CMPE12 Notes

LC-3
Instruction Set Architecture

(Textbook’s Chapter 5 and 6)
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?
  - 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

Generally

- **access time**: several clock cycles
- **volatile**: loses content at power off

For LC-3

- **address space**: $2^{16}$ locations
  - **address bus**: 16 bits
- **addressability**: 16 bits per location
  - **data bus**: 16 bits
LC-3 Registers: GPRs

8 General-Purpose Registers in the CPU’s register file
- address space: $2^3$ locations
- addressability: 16 bits per register
- access time: 1 clock cycle
- volatile: lose content at power off
## LC-3 General Purpose Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0000</td>
</tr>
<tr>
<td>R1</td>
<td>0001</td>
</tr>
<tr>
<td>R2</td>
<td>0010</td>
</tr>
<tr>
<td>R3</td>
<td>0011</td>
</tr>
<tr>
<td>R4</td>
<td>0100</td>
</tr>
<tr>
<td>R5</td>
<td>0101</td>
</tr>
<tr>
<td>R6</td>
<td>0110</td>
</tr>
<tr>
<td>R7</td>
<td>0111</td>
</tr>
</tbody>
</table>
Registers: Special

- **PC**: Program Counter *(ISA)*
  - Points to the next instruction to execute (holds address of next instruction)

- **IR**: Instruction Register
  - Stores current instruction

- **MAR**: Memory Address Register
  - Address of current memory access

- **MDR**: Memory Data Register
  - Data to write to or read from memory

- **Condition codes register (ISA)**
  All are 16 bits wide, except the condition codes which is 3 bits wide
Instructions

What do instructions look like?

Mostly defines locations, or sometimes values.
Instruction Set Architecture

1) Opcodes
   – 16 opcodes (1 unused/reserved)
   – *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
   – A few miscellaneous “side-effects”: set/clear condition codes, based on result:
     • N = negative (< 0), Z = zero, P = positive (> 0)

2) Data Types
   – 16-bit 2’s complement integer

3) Addressing Modes
   – how operands are specified, or how the next instructions to execute is specified
Addressing Modes

1. REGISTER: a source or destination operand is specified as content of one of the registers R0–R7. Data: ADD, AND, NOT, LD, LDI, LDR, LEA, ST, STI, STR. Control: JMP, RET, JSRR.

2. IMMEDIATE: a numeric value embedded in the instruction is the actual operand. Data: ADD, AND, LEA. Control: TRAP.

3. PC-RELATIVE: a data or instruction memory location is specified as an offset relative to the incremented PC. Data: LD, ST. Control: BR, JSR.

4. BASE+OFFSET: a data or instruction memory location is specified as a signed offset from a base register. Data: LDR, STR. Control: RTI.

5. MEMORY-INDIRECT: a data memory location (specified as PC-RELATIVE) is a pointer to a data memory location. Data: LDI, STI.

You must know them intimately! (the addressing modes)
Register Addressing Mode

Syntax: ADD DR, SR1, SR2  Ex: ADD R1, R2, R7

R2 + R7 -> R1

00000000000000101 + 00…00 -> R1
Register Addressing Example

1. Where does the sum go?  
   A. register 3  
   B. location 011  
   C. Memory  
   D. Cannot tell from inst.

2. What is the sum?  
   A. 111+010  
   B. 9  
   C. 3  
   D. Cannot tell
Register and Immediate Addressing Modes

Syntax: ADD, DR, SR1, Imm5      Ex: ADD R0, R2, 1

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

ADD  DR  SRC1  Imm5

Designates as Immediate

\[R2 + XXXXX_{SX} \rightarrow R0\]
PC-Relative Addressing Mode

Syntax: BRx Label  
Ex: BRz Loop

2’s Comp. Sign Extended

PC* + XXXXXXXXXX_{SX} \rightarrow PC
**Base+Offset Addressing Mode**

Syntax: LDR DR, BaseR, offset6
Ex: LDR R4, R3,#5

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

LDR    DR    BaseR    offset6

2’s Comp. Sign Extended

Mem[BaseR + XXXXXX sx] -> DR
Indirect Addressing Mode

Syntax: LDI DR, LABEL  Ex: LDI, R5, LABEL

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

LDI  \quad DR  \quad \text{offset9} \quad \leftarrow

2’s Comp. Sign Extended

\text{mem[mem[PC* + XXXXXXXXXX_{sx}]]} \rightarrow \text{DR}
Some Practice

R0 = x3600
R1 = xA359
R2 = x3601
R3 = x0000
mem[x3600]=x4500
mem[x3601]=x3605
mem[x3602]=xABCD
mem[x3603]=x0000
A: mem[x3604]=xFFFE
mem[x3605]=x1118

ADD R5, R1, R2
ADD R2, R2, R2
ADD R5, R1, #-5
NOT R5, R1
AND R5, R1, R2
LDR R5, R2, #3
STR R0, R2, #4
LEA R5, A
LD R6, A
Full LC3 instruction set

Table in back of Book. Syntax for instructions: Appendix 1 (pp. 521-543)
Sample code:

```assembly
.orig x3000
AND R0, R0, #0
LD R1, UNO
LD R2, DUE
INTEL ADD R0, R0, R2
ADD R1, R1, #-1
BRZ AMD
BRNZP INTEL
AMD ST R0, TRE
HALT
UNO .FILL X000E
DUE .FILL X0003
TRE .FILL XFFFF
.end
```
Sample code in the simulator
Operate Instructions

Only three operations: **ADD, AND, NOT**

Source and destination operands are registers
- These instructions *do not* reference memory.
- **NOT** uses REGISTER addressing mode
- **ADD** and **AND** can use either
  - REGISTER addressing mode, when all operands are registers, or
  - IMMEDIATE + REGISTER addressing modes, where one of the source operands is an explicit number encoded within the instruction.
**NOT**

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

**Addressing mode(s):**
- REGISTER

Note: works only with registers.

```
NOT 1 0 0 1 Dst Src 1 1 1 1 1 1
```
This zero means "register mode"
ADD/AND

Addressing mode(s): REGISTER and IMMEDIATE

Note: Immediate field is sign-extended.

This one means "immediate mode"

[Diagram of an ALU with register file and instruction register]
Are these enough?

With only ADD, AND, 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**: memory-indirect mode

Store - write data from register to memory:
- **ST**: PC-relative mode
- **STR**: base+offset mode
- **STI**: memory-indirect mode

Load effective address - compute address, save in register
- **LEA**: PC-relative mode
  - does not access memory
PC-Relative Addressing mode

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)**

Addressing mode(s):
- **PC-RELATIVE (and REGISTER)**
- \( DR = M[PC + SX(PC\text{offset}9)] \)
Addressing mode(s):
PC-RELATIVE (and REGISTER)
M[PC+SX(PCoffset9)] = SR
Load Effective Address

Computes address as 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

Addressing mode(s):
IMMEDIATE (and REG.)
Dst = SX(PCoffset9)
Base + Offset addressing mode

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:
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.
Addressing mode(s):

BASE+OFFSET (and REG.)

DR ← M[Base+SX(IR[5:0])]]
Addressing mode(s):
BASE+OFFSET (and REG.)
M[Base+SX(IR[5:0])] <- Src
Memory-indirect addressing mode

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.
Addressing mode(s): MEMORY INDIRECT (and REG)
DR = M[M[PC+SX(PCoffset9)]]
Addressing mode(s):
MEMORY INDIRECT (and REG)
DR = M[M[PC+SX(PCoffset9)]]
Control Instructions

Used to alter the sequence of instructions. This is done by changing the PC. Remember: the PC holds the address of (points) to the next instruction.

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 next sequential inst.

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 in the condition code register:

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: target address is made by adding signed offset (IR[8:0]) to current PC.
  – Syntax: BR[n|z|p], BRNZP = BR = branch always
• 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.
Addressing mode:

PC-RELATIVE

PC = PC + SX(PCoffset9)
Example: Using a Branch

Compute sum of 12 integers (in C)

```c
count = 0;
array.length = 12;
For (i=0; i < array.length; ++i) {
    count = count + array[i];
}
(Next Instruction)
```

How do we do this in assembly?
Example: Using a Branch

Compute sum of 12 integers
Numbers start at location at the end of this short program. Program starts at location x3000.

Point to Array
Initialize i
Initialize count

End of Array?

NO
Count ← (array[i] + Count)
Increment i

YES

Done
Example: Using a Branch

Compute sum of 12 integers
Numbers start at location at the end of this short program. Program starts at location x3000.

R1 ← add(array)
R3 ← 0
R2 ← 12

R2=0?

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

NO

YES

Halt
Using a Branch: LC-3 source code

```assembly
.orig x3000
LEA R1, ARRAY
AND R3, R3, #0
AND R2, R2, #0
ADD R2, R2, #12

LOOP
BRz DONE
LDR R4, R1, #0
ADD R3, R3, R4
ADD R1, R1, #1
ADD R2, R2, #-1
BRNZP LOOP

DONE
HALT

ARRAY .blkw #12 #3

.end
```

R1 points to array element
R2 is index (counting down)
R3 holds sum
### Example: Using a Branch

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>1 1 1 0 0 0 0 0 1 0 1 0</td>
<td>LEA R1, #10</td>
<td></td>
</tr>
<tr>
<td>x3001</td>
<td>0 1 0 1 0 1 1 0 1 1 1 0 0 0 0</td>
<td>AND R3, R3, #0</td>
<td></td>
</tr>
<tr>
<td>x3002</td>
<td>0 1 0 1 0 1 0 0 1 0 1 0 0 0 0</td>
<td>AND R2, R2, #0</td>
<td></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 R2, R2, #12</td>
<td></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 #5</td>
<td></td>
</tr>
<tr>
<td>x3005</td>
<td>0 1 1 0 1 0 0 0 1 0 0 0 0 0 0</td>
<td>LDR R4, R1, #0</td>
<td></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 R3, R3, R4</td>
<td></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 R1, R1, #1</td>
<td></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 R2, R2, #-1</td>
<td></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></td>
</tr>
</tbody>
</table>
JMP

Jump is an unconditional branch -- \textit{always} taken.

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

\begin{itemize}
  \item \textbf{JMP:} \begin{array}{cccccccccccccc}
    15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
    1 & 1 & 0 & 0 & 0 & 0 & 0 & \textbf{Base} & 0 & 0 & 0 & 0 & 0 & 0 & 0
  \end{array}
\end{itemize}

\textit{Addressing mode:}

\begin{itemize}
  \item \textit{REGISTER}
  \item \textit{PC = Base}
\end{itemize}
JSR

*Jump to subroutine*, used to implement function calls (more on this later).

- Saves the current incremented PC in **R7**, used to return from the subroutine.
- Adds the 11-bit signed offset to the PC.

Addressing mode: 
**PC-RELATIVE**

\[ PC = PC + SX(PC_{offset11}) \]
**JSRR**

Jump to subroutine *register*, used to implement function calls. Same as **JSR** with different addressing mode

- Saves the current incremented PC in **R7**, used to return from the subroutine
- The PC gets the value specified by a register, **BaseR**

![Addressing mode:](image)

```assembly
0 1 0 0 0 0 0 BaseR 0 0 0 0 0 0
```

- **JSR**
- **R**
RET

Return from subroutine.

– Replaces the PC with the content of register \texttt{R7}.

– \texttt{RET} is just a different mnemonic for \texttt{JMP R7}, encoded exactly in the same way.

– Addressing mode is a special case of \texttt{REGISTER} with \texttt{REGISTER} = \texttt{R7} always.

1 1 0 0 0 0 0 1 1 1 0 0 0 0 0

Addressing mode:
\begin{align*}
\text{REGISTER} \\
(\text{special case with } \text{REGISTER} = \text{R7}) \\
\text{PC} = \text{R7}
\end{align*}
RTI

Return from interrupt.

- Replaces the PC with the content the memory location pointed to by $R6$ (the stack pointer).

Addressing mode(s):

$BASE + OFFSET$

*(Special case with $BASE = R6$ and $OFFSET = 0$)*

$PC = M[R6]$
TRAP

When routine is done, PC is set to the instruction following TRAP.

Calls a service routine, identified by 8-bit “trap vector.” Examples:

<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>

R7 <-- PC
PC <-- mem[ZEXT(trapvect8)]

When routine is done, PC is set to the instruction following TRAP.
TRAP

Jump to a TRAP subroutine, used to implement operating-system like function calls.

– Just like JSR/JSRR, saves the current incremented PC in R7, used to return from the TRAP subroutine
– The PC gets the value specified in a TRAP VECTOR, located in memory at the absolute address indicated in the immediate field
– Addressing mode is a special case of BASE+OFFSET with no base register and an unsigned offset.

Addressing mode(s):

BASE+OFFSET
(Special case with no base and unsigned offset)
PC = M[ZX(trapvect8)]
Structured Programming using Systematic Decomposition

Task

Sequential

First Task

Second Task

Conditional

Test

T

This

F

That

Iterative

Test

Sub Task

Not finished

Finished
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)
First level of Decomposition

Start

Initialize Everything

Input Character

Count Chars In File

Output Answer

Stop
Systematic Decomposition of “Count Chars In File”

- **Count Chars In File**
  - **Test**
  - **Sub Task**
  - **Are we done yet?**
  - **Get one char and increment counter if it is the one we are looking for.**
  - **iterative**
Flow Chart

Input character

Done?

Increment Count if match

Get next character

Get one char and increment counter if it is the one we are looking for

Count Characters in file

Output character
Flow Chart

Count = 0
(R2 = 0)

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

Input char from keybd
(TRAP x23)

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

Done?
(R1 ?= EOT)

YES

Match?
(R1 ?= R0)

NO

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)

Get one char and increment counter if it is the one we are looking for.

Count characters in file
Char count program (1/2)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>0101010010100000</td>
<td>AND R2,R2,#0</td>
<td>R2 holds count</td>
</tr>
<tr>
<td>x3001</td>
<td>0010011100010000</td>
<td>LD R3,Array</td>
<td>Array[0] at x3012</td>
</tr>
<tr>
<td>x3002</td>
<td>1111000000100011</td>
<td>TRAP x23</td>
<td>Input from keyboard to R0</td>
</tr>
<tr>
<td>x3003</td>
<td>0110001011000000</td>
<td>LDR R1,R3,#0</td>
<td>R1 has array[x]</td>
</tr>
<tr>
<td>x3004</td>
<td>0001100011111100</td>
<td>ADD R4,R1,#-4</td>
<td></td>
</tr>
<tr>
<td>x3005</td>
<td>0000010000000100</td>
<td>BRz Convert</td>
<td>If array[x] = 4, convert</td>
</tr>
<tr>
<td>x3006</td>
<td>1001001001111111</td>
<td>NOT R1,R1</td>
<td></td>
</tr>
<tr>
<td>x3007</td>
<td>0001001001100001</td>
<td>ADD R1,R1,+1</td>
<td>R1 has –array[x]</td>
</tr>
<tr>
<td>x3008</td>
<td>0001001001000000</td>
<td>ADD R1,R1,R0</td>
<td></td>
</tr>
<tr>
<td>x3009</td>
<td>0000101000000001</td>
<td>BRnp</td>
<td>Skip instruct if R0 != R1</td>
</tr>
</tbody>
</table>

**opcode**
# Char count program (2/2)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x300A</td>
<td>00010100101000001</td>
<td>ADD R2,R2,#1</td>
<td>count &lt;- count+1</td>
</tr>
<tr>
<td>x300B</td>
<td>00010110111000001</td>
<td>ADD R3,R3,#1</td>
<td>x &lt;- x+1</td>
</tr>
<tr>
<td>x300C</td>
<td>01100010110000000</td>
<td>LDR R1,R3,#0</td>
<td>Do you need?</td>
</tr>
<tr>
<td>x300D</td>
<td>00001111111110110</td>
<td>BRnzp</td>
<td>PC &lt;- location x3004</td>
</tr>
<tr>
<td>x300E</td>
<td>00100000000000100</td>
<td>LD R0,Array[1]</td>
<td>Convert: R0&lt;-“0”</td>
</tr>
<tr>
<td>x300F</td>
<td>00010000000000010</td>
<td>ADD R0,R0,R2</td>
<td>count converted to ascii</td>
</tr>
<tr>
<td>x3010</td>
<td>11110000001000001</td>
<td>TRAP x21</td>
<td>page 543</td>
</tr>
<tr>
<td>x3011</td>
<td>11110000001001011</td>
<td>TRAP x25</td>
<td></td>
</tr>
<tr>
<td>X3012</td>
<td>00000000000101000</td>
<td>Data</td>
<td>Starting Address of Array</td>
</tr>
<tr>
<td>x3013</td>
<td>00000000000110000</td>
<td>Data</td>
<td>Ascii 0</td>
</tr>
</tbody>
</table>

*opcode*
Exercises
black to turn in, red for practice

- Ex 5.2 5.4, 5.8, 5.9, 5.11, 5.13, 5.14, 5.15
- Ex 5.16, 5.22, 5.23, 5.24, 5.25, 5.30, 5.31, 5.32
- Ex 5.40, 5.41, 5.42
- Ex 6.5, 6.16, review slides 64, 65 and fully understand them.
Practice

mem[x3000]= LEA R2,I
mem[x3001]= LD R3,I
mem[x3002]= LDR R4,R2,#1

1. R2 gets
2. R3 gets
3. R4 gets