FETCH

Load next instruction (at address stored in PC) from memory into Instruction Register (IR).

- Copy contents of PC into MAR.
- Send “read” signal to memory.
- Copy contents of MDR into IR.

Then increment PC, so that it points to the next instruction in sequence.

- PC becomes PC+1.
DECODING

0001

First identify the opcode.
- In LC-3, this is always the first four bits of instruction.
- A 4-to-16 decoder asserts a control line corresponding to the desired opcode.

Depending on opcode, identify other operands from the remaining bits.
- Example:
  - for LDR, last six bits is offset
  - for ADD, last three bits is source operand #2
EVALUATE ADDRESS

For instructions that require memory access, compute address used for access.

Examples:
- add offset to base register (as in LDR)
- add offset to PC
- add offset to zero
FETCH OPERANDS

Obtain source operands needed to perform operation.

Examples:
- load data from memory (LDR)
- read data from register file (ADD)
EXECUTE

Perform the operation, using the source operands.

Examples:
- send operands to ALU and assert ADD signal
- do nothing (e.g., for loads and stores)
STORE RESULT

Write results to destination. (register or memory)

Examples:

- result of ADD is placed in destination register
- result of memory load is placed in destination register
- for store instruction, data is stored to memory
  - write address to MAR, data to MDR
  - assert WRITE signal to memory
Changing the Sequence of Instructions

In the FETCH phase, we increment the Program Counter by 1.

What if we don’t want to always execute the instruction that follows this one?
- examples: loop, if-then, function call
We need special instructions that change the contents of the PC.

- **jumps** are unconditional -- they always change the PC
- **branches** are conditional -- they change the PC only if some condition is true (e.g., the result of an ADD is zero)
Ex: LC-3 JMP

Set the PC to the value contained in a register. This becomes the address of the next instruction to fetch.

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

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

“Load the contents of R3 into the PC.”
Instruction Processing Summary

Instructions look just like data -- it’s all interpretation.

Three basic kinds of instructions:
- computational instructions (ADD, AND, ...)
- data movement instructions (LD, ST, ...)
- control instructions (JMP, BRnz, ...)
Six basic phases of instruction processing:

F → D → EA → OP → EX → S

• Not all phases are needed by every instruction
• Phases may take more than 1 machine cycle
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).
LC-3 Overview

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

\[ \text{NOT} \quad 1 \ 0 \ 0 \ 1 \quad \text{Dst} \quad 1 \ 1 \ 1 \ 1 \ 1 \ 1 \]

\[ \text{NO} \rightarrow R1, R1 \]

Note: Src and Dst could be the same register.

Register File

Note: works only with registers.
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 R1, R1, R2

Register File

This zero means "register mode"
ADD

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Dst</th>
<th>Src1</th>
<th>Imm5</th>
</tr>
</thead>
</table>

AND

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Dst</th>
<th>Src1</th>
<th>Imm5</th>
</tr>
</thead>
</table>

Note: Immediate field is sign-extended.

AND R1, R1, 3

-2^4, -16, -15
Using Operate Instructions

With only ADD, AND, NOT...

- How do we subtract?
  \[ R_3 = R_1 - R_2 \]
  \[ R_2 = \text{Not}(R_2) \]
- 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)
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.
PC-Relative Addressing Mode

LD (Load 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>LD</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Dst</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>

1. Instruction Reg
2. Sext
3. IR[8:0]
4. Memory

Register File

MAR

MDR
PC-Relative Addressing Mode

ST (Store Data)

| ST  | 0 0 1 1 | Src | PCoffset9 |

Register File

Memory

Instruction Reg

①

Sext

IR[8:0]
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

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

Indirect Addressing Mode

STI 1 0 1 1 Src PCoffset9

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

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

<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>Dst</td>
<td></td>
<td></td>
<td>Base</td>
<td></td>
<td></td>
<td></td>
<td>offset6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Reg

1. Sext

2. MAR

3. MDR

Register File

Memory
Base + Offset Addressing Mode

STR

0 1 1 1 Src Base offset6

Register File

Memory

1. Sext
2. IR[5:0]
3. +
4. MDR

CMPE-012/L

Load Effective Address

Computes address like PC-relative (PC plus signed offset) and stores the result into a register.

Note: The \textit{address} is stored in the register, not the contents of the memory location.
LEA (Immediate)

LEA 1110 Dst PCoffset9

PC → IR[8:0] → Instruction Reg → Sext

PC → Register File → Dst → PC + Offset
## 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 0 1</td>
<td><strong>LEA</strong></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 0</td>
<td><strong>ADD</strong></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</td>
<td><strong>ST</strong></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 0 1 0 0 0</td>
<td><strong>AND</strong></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 0 1</td>
<td><strong>ADD</strong></td>
<td>R2 ← R2 + 5 = 5</td>
</tr>
<tr>
<td>x30FB</td>
<td>0 1 1 1 0 1 0 0 0 1 1 0 0 1 1 0</td>
<td><strong>STR</strong></td>
<td>M[R1+14] ← R2 M[x3102] ← 5</td>
</tr>
<tr>
<td>x30FC</td>
<td>1 0 1 0 0 0 1 1 1 1 1 1 0 1 1 1</td>
<td><strong>LDI</strong></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
Control Instructions

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.
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.
Instructions

BR (PC-Relative)

BR 0000nzp PCoffset9

1. PC
2. taken
3. PCMUX

Logic

IR[11:9]

Sext
IR[8:0]

Instruction Reg

+
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:
- \( R2 = 0? \)
  - NO: \( R4 \leftarrow M[R1] \)
  - YES: \( R3 \leftarrow R3 + R4 \)
  - \( R1 \leftarrow R1 + 1 \)
  - \( R2 \leftarrow R2 - 1 \)
### 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>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 0 0 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 0 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 1</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 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 1 0 1 1 1 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**
Instructions

JMP

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

```
JMP 11000000
```

![Diagram of PC and Register File]
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 (lets 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)

Match?
(R1 ≠ R0)

Incr Count
(R2 = R2 + 1)

Load next char from array
(R3 = R3 + 1, R1 = M[R3])

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

Print count
(TRAP x21)

HALT
(TRAP x25)
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>×3000</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>×3001</td>
<td>0 0 0 1 0 0 0 1 1 0 0 0 0 1 0 0 0 0</td>
<td>LD</td>
<td>R3 ← M[×3012] (ptr)</td>
</tr>
<tr>
<td>×3002</td>
<td>1 1 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1</td>
<td>TRAP</td>
<td>Input to RO (TRAP x23)</td>
</tr>
<tr>
<td>×3003</td>
<td>0 1 1 0 0 0 0 1 1 0 1 1 0 0 0 0 0 0</td>
<td>LDR</td>
<td>R1 ← M[R3]</td>
</tr>
<tr>
<td>×3004</td>
<td>0 0 0 1 1 0 0 0 0 0 1 1 1 1 0 0</td>
<td>ADD</td>
<td>R4 ← R1 + 4 (EOT)</td>
</tr>
<tr>
<td>×3005</td>
<td>0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0</td>
<td>BRz</td>
<td>If Z, goto ×300E</td>
</tr>
<tr>
<td>×3006</td>
<td>1 0 0 1 1 0 0 0 0 0 1 1 1 1 1 1</td>
<td>NOT</td>
<td>R1 ← NOT R1</td>
</tr>
<tr>
<td>×3007</td>
<td>0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 0 1</td>
<td>ADD</td>
<td>R1 ← R1 + 1</td>
</tr>
<tr>
<td>×3008</td>
<td>0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0</td>
<td>ADD</td>
<td>R1 ← R1 + RO</td>
</tr>
<tr>
<td>×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 ×300B</td>
</tr>
<tr>
<td>Address</td>
<td>Instruction Bits</td>
<td>Instruction</td>
<td>Comments</td>
</tr>
<tr>
<td>---------</td>
<td>------------------</td>
<td>-------------</td>
<td>----------</td>
</tr>
<tr>
<td>x300A</td>
<td>0 0 0 1 1 0 0 1 0 1 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 1 0 1 1 0 1 1 1 0 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 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 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 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 1 1 0 0 0 0</td>
<td>Data</td>
<td>ASCII x30 (‘0’)</td>
</tr>
</tbody>
</table>
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).