Abstraction and Computing Systems

Textbook Chapter 1
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.
Computing machines: distinguishing features

- Speed
- Cost
- Price/performance
- Ease of use, software support & interface
- Scalability
- Power
- Size
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.
2\textsuperscript{nd} 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
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 ("%d", k);
    ...
```
Programming Flow

**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!
CPU: Package

Intel Core™ i7
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

KRAIT CPU
ADRENO GPU
HEXAGON DSP

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
Miicast 1080p HD support
IZat GHSS with support for three GPS constellations

CMPE-012/L
CPU: Microarchitecture

Intel Core i7
CPU: Die

Intel Core i7
CPU: Die with graphics core

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

Dual Core Die Shown Above

Processor Graphics

Core

Core

System Agent, Display Engine & Memory Controller

Shared L3 Cache**

Memory Controller I/O

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

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?

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) where built for each desired behavior.
Role of General Purpose Computers

A general purpose computer is the bridge that links the desired behavior (application) and the basic building blocks (electronic devices).
Our computer model for now

- CPU
  - Control info
  - Write data
  - Read data
- Memory

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
Number Systems, Again

(ch 2 +)
Positional Fractions

Mesopotamians used positional fractions

$$\sqrt{2} = 1.24,51,10_{60} = 1 \times 60^0 + 24 \times 60^{-1} + 51 \times 60^{-2} + 10 \times 60^{-3}$$

$$= 1.414222$$

Most accurate approximation until the Renaissance
Generalized Representation

For a number “f” with “n” digits to the left and “m” to the right of the decimal place.

Position is the power

Decimal point
Fractional Representation

- What is $3E.8F_{16}$?
  \[= 3 \times 16^1 + E \times 16^0 + 8 \times 16^{-1} + F \times 16^{-2}\]
  \[= 48 + 14 + \frac{8}{16} + \frac{15}{256}\]

- How about $10.101_2$?
  \[= 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}\]
  \[= 2 + 0 + \frac{1}{2} + \frac{1}{8}\]
Converting Decimal -> Binary fractions

- Consider left and right of the decimal point separately.
- The stuff to the left can be converted to binary as before.
- Use the following table/algorithm to convert the fraction
For $0.8_{10}$ to binary

<table>
<thead>
<tr>
<th>Fraction</th>
<th>Fraction x 2</th>
<th>Digit left of decimal point</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.8</td>
<td>1.6</td>
<td>1 ← most significant ($f_{-1}$)</td>
</tr>
<tr>
<td>0.6</td>
<td>1.2</td>
<td>1</td>
</tr>
<tr>
<td>0.2</td>
<td>0.4</td>
<td>0</td>
</tr>
<tr>
<td>0.4</td>
<td>0.8</td>
<td>0</td>
</tr>
<tr>
<td>0.8</td>
<td>(it must (\downarrow), repeat from here!!)</td>
<td>(\downarrow)</td>
</tr>
</tbody>
</table>

- Different bases have different repeating fractions.
- \(0.8_{10} = 0.110011001100\ldots_2 = 0.1100_2\)
- Numbers can repeat in one base and not in another.
What is $2.2_{10}$ in:

- **Binary**: $10.0011$
- **Hex**: $2.3333_{16}$

$\begin{array}{rl}
.2 \times 2 &= 0.4 \\
.2 \times 2 &= 0.8 \\
.2 \times 2 &= 1.6 \text{ (or } 1 + .6) \\
.2 \times 2 &= 1.2 \text{ (or } 1 + .2)
\end{array}$
Binary Division Example

\[
\begin{array}{c}
11.1110 \\ \hline \\
11 | 10 \ 11 \ \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \\
\hline \\
10 \\
\hline \\
101 \\
\hline \\
10 \\
\hline \\
100 \\
\hline \\
10 \\
\hline \\
100 \\
\hline \\
0 \\
\hline \\
1 \\
\hline \\
4 - (2 + 1) \\
\hline \\
100 \\
\hline \\
11 \\
\hline \\
1 \\
\hline
\end{array}
\]
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

**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

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
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

- 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”
Instruction Fetch / Execute Cycle

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fred</td>
<td>ADD A, D, 4</td>
<td></td>
</tr>
<tr>
<td>fred</td>
<td>ADD A, D, 3</td>
<td></td>
</tr>
<tr>
<td>BRn  fred</td>
<td>BRn fred</td>
<td></td>
</tr>
<tr>
<td>ADD A, B, C</td>
<td>ADD A, B, C</td>
<td></td>
</tr>
</tbody>
</table>

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

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

ADD a, b, c
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

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.

This is where the use of registers comes into play.
Von Neumann Model

Memory

$2^k \times m$ 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.
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

Maxwell James Dunne – Fall 2015
Von Neumann Model

Input and Output

Devices for getting data into and out of computer memory

Each device has its own interface, usually a set of registers like the memory’s MAR and MDR

- 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*. 
Von Neumann Model

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
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 | 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 | 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

```
<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>LDR</td>
<td>Dst</td>
<td>Base</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

“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.
DECODE

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
Instruction Processing

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.

These are those *control instructions* from before.

- **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 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 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