Scanf

Abstraction and Computing Systems
Computing machines are everywhere

• General purpose
  – Servers, desktops, laptops, tablets, smart phones, etc.

• Special purpose
  – Cash registers, ATMs, games, telephone switches, etc.

• Embedded
  – Cars, hotel doors, printers, VCRs, industrial machinery, medical equipment, etc.

6 components / computer
Computing machines: distinguishing features

- Speed: 40 MHz
- Cost: 1¢ vs. $100,000
- Price/performance
- Ease of use, software support & interface
- Scalability
- Power
- Size
10mil

001

10mil

002

10mil

003

10mil

P 1
Computing System
1st Very Important Idea

• Universal Computational Devices
  - Given enough time and memory, all computers are capable of computing exactly the same things
  - Irrespective of speed, size, or cost

• Turing’s Thesis
  - Every computation can be performed by some Turing Machine – a theoretical universal computational device
Alan Turing’s original model

(1912-1954)
A Turing Machine

Also known as a *Universal Computational Device*: a theoretical device that accepts both input data and instructions on how to operate on the data.
2nd Very Important Idea

• Problem Transformation
  – The ultimate objective is to transform a problem expressed in natural language into electrons running around a circuit

• This is computer science and computer engineering
  – A continuum that embraces software and hardware

< CSE
E E C E
Computer Architecture

- Problems
- Algorithms
- Language
  - Instruction Set Architecture
  - Microarchitecture
- Circuits
- Devices
Computer Science

**Definition:** The study of algorithms and data structures to solve problems.

**Abstraction:** Use of level of abstraction in software design allows the programmer to focus on a critical set of problems without having to deal with irrelevant details.
Procedure or Function

```
int average (a, b)
    begin
        int avg;
        avg = (a+b)/2;
        return (avg);
    end
```

```
main ()
    ...
    x = 4;
    y = 2;
    k = average (x, y);
    print ("8d", k);
    ...
```
**Programming Flow**

- **High Level Language** → **Compiler** → **Assembly Language** → **Assembler** → **Machine Language**

**Compiler**: A computer program that translates code written in a high level language into an intermediate level abstract language.

**Assembler**: A computer program that translates code written in assembly language to the binary form that the CPU can execute.
Computer Engineering

Definition: The creative application of engineering principles and methods to the design and development of hardware and software systems.

Abstraction: Use of level of abstraction in hardware design allows the designer to focus on a critical set of problems without having to deal with irrelevant details.
Instruction Set Architecture (ISA)

Definition: Interface between a computer’s hardware and its software. Defines exactly what the computer’s instructions do, and how they are specified.
Central Processing Unit

The heart of computing systems

ca 1980
It took 10 of these boards to make a Central Processing Unit (CPU)

ca 2000
No wonder they called this CPU a microprocessor!
Motherboard: System
CPU: Package

$1500

1.5"
SoC – System on a chip

800 PROCESSOR

Krait 400 CPU features 28HPm process technology superior 2GHz+ performance

Adreno 330 for advanced graphics

Hexagon QDSP6 for ultra low power applications and custom programmability

Integrated LTE, 802.11ac, USB 3.0 and BT 4.0 offers broad array of high speed connectivity

MULTIMEDIA
Audio, Video and Gestures

CAMERA

DISPLAY/LCD

NAVIGATION

CONNECTIVITY
4G LTE, WiFi, USB, BT and FM

Ultra HD Capture and Playback
DTS-HD and Dolby Digital Plus audio
Expanded Gestures

55MP with dual ISP
Support for up to 2560x2048 display
Miracast 1080p HD support
IZat GHSS with support for three GPS constellations

CMPE-012/L
Maxwell James Dunne
CPU: Microarchitecture

Intel Core i7
CPU: Die

Memory Controller

Core

Core

Queue

Core

Core

Shared L3 Cache

Intel Core i7
CPU: Die with graphics core

Intel® Core™ M Processor Die Map
14nm 2nd Generation Tri-Gate 3-D Transistors

Processor Graphics

Dual Core Die Shown Above

Transistor Count: 1.3 Billion
4th Gen Core Processor (Y series): .96B
** Cache is shared across both cores and processor graphics

Shared L3 Cache**

Memory Controller I/O

System Agent, Display Engine & Memory Controller

Die Size: 82mm²
4th Gen Core Processor (Y series): 131mm²

Intel Confidential – UNDER EMBARGO UNTIL SEPTEMBER 5TH 2014 8:30AM PT
*Other names and brands may be claimed as the property of others

All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.
Two recurring themes

1) Abstraction

– The notion that we can concentrate on one “level” of the big picture at a time, with confidence that we can then connect effectively with the levels above and below.

– Framing the levels of abstraction appropriately is one of the most important skills in any undertaking.
Two recurring themes

2) Hardware vs. Software

– On the other hand, abstraction does not mean being clueless about the neighboring levels.
– In particular, hardware and software are inseparably connected, especially at the level we will be studying.
What is Computer Organization?

Electronic Devices → Desired Behavior

There is a fundamentally wide gap between the intended behavior desired and the workings of the electronic devices that do the work.

Before the digital computers of today special purpose analog devices (mechanical, electrical, or electronic) were built for each desired behavior.
A general purpose computer is the bridge that links the desired behavior (application) and the basic building blocks (electronic devices).
video
Our computer model for now

CPU Interacts with the memory in 3 ways:
- fetches instructions
- loads the value of a variable
- stores the new value of a variable

Memory is capable of only 2 operations:
- reads – a load or a fetch
- writes – operation of storing the value of a variable
Our LC-3 Model

MEMORY

registers

ALU
LC-3
Instruction Set Architecture and Beginning LC-3 Programming
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/effect by instructions
  - PC (program counter), condition codes
CPU
- registers

Memory

Hard drive
- SSD

You
- flash cards

Bookshelf

Next Door
- nearest library

Library of Congress

Moon
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
    
```
.ORIG x3000
LEA R0, HELLO
Puts
HALT
HELLO .STRINGZ "Hello CMPE12"
.END
```

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

; add x to itself
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
```
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.

ADD DST, SRC1,4
ADD R2, R3, 6

\[ R2 = R3 + 6 \]
Using Operate Instructions

With only ADD, AND, NOT...

– How do we subtract? \( R_3 = R_1 - R_2 \)

– How do we OR?

Demorgon's

– How do we copy from one register to another?

– How do we initialize a register to zero?
Using Operate Instructions

With only ADD, AND, NOT...

- How do we subtract?

- How do we OR?

Demorgon's

- 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
- 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 0x3000 by convention

LEA R0, HELLO
X3000  ADD R1, R2, R2
X3001  NOT R1, R2, RG
X3002  AND R2, R3, RG
X3003  \textit{Hello}
X3004
LD (Load Data)

- Loads the contents of LABEL and stores it in DST

LD DST, LABEL
LD R3, FOO

$R3 = \text{Mem}[\text{Foo}]$
SD (Store Data)

- Stores the contents of SRC in LABEL

\[
\text{SD SRC, LABEL} \\
\text{SD R3, FOO} \\
\text{Mem [FOO]} = R3
\]
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
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
```