Learning Compiler 3

LC-3

Instruction Set Architecture and Beginning LC-3 Programming
Summer

Onebit register

onebit adder

clk

ew
display it
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.
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 (what commands can we give to the computer)
  - 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 Architecture

- Very RISC, only 15 instructions
- 16-bit data and address
- 8 general purpose registers (GPR)

Architecture
- Program Counter (PC)
- Instruction Register (IR)
- Condition Code Register (CC)
- Process Status Register (PSR)
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/effectuated by instructions
  - PC (program counter), condition codes
I7: 16 registers
Cache: 8MB
RAM: 9x slower
SSD: 100's slower
SM: 1000's
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**
Hello World

- Traditional First program on a system
  - Can be difficult to get to __ 2 weeks

.ORIG x3000
LEA R0, HELLO
PUTS
HALT
HELLO .STRINGZ "Hello CMPE12"
.END

Print("Hello world")
Syntax of LC-3

- One instruction, declaration per line
- Comments are anything on a line following ";"
- Comments may not span lines

```
ADD R0,R0,R0 ; Hi;
```
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.
NOT

- Takes the bitwise not of the SRC and puts it in the DST.
- Note: SRC and DST could be the same register.

NOT DST, SRC
NOT R0,R1

R0 = \overline{R1}
ADD/AND

- Takes the addition/and of SRC1 and SRC2 and puts it in the DST.
- Note: All three could be the same register.

ADD DST, SRC1, SRC2
ADD R6, R1, R2

R6 = R1 + R2
ADD/AND (with constants)

- Takes the addition/and of SRC1 and constant and puts it in the DST.
- Note: All three could be the same register.

\[
\begin{align*}
\text{ADD DST, SRC1,4} & \quad \text{CONSTRANT} \\
\text{ADD R2, R3,6} & \quad 2SC \ 5\text{-bit number}
\end{align*}
\]
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?

\[
\begin{align*}
R_3 &= R_1 - R_2 \\
R_2 &= \overline{R_2} \\
R_1 &= R_2 + 1 \\
R_3 &= R_1 + R_2 \\
\text{AND } R_3, R_2, R_2 \\
\text{AND } R_1, R_1, 0
\end{align*}
\]
reset the processor

\[
x = 3
\]
reset

\[
x = 3
\]
reset

\[
x = 4
\]
reset

resetting 1000's of times a second

secure delete

nothing

Once

7-9 times
Data Movement Instructions

Load -- read data from memory to register
  - LD
  - LDR

Store -- write data from register to memory
  - ST
  - STR

Load effective address -- compute address, save in register
  - LEA

LDI, and STI will be covered when we go over the architecture.

We will use labels instead for now.
Labels

- Symbolic names that are used to identify memory locations
- Location for target of a branch or jump
- Location for a variable for loading and storing
- Can be 1-20 characters in size
- We start at address 0x300 by convention

LEA R0, HELLO
LD (Load Data)

- Loads the contents of LABEL and stores it in DST

LD DST, LABEL
LD R3, FOO
SD (Store Data)

- Stores the contents of SRC in LABEL

SD SRC, LABEL
SD R3, FOO
Load Effective Address

Computes a memory location from LABEL and stores it in DST.
We use it a lot for output

LEA DST, LABEL
LEA R0,HELLO
LDR (Load Data with Register)

- Use SRC as memory address and adds OFFSET to it. The contents of this new address is then stored in DST.
  - Offset can be 0

LDR DST, SRC, OFFSET
LDR R3, R0, 2

LEA R0, Hello
LDR R0, R0, 0
LD R0, Hello
SDR (Store Data with register)

- Use DST as memory address and adds OFFSET to it. This new memory address has SRC stored in it.

```
SDR SRC, DST, OFFSET
SD R1,R2,0
```
TRAP
(System Calls)

• Very tedious and dangerous for a programmer to deal with IO at the OS level.
• Need an instruction though to get the attention of the OS.

Use the “TRAP” instruction and a “trap vector”.
### Trap Service Routines

<table>
<thead>
<tr>
<th>Trap Vector</th>
<th>Assembler Name</th>
<th>Usage &amp; Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>GETC</td>
<td>Read a character from console into R0, not echoed.</td>
</tr>
<tr>
<td>0x21</td>
<td>OUT</td>
<td>Write character in R0 to console.</td>
</tr>
<tr>
<td>0x22</td>
<td>PUTS</td>
<td>Write string of characters to console. Start with character at address contained in R0. Stops when 0x0000 is encountered.</td>
</tr>
<tr>
<td>0x23</td>
<td>IN</td>
<td>Print a prompt to console and read in a single character into R0. Character is echoed.</td>
</tr>
<tr>
<td>0x24</td>
<td>PUTSP</td>
<td>Write a string of characters to console, 2 characters per address location. Start with characters at address in R0. First [7:0] and then [15:0]. Stops when 0x0000 is encountered.</td>
</tr>
<tr>
<td>0x25</td>
<td>HALT</td>
<td>Halt execution and print message to console.</td>
</tr>
</tbody>
</table>
To print a character

; the char must be in R0.
TRAP x21
or
OUT

To read in a character

; will go into R0, no echo.
TRAP x20
or
GETC
To end your program:

TRAP  x25

or

HALT
**Directives** give information to the assembler. All directives start with ‘.’ (period)

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ORIG</td>
<td>Always 0x3000 for now (the start of our program)</td>
</tr>
<tr>
<td>.FILL</td>
<td>Declare a memory location</td>
</tr>
<tr>
<td>.BLKW</td>
<td>Reserve a group of memory locations</td>
</tr>
<tr>
<td>.STRINGZ</td>
<td>Declare a group of characters in memory</td>
</tr>
<tr>
<td>.END</td>
<td>Tells assembly where your program source ends</td>
</tr>
</tbody>
</table>
Hello World (again)

- Traditional First program on a system
  - Can be difficult to get to
  
    .ORIG x3000
    LEA R0, HELLO
    PUTS
    HALT

    HELLO .STRINGZ "Hello CMPE12"
    .END
Control Instructions

Used to alter the sequence of instructions. This allows us to move to a particular instruction.

Conditional Branch

– branch is *taken* if a specified condition is true
– else, the branch is *not taken*

• next sequential instruction is executed

```
if x > 3
```
Unconditional Branch (or Jump)

- always changes instruction

TRAP

- changes to 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.

There are hardware limits on how far you can branch.
BR (Branch)

- It does no computation, only looks at condition codes
- If condition code is set, go to LABEL, can combine codes
  - BRz
  - BRn
  - BRp
  - BRzp

ADD R0,R1,R2
BRz FOO ; if zero we go to label FOO
BR (unconditionally)

- Degenerate case, always goes to LABEL
  - BRnzp

BRnzp FOO
BRnzp FOO ; always go to label FOO

Start.
"Hello"
CHECK POWER AND GROUND 40%
Example: Using a Branch

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

\[ R2 = 12 \]
\[ \text{Zero} \]
**Example: Using a Branch**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA R1,NUMS</td>
<td>R1 ← NUMS</td>
</tr>
<tr>
<td>AND R3,R3,0</td>
<td>R3 ← 0</td>
</tr>
<tr>
<td>AND R2,R2,0</td>
<td>R2 ← 0</td>
</tr>
<tr>
<td>ADD R2,R2,12</td>
<td>R2 ← 12</td>
</tr>
<tr>
<td><strong>START BRz END</strong></td>
<td>If Z, goto END</td>
</tr>
<tr>
<td>LDR R4,R1,0</td>
<td>Load next value to R4</td>
</tr>
<tr>
<td>ADD R3,R4,R3</td>
<td>R3 ← R4 + R3</td>
</tr>
<tr>
<td>ADD R1,R1,1</td>
<td>Increment R1 (pointer)</td>
</tr>
<tr>
<td><strong>ADD R2,R2,-1</strong></td>
<td>Decrement R2 (counter)</td>
</tr>
<tr>
<td><strong>BRnzp START</strong></td>
<td>Goto START</td>
</tr>
<tr>
<td><strong>END</strong></td>
<td># done adding</td>
</tr>
</tbody>
</table>
\textbf{AND} R3, R3, R3

decimal to binar+ converter
Instructions

JMP

Jump is an unconditional branch -- always taken.
   – Target is contents of a register, not a label.

JMP R1
Print Single Digit Number
LC-3 Multiply
LC-3 Integer Division