ECSE324 : Computer Organization

Computer Technology and Abstractions

Textbook§Chapters 1 and 2

Christophe Dubach
Fall 2021

Revision history: Warren Gross – 2017, Christophe Dubach – W202&F2020, Brett H. Meyer – W2021, Christophe Dubach – F2021
Some material from Hamacher, Vranesic, Zaky, and Manjikian, *Computer Organization and Embedded Systems, 6 th ed*, 2012, McGraw Hill and Patterson and Hennessy, *Computer Organization and Design, ARM Edition*, Morgan Kaufmann, 2017, and notes by A. Moshovos

Timestamp: 2021/09/03 08:42:00

Disclaimer

Lectures are recorded live and posted unedited on *MyCourses* on the same day.

It is possible (and even likely) that I will (sometimes) make mistakes and give incorrect information during the live lectures. If you have any doubts, please check the textbook, or ask on the online forum for clarification.

Introduction

Introduction

A brief history of computer technology

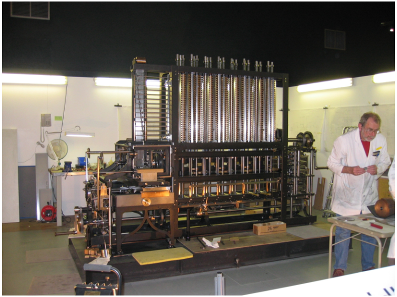
Textbook§1.6, 1.7

Mechanical Computers

Charles Babbage's Difference Engine, 1822

A mechanical special-purpose computer designed to calculate polynomials using numerical difference method.

- Number of parts: 25,000
- Cost: £17,470 ≅ CAD$3M today
- Babbage never completed it
- A working version was finished in 1991 (London Science museum)



https://youtu.be/KBuJqUfO4-w?t=203

The First Program(mer)

- Babbage saw his Difference Engine as a way to simplify the calculation of mathematical (*e.g.*, actuarial) tables.
- Ada Lovelace saw its greater potential. In 1843, she wrote a program for it that calculates Bernoulli numbers.
- She didn't stop there, speculating about universal computation, and even artificial intelligence.

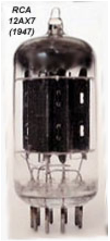
Ada Lovelace, 1815–1852

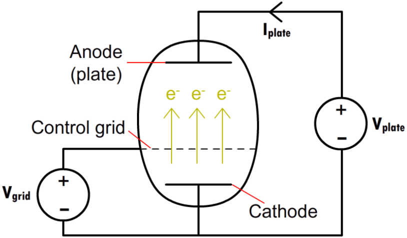
"We might even invent laws for series or formulæ in an arbitrary manner, and set the engine to work upon them, and thus deduce numerical results which we might not otherwise have thought of obtaining."

Vacuum Tubes ($\sim$ 1904-06)

Act like an amplifier (make weak signals stronger)
or a switch (start and stop flow of electricity, very quickly).

If you have a very fast and small switch, you can implement efficient logic gates





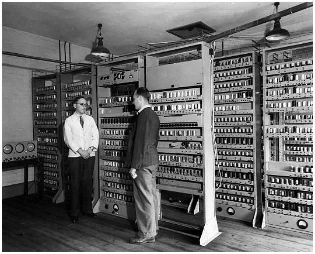
source: www.engineering.com

First Generation Electrical Computers: 1940's

Vacuum tubes = enabling technology

- 1000x faster than mechanical computers
- Programming was done at the machine level in machine language or "assembly language"

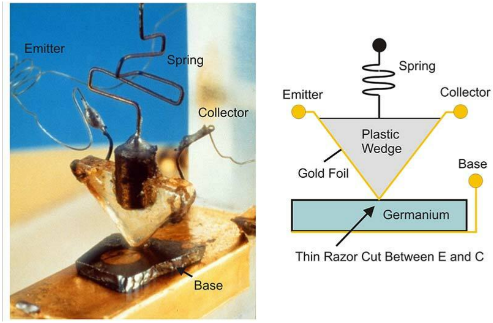
Stored-program computers were a revolutionary concept and the basis for today's computers. Programs and data stored in same memory!



EDSAC, 1949,
University of Cambridge, UK

https://youtu.be/2iPrFEC7Vhg?t=101

Transistors: 1948



Discrete transistors

First transistor, Bell Labs, 1948

source: www.nutsvolts.com

Replaced the large, fragile, power-hungry, and slower vacuum tubes.

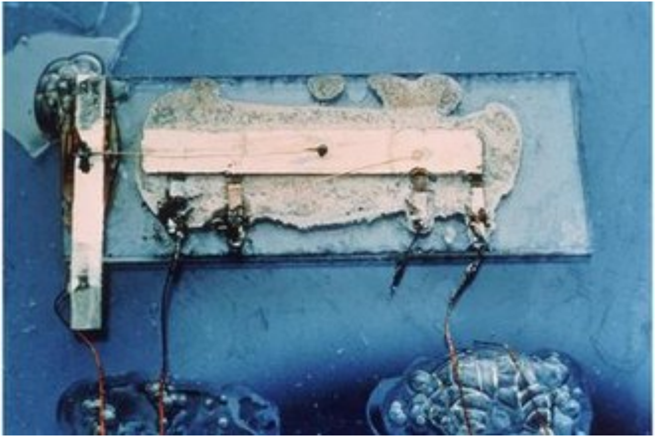
Second Generation (1950s), Transistor-based Computers

First high-level programming languages used: FORTRAN (John Backus)

First compilers developed to translate high-level programs into assembly

IBM 7070, 1958

Integrated Circuits (IC)



First integrated circuit (Phase shift oscillator). Jack Kilby, Texas Instruments (1958), Nobel Prize in 2000.

Third Generation (1960s): Integrated-circuits Computers

Many of today’s ideas in computer organization first appeared:

- Parallelism
- Pipelining
- Cache and virtual memory

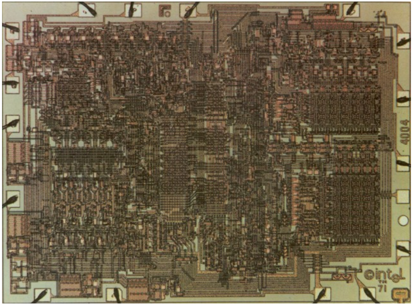
Operating systems allowed several programs to run on a single machine

DEC PDP-8 (1965)

Fourth Generation (1970s): Large Scale Integration

Large Scale Integration

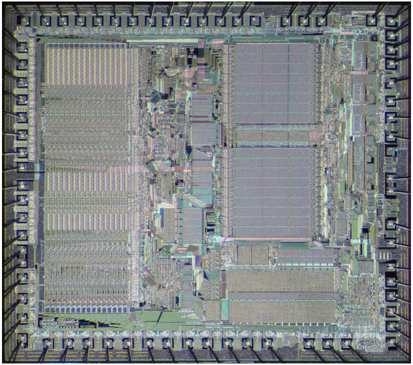
Whole CPU on a single chip (microprocessor)



Intel 4004 (1971)

- 1,000 logic gates
- 92,000 instructions per second
- 2,250 ($\sim 10^4$) transistors

Very Large Scale Integration (VLSI) : 1980s



Motorola 68000

Major architectural step in microprocessors:

- 16/32 bit architecture

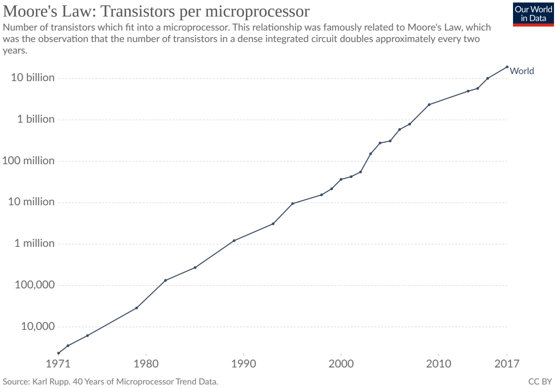
First implementation in 1979:

- 68,000 ($\sim 10^5$) transistors
- $\sim$ 1 MIPS (Millions of Instructions Per Second)

Used in:

- Apple Macintosh
- Sun, Silicon Graphics, Apollo workstations

Moore’s Law


Moore's Law: Transistors per microprocessor

Number of transistors which fit into a microprocessor. This relationship was famously related to Moore's Law, which was the observation that the number of transistors in a dense integrated circuit doubles approximately every two years.

Source: Karl Rupp. 40 Years of Microprocessor Trend Data.

CC BY

Gordon Moore (1929–), Intel co-founded

source: Intel Free Press / CC BY-SA 2.0

Transistor count doubles every two years.

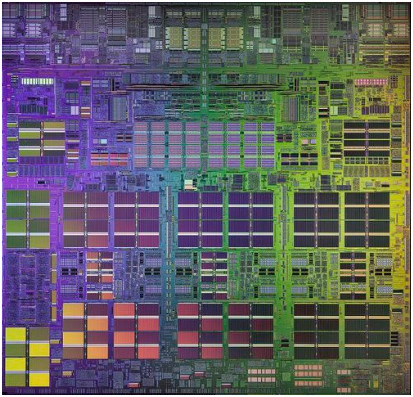
Moore’s “Law” = Observation

Computer Organization = how to *organize* all these transistors

- Make efficient use of them (improving performance, or energy efficiency)
- Do we design a machine that we can reuse for multiple problems?
  ⇒ programmable, general-purpose computers ⇒ software
- Or, do we design a machine for a specific purpose?
  ⇒ application-specific hardware

We will focus on general-purpose computers in this course, but later courses explore a variety of ways computers are optimized.

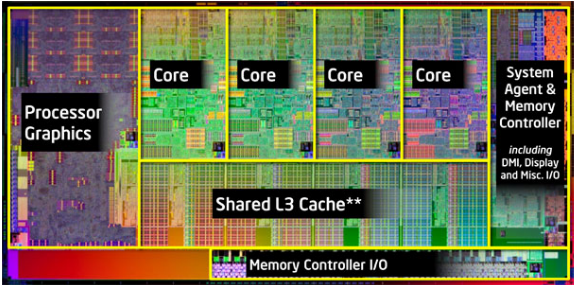
Multicore Processors (2000s)



IBM Power4 (2001)

- First commercial multicore processor
- Two identical copies on the same substrate
- 174M ($\sim 10^8$) transistors

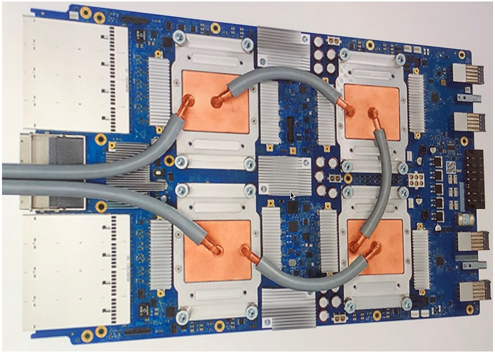
General-purpose Heterogeneous Processors (2010s)



Intel Sandybridge (2011)

- One of the first CPU + integrated GPU chips
- 1B (10^9) transistors

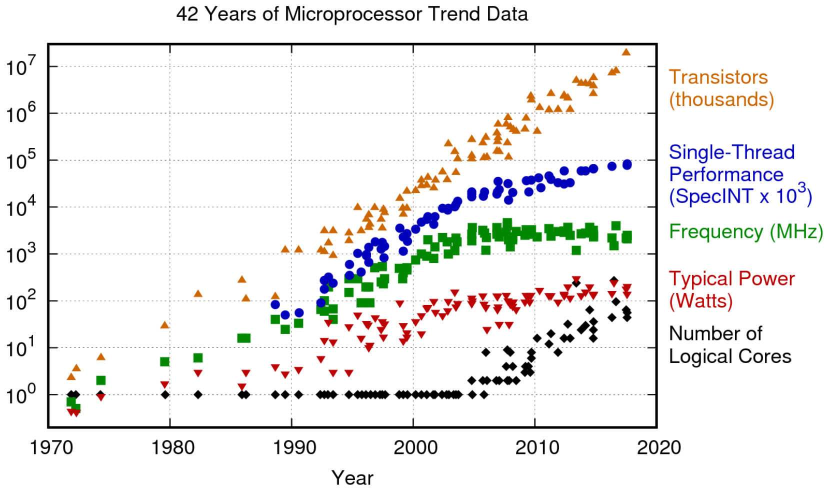
Domain-specific Architectures (2020s)



Google TPU (2016)

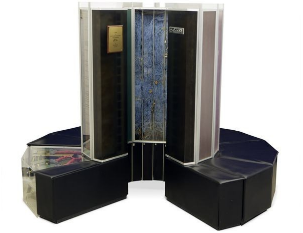
- Accelerate neural networks
- 8-bit matrix multiplication engine

Historical Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

How far have we come?



Cray-1 Supercomputer (1975)

- 80 MHz
- 250 MFLOPS (10^8)
- 115 KW
- $8.86 Million

AMD Radeon RX 5600 (2020)

- 1560 MHz
- 6,390 GFLOPS (10^{12})
- 150 W
- $300

Introduction

Classes of Computers

Textbook§1.1

Notation

Bit vs. Byte

b = bit

B = byte (= 8 bits)

Term	Abbreviation	Approx. value	Actual value
byte	B	10^0	2^0
kilobyte	KB	10^3	2^{10}
megabyte	MB	10^6	2^{20}
gigabyte	GB	10^9	2^{30}
terabyte	TB	10^{12}	2^{40}
petabyte	PB	10^{15}	2^{50}
exabyte	EB	10^{18}	2^{60}
zettabyte	ZB	10^{21}	2^{70}
yottabyte	YB	10^{24}	2^{80}

In practice, the “powers of two values” are used.

Except for storage, *e.g.*, 1TB = 10^{12} bytes $< 2^{40} bytes$.

In this course, we will use “powers of two values” for everything.

Servers

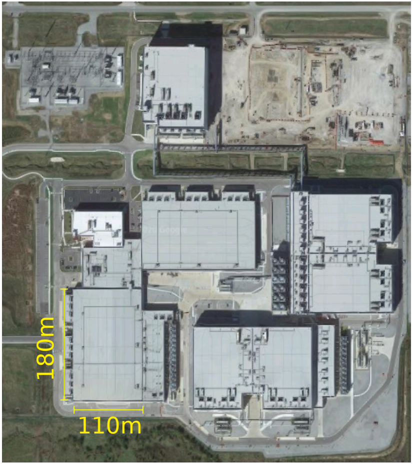
- Computers used for running large programs for multiple users, typically accessed only via a network.
- Price: $5,000 – $2M

Cloud Computing

- 10-100K of servers
- Housed in large data centers

source: By Hugovanmeijeren - Own work, CC BY-SA 3.0

CERN Data Center (2010)



source: Google Maps

Google Data Center, Pryor, Oklahoma

Personal Computers (PC)

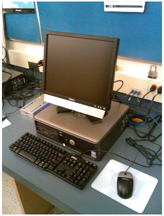
- Price range $300 – $4000
- Runs a large variety of different software applications

PC Example: Desktop

source: By Veradrive - Own work, CC BY 4.0

IBM XT, 1983

CPU: Intel 8088 @ 4.77 MHz
Memory: 640 KB RAM



source: By Jeremy Banks - originally posted to Flickr as New Computers, CC BY 2.0

Dell PC, 2007

CPU: Intel Core 2 Quad @ 3.33 GHz
Memory: 8 GB RAM

Personal Computers (PC)

PC Example: Laptop

source: By https://es.ifixit.com/User/524640/Sam+Lionheart - https://d3nevzfk7ii3be.cloudfront.net/igi/gFgosPJbspyCBD5P, CC BY-SA 3.0

MacBook Air 2008

- CPU: Intel Core 2 Duo with 4 MB on-chip L2 cache @ 1.8 GHz
- Memory: 2 GB of 667 MHz DDR2 SDRAM
- Storage: 64 GB SSD
- Input/Output (I/O) devices: keyboard, touchpad, screen, mic, speakers, USB port, WiFi/Ethernet

PMD: Personal Mobile Device

- Price: $100 – $1000
- All the elements of a computer are there: touchscreen, virtual keyboard, WiFi, processors, memory, storage
- Majority of these devices use ARM processors

  ⇒ energy-efficient processors (2 W)

source: By Carl Berkeley from Riverside California - iPhone First Generation 8GBUploaded by Partyzan_XXI, CC BY-SA 2.0

First iPhone, 2007

source: By matt buchanan - , CC BY 2.0

First iPad, 2010

Embedded Computers

A computer inside another device

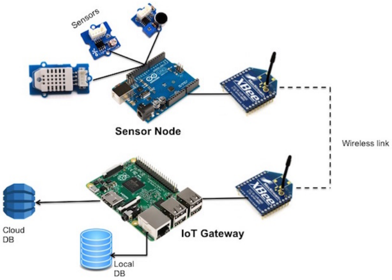
Often referred to as an *embedded system*

- 98% of all processors are in embedded systems
- Users do not necessarily realize they are interacting with a computer
- *E.g.*, there are 25-50 processors in a typical car
    - Engine management
    - Entertainment
    - Safety systems (ABS, traction control, pedestrian detection)
    - Telemetry (*e.g.*, automatic roadside assistance)
    - Input: sensors (*e.g.*, accelerometer) output: mechanical control

source: https://www.lg.com/

Internet of Things (IoT)

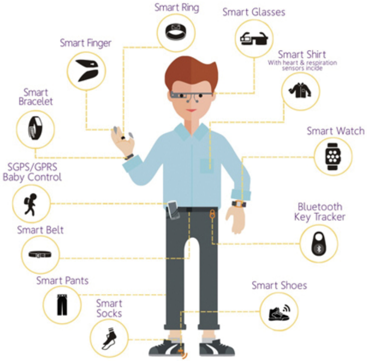
- Embedded systems
- Connected to other devices



source: thenewstack.io

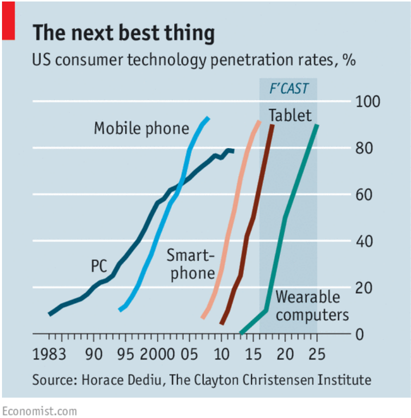
Arduino - Rasberry Pi IoT nodes

E.g., wearable devices



source: www.ept.ca

Consumer Devices Market Penetration



source: https://www.economist.com/business/2016/09/10/still-ringing-bells

Internally, all these *computers* are *organized* using the same concepts:

- Instruction Set Architecture
- Software (assembler, compilers, operating system)
- I/O (Input/Output)
- Memory
- Processor

Introduction

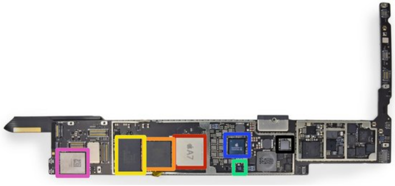
Under the Hood

What’s under the hood?

source: www.ifixit.com

iPad Air LTE Teardown

Main Board ("Mother" Board)



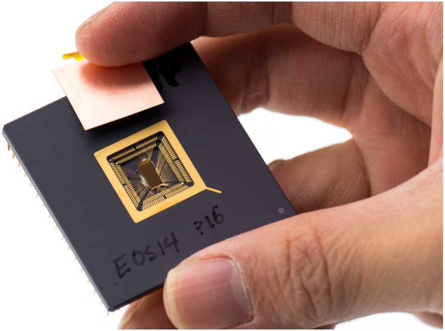
source: www.ifixit.com

iPad Air LTE board

- 🟥 Apple A7 Processor - dual-core ARMv8-A *processor*
- 🟧 Elpida 1 GB LPDDR3 SDRAM *memory*
- 🟨 Toshiba 16 GB NAND Flash *storage*
- 🟩 NXP LPC18A1 (Apple M7 Motion Co-Processor) ARM Cortex-M3 core
- 🟦 Dialog Power Management IC
- 🟥 USI 339S0213 Wi-Fi Module

Processor

This is where all computations happen, where data is *processed*.



source: https://www.flickr.com/photos/dcoetzee/8694597164/ CC0 1.0

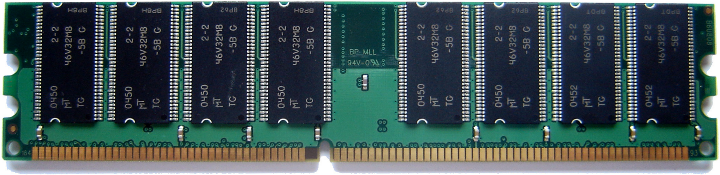
RISC V prototype chip, 2013

Components

- Processor die: single piece of semiconductor (silicon); does the work
- Processor package: plastic/ceramic housing with gold pin contacts; I/O, and heat removal

Memory

This is where the program, and the data it processes, are *temporarily* stored.



source: Utente:Sassospicco / CC BY-SA 2.5

1 GB RAM module

Memory is often *volatile*: when power is off, the data is lost.

Storage

This is where programs and data are stored for the *long-term*.

source: Evan-Amos / CC BY-SA 3.0

500GB WD hard disk drive (HDD)
2.5-inch (6cm × 1cm × 10cm)
CAD $65 (Aug. 2020)

source: https://www.newegg.ca

512GB Samsung solid state drive (SSD)
(22mm × 2mm × 80mm)
CAD $240 (Aug. 2020)

An SSD is also referred to as *non-volatile* memory (NVM).

I/O Devices

I/O is used to *exchange data* with humans or other machines.

source: https://commons.wikimedia.org/wiki/File:Computer_keyboard_US.svg Public domain

Computer keyboard

source: https://en.wikipedia.org/wiki/Computer_mouse#/media/File:Sega-Dreamcast-Mouse-BL.jpg

Computer mouse

source: https://www.dell.ca

Computer screen

Basic Abstractions

Textbook§1.5

What can a computer do?

It can *move* data in and out of variables:

```
country = ”Canada”;
b = a;
```

It can *operate* on data:

```
b = a*12;
course = ”Computer” + ” Organization”;
```

It can *decide what to do next* based on a condition:

```
if (b < 0)
  c = c+1;
else
  c = c-1;
```

Have you ever wondered how the machine executes more complex code such as:

```
  for(int i=0; i<10; i++) {
    printf(”The value of i is: %d”,i);
  }
```

High-level languages such as C or Java provide a convenient *abstraction* that makes programs:

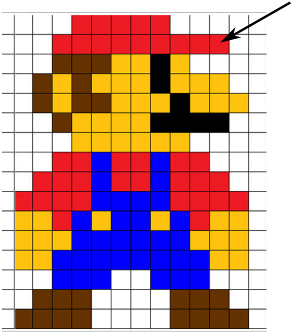
- easy to code (usability)
- easy to understand (readability)
- easy to re-use (reusability)
- easy to re-target to different machines (portability)

How is data represented?

- All data is represented internally as binary numbers in the machine.
  E.g., the number 3 is represented as `0000 0011`.
- Text can be represented by a code that assigns each text character a number.
  E.g., the ASCII code for the character “C” is 67, which is represented as `0100 0011`.

How about images?

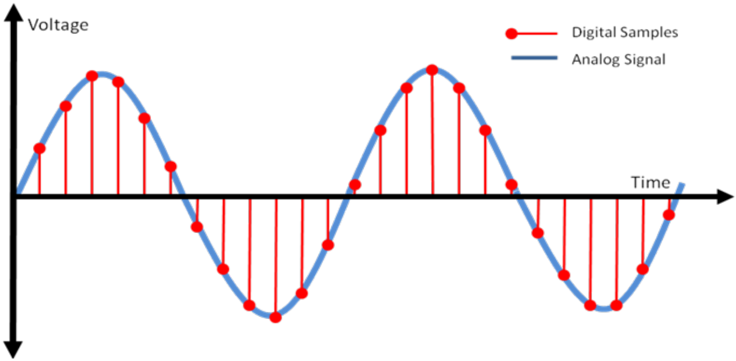
- Images are 2D arrays of picture elements (pixels)
- Each pixel is represented by a set of numbers indicating the intensity of colors such as red, green and blue.



4 bytes RGBA format
(Red,Green,Blue,Alpha)
(255,0,0,255)

Sound

Sound can also be represented by collections of numbers representing the magnitude of audio signals sampled at regular time intervals, or by collections of numbers describing the frequency content of the signal.



Hardware

To process data, a computer uses digital circuits.

In ECSE 222 (Digital Logic), you learned all the basic digital circuit building blocks that you need to make a computer:

- Logical functions
  E.g., AND, OR, NOT, XOR
- Binary arithmetic functions
  E.g., Addition, shifting
- Memory
  E.g., Flip-flops (registers)

Computers are Sequential Digital Machines

Computers process sequences of input data to compute sequences of output data.

Order matters!

This notion of time means that a computer must be a *sequential circuit*. Therefore it must have:

- *A clock* to synchronize different computer component's operations, and
- *Memory* to store past results.

All computers have memory. Not all computers have clocks; most do!

Computer Programs are Sequences of Simple Operations

A program written in a high-level language must be translated to a program that consists only of the simple operations that the computer hardware can actually perform:

Reading from memory, operating on data with logical and arithmetic functions, and writing to memory.

A programming language consisting of these simple operations is called *machine language*.

A machine language program is made up of a list of statements called *instructions*.

Machine and Assembly Language

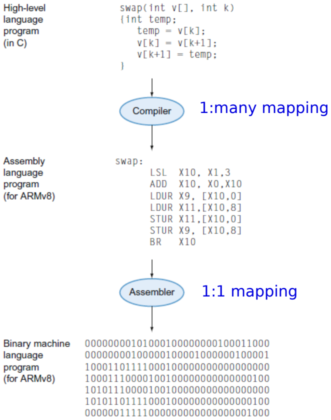
Instructions are simple operations implemented with digital logic.

Even the most complex tasks (self-driving car, Siri, 3D game) are executed by programs consisting of instructions.

Instruction = Data

An instruction is represented by a number, just like data.

This is a fundamental concept: both data and instructions are represented by binary numbers; both are stored in memories.



High-level language program (in C)
Compiler
1:many mapping
Assembly language program (for ARMv8)
Assembler
1:1 mapping
Binary machine language program (for ARMv8)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
      LSL  X10, X1,3
      ADD  X10, X0,X10
      LDUR X9, [X10,0]
      LDUR X11,[X10,8]
      STUR X11,[X10,0]
      STUR X9, [X10,8]
      BR   X10
```

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

Where are the instructions stored?

Two type of architectures:

- *Harvard architecture* (1944)
  Instructions and data are stored in separate memories.
- *Von Neumann architecture* (1945)
  Instructions and data are stored in the same memory.

John von Neumann (1903-1957)

Today, most computers use the von Neumann architecture, at least as far as software is concerned.

Internally (invisible to software), Harvard architecture is almost always used, for performance reasons.

Binary Integer Arithmetic (Recap)

Textbook§1.4, 1.5

Unsigned Integers

Decimal number $D = d_{n-1}d_{n-2}\dots d_1d_0$ where $d_i \in \{0, 1, \dots, 9\}$

Value in base 10 $V(D) = \sum_{i=0}^{N-1} d_i \times 10^i$

$e.g.$, $67 = 6 * 10^1 + 7 * 10^0 = 67$

Binary number $B = b_{n-1}b_{n-2}\dots b_1b_0$ where $b_i \in \{0, 1\}$

Value in base 10 $V(D) = \sum_{i=0}^{N-1} d_i \times 2^i$

$e.g.$, $0100\ 0011 = 1 \times 2^6 + 1 \times 2^1 + 1 \times 2^0 = 67$

$E.g.$, $67_{10} = 0100\ 0011_2$
$E.g.$, $13_{10} = 0000\ 1101_2$

The range of values depends on number of bits n: $V(D) \in [0; 2^n - 1]$.
$E.g.$, if $n = 8$ bits, the maximum value is $2^8 - 1 = 255_{10} = 1111\ 1111_2$.

Binary Addition

Decimal addition

$$\begin{array}{r} 67 \\ + \quad 13 \\ \hline {}^{1} \\ 80 \end{array}$$

Binary addition

$$\begin{array}{r} 0100\ 0011 \\ + \quad 0000\ 1101 \\ \hline {}^{1\ \ 1\,1\,1} \\ 0101\ 0000 \end{array}$$

Watch out for overflow!

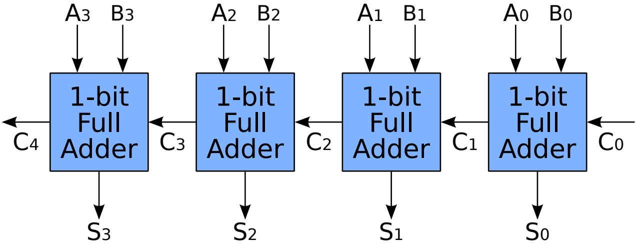
$$\begin{array}{r} 195 \\ + \quad 141 \\ \hline {}^{1} \\ 336 \end{array}$$

$$\begin{array}{r} 1100\ 0011 \\ + \quad 1000\ 1101 \\ \hline {}^{1\qquad 1\ \ 1\,1\,1} \\ 0101\ 0000 \end{array}$$

336 is larger than the maximum value (255) we can represent with 8 bits. The carry-out indicates the overflow.

Binary Addition in Hardware

Ripple carry adder: $S = A + B$



source: https://commons.wikimedia.org/wiki/File:4-bit_ripple_carry_adder.svg en:User:Cburnett / CC BY-SA

Signed Integers: Sign-and-magnitude Representation

We need to encode the sign in the representation of the binary number.

Sign-and-magnitude is the simplest approach: use the leftmost bit (MSB) to represent the sign, and the remaining bits to represent the magnitude (*i.e.*, absolute value). Example for 8 bits:

$MSB = 0 \Rightarrow positive$ $\qquad +13 = 0000\ 1101$

$MSB = 1 \Rightarrow negative$ $\qquad -13 = 1000\ 1101$

Problems with sign-and-magnitude:

- Two representations for zero = 0000 0000 = 1000 0000
- We need extra hardware to handle the addition of a positive number and a negative one (we cannot simply add the numbers together)

Signed Integers: 1’s-complement Representation

To get a negative value: invert each bit of the corresponding positive representation, and vice-versa.

This representation has the advantage that signed and unsigned arithmetic can use the same hardware.

$$\begin{aligned} +13 &= 00001101 \\ -13 &= 11110010 \end{aligned}$$

$$\begin{array}{rrl} & 0001\,0000 & = (16_{10}) \\ + & 1111\,0010 & = (-13_{10}) \\ \hline & {}^{1\,1\,1\,1}\,0000\,0010 & = (2_{10}) \end{array}$$

This result is off by one; carry out, but no overflow.

Overflow

Overflow occurs when the result of an arithmetic operation does not fit into the range of the n-bit representation, *e.g.*, $[-2^{n-1}, 2^{n-1} - 1]$ when a bit is used to represent the sign.

If there is a carry out during *unsigned* arithmetic, overflow has occurred.

$$\begin{array}{rrl} & 0001\,0000 & = (16_{10}) \\ + & 1111\,0010 & = (242_{10}) \\ \hline & {\scriptstyle 1\,1\,1\,1} & \\ & 0000\,0010 & = (2_{10}) \end{array}$$

Here, the result is off by 256; the carry out (2^8) indicates overflow.

In *signed* arithmetic, overflow must be detected differently.

Back to 1's-complement Representation

$$\begin{aligned} +13 &= 00001101 \\ -13 &= 11110010 \end{aligned}$$

$$\begin{array}{rrl} & 0001\,0000 & = (16_{10}) \\ + & 1111\,0010 & = (-13_{10}) \\ \hline & {\scriptstyle 1\,1\,1\,1} & \\ & 0000\,0010 & = (2_{10}) \end{array}$$

This result is off by one;
carry out, but no overflow.

Problems:

- Still two representations for zero = 0000 0000 = 1111 1111
- Need to add 1 to the result when an operand is negative (try as an exercise with $(-2)_{10} + (-2)_{10}$)
- Need a way to identify overflow

Signed Integers: 2's-complement Representation

To get a negative value: invert each bit of the corresponding positive representation and add one (works in reverse as well).

$$\begin{aligned} +13 &= 0000\ 1101 \\ -13 &= 1111\ 0010 + 1 \\ &= 1111\ 0011 \end{aligned}$$

$$\begin{array}{rrl} & 0001\ 0000 & = (16_{10}) \\ + & 1111\ 0011 & = (-13_{10}) \\ \hline & {\scriptstyle 1\,1\,1\,1} \\ & 0000\ 0011 & = (3_{10}) \end{array}$$

Correct value; however, carry out without actual overflow again!

Problem:

- Still need a way to identify overflow

Ranges

Integer representations, assuming $n = 4$ bits:

Binary	Decimal Value		
	Sign and Magnitude	1's Complement	2's Complement
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	-3	-4
1101	-5	-2	-3
1110	-6	-1	-2
1111	-7	-0	-1
0000	+0	+0	+0
0001	+1	+1	+1
0010	+2	+2	+2
0011	+3	+3	+3
0100	+4	+4	+4
0101	+5	+5	+5
0110	+6	+6	+6
0111	+7	+7	+7
Range:	$[-7; +7]$	$[-7; +7]$	$[-8; +7]$
	$[-2^{n-1} - 1; 2^{n-1} - 1]$	$[-2^{n-1} - 1; 2^{n-1} - 1]$	$[-2^{n-1}; 2^{n-1} - 1]$

Overflow in 2's Complement Addition

Recall that overflow occurs when the answer does not fit into the representable range of numbers.

Observations:

- With signed addition, the carry-out does not indicate overflow.
- Overflow can only happen if both numbers have the same sign.

Rule: Overflow only occurs if both summands have the same sign, and the sum has a different sign than that of the summands.

$$\begin{array}{rrl} & 0110 & = (+6_{10}) \\ + & 0100 & = (+4_{10}) \\ \hline & 1010 & = (+10_{10}) \end{array}$$

No carry out, different sign

$\Rightarrow$ overflow!

$$\begin{array}{rrl} & 1110 & = (-2_{10}) \\ + & 1001 & = (-7_{10}) \\ \hline & {}^{1}0111 & = (9_{10}) \end{array}$$

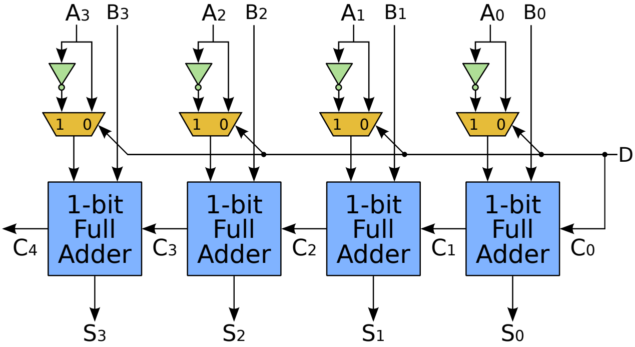
Carry out, different sign

$\Rightarrow$ overflow!

Subtraction

$B - A = B + (-A)$: form the 2's complement inverse of A and add to B.

In hardware, invert the bits and add one using the carry in signal C_0. The signal D selects between addition and subtraction.



source: https://commons.wikimedia.org/wiki/File:4-bit_ripple_carry_adder-subtracter.svg en:User:Cburnett / CC BY-SA

Sign Extension

Sometimes you will want to convert an n-bit number to an m-bit number, where $m > n$.

The rule for 2's complement numbers is to replicate the sign bit.

4-bit value	8-bit value	
0010	0000 0010	$= (2_{10})$
1110	1111 1110	$= (-2_{10})$

Sign-extension is important if (when) we store numbers in memory using fewer bits than our processor uses for its operations.

Hexadecimal Representation

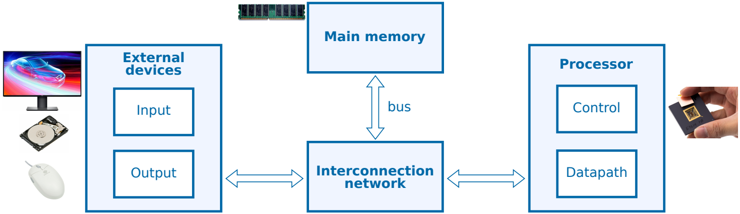
- Binary can be very unwieldy when representing large values: $7748_{10} = 0001111001000100_2$
- We can use the base-16 hexadecimal (hex) representation. Each hex digit has $16 = 2^4$ possible values and represents 4 bits.
- We can write the above binary number more compactly in base-16 as $1E44_h$.
- Get good at converting back and forth between bin, hex, and dec!

Decimal	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Basic Computer Organization

Textbook§1.2, 1.3, 2.1, 2.2

Computer System Components



Processor

- CPU (Central Processing Unit)
- This course: only single core

External devices

- IO (Input/Output)
- *E.g.*, keyboard, HDD, display

Main memory

- RAM (Random Access Memory)
- Stores program and data

Interconnection network

- Communication medium
- Uses shared buses

Basic Computer Organization

Textbook§1.2, 1.3, 2.1, 2.2

Memory

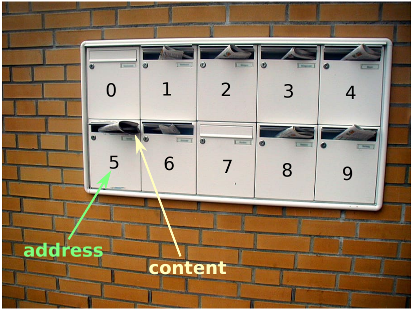
Main memory

Usually, data in a program resides in main memory. Conceptually (*i.e.*,, in software's view), each variable is allocated in main memory.

```
int   z = 42;  // 32 bits = 4 bytes (ARMv7-A)
short s = 11;  // 16 bits = 2 bytes (ARMv7-A)
char  c = 30;  // 8 bits = 1 byte in C (16 bits in Java)

int arr[10];   // 10*4 bytes = 40 bytes (ARMv7-A)
```

Memory Addresses and Content



- Computer memory is organized as a linear array of bytes.
- Each byte in the memory has its own unique address (“byte-addressable).
- The processor can read or write the content (byte value) of the memory at a given address.

Address Space

Addresses are represented using k bits = address size.

There are 2^k addressable locations in the address space of the computer, numbered from 0 to $2^k - 1$.

E.g., a 32-bit address space has $2^{32} = 2^{30} \cdot 2^2 = 4G$ addresses.

Since each address corresponds to a location that stores a byte, the capacity of the memory is 4 GB (GigaBytes) for a 32-bit address space.

How many for a 24-bit address space?

Accessing Multiple Bytes

Most computers process data in chunks of several bytes: a *word*.

- A typical *word size* or *word length* is 32 bits (4 bytes).
- The word size and address size of a computer are *often* equal.

The memory has a mechanism to read or write multiple consecutive bytes with a single request instead of having to access bytes individually multiple times.

Data can be accessed (read or write) in chunks of multiple bytes by giving the address of the starting byte and the size of the chunk, usually one *byte*, 2 bytes (*half word*), or 4 bytes (*word*).

Example: Accessing Multiple Bytes

Read the word stored starting at address 0x4.

address	value
0x00	0xD1
0x01	0x4B
0x02	0x45
0x03	0xC4
0x04	0x90
0x05	0x12
0x06	0x4F
0x07	0xEE
0x08	0x78
0x09	0x91
0x0A	0x03
0x0B	0x70
0x0C	0xB3
0x0D	0xDA
0x0E	0x7F
0x0F	0xE6

address	value
0x00	0xD1
0x01	0x4B
0x02	0x45
0x03	0xC4
0x04	**0x90**
0x05	**0x12**
0x06	**0x4F**
0x07	**0xEE**
0x08	0x78
0x09	0x91
0x0A	0x03
0x0B	0x70
0x0C	0xB3
0x0D	0xDA
0x0E	0x7F
0x0F	0xE6

Read(0x04) = 0x 90 12 4F EE? or 0x EE 4F 12 90?

It depends on byte ordering.

Byte Ordering (Endianness)

address	0x04	0x05	0x06	0x07
value	**0x90**	**0x12**	**0x4F**	**0xEE**

Big Endian: starts with the Most Significant Byte (Big byte).
E.g., Read(0x04) = 0x 90 12 4F EE

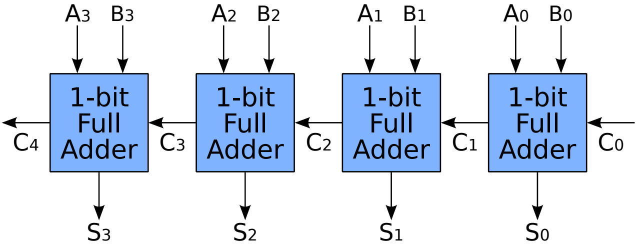
Little Endian : starts with the Least Significant Byte (Little byte).
E.g., Read(0x04) = 0x EE 4F 12 90

Origin of the name

Gulliver's Travels, 1726
(Jonathan Swift).

Why Little-Endian?

One of the reasons: consider the ripple carry adder:



The carry chain starts with the least significant bit. When the first byte-addressable micro-controllers and processors appeared, little-endian ordering makes addition more efficient: increment addresses by 1 and feed the values into the adder from the least significant byte to the most significant one.

Alignment

Some computers require memory accesses to start on an address that is a multiple of the chunk size in bytes:

- 32-bit words can only be accessed at address 0,4,8,...
- 16-bit half-words can only be accessed at address 0, 2, 4, 6, ...
- Bytes can be accessed at any address 0, 1, 2, 3, ...

Multiple reasons for this, mostly due to the way the processor and memory sub-system are implemented.

An access at address *addr* to data of size *sze* is aligned if and only if:

$$addr \bmod sze = 0$$

ARMv7-A supports unaligned accesses, but such accesses may be slower than aligned accesses.

Alignment

32-bit word (4-byte) alignment

address	value
0x00	0xD1
0x01	0x4B
0x02	0x45
0x03	0xC4
0x04	0x90
0x05	0x12
0x06	0x4F
0x07	0xEE
0x08	0x78
0x09	0x91
0x0A	0x03
0x0B	0x70
0x0C	0xB3
0x0D	0xDA
0x0E	0x7F
0x0F	0xE6

16-bit half-word (2-byte) alignment

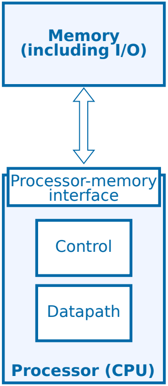
address	value
0x00	0xD1
0x01	0x4B
0x02	0x45
0x03	0xC4
0x04	0x90
0x05	0x12
0x06	0x4F
0x07	0xEE
0x08	0x78
0x09	0x91
0x0A	0x03
0x0B	0x70
0x0C	0xB3
0x0D	0xDA
0x0E	0x7F
0x0F	0xE6

Basic Computer Organization

Textbook§1.2, 1.3, 2.1, 2.2

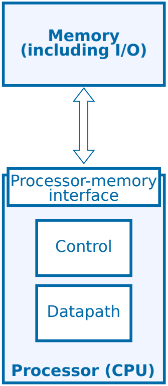
Processor

Processor Overview



- From the processor point's of view, everything outside is memory (including I/O).
- The processor interacts with the outside world through a *memory interface*.

Processor Overview



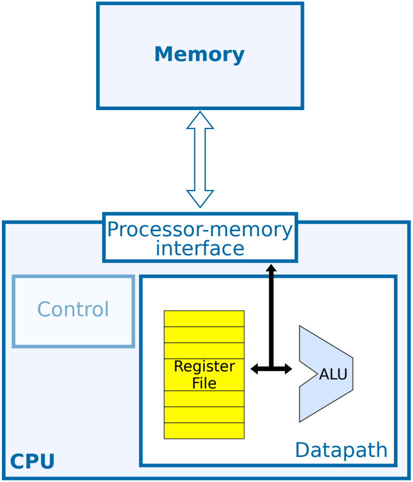
CPU’s Control Logic

- The control logic coordinates the execution of the instructions in the datapath
- Mainly sequential logic consisting of finite state machines

CPU’s Datapath Logic

- The datapath is driven by the control logic and processes data accordingly
- Mainly combinational logic (with the exception of temporary storage) implementing instructions

Processor Datapath



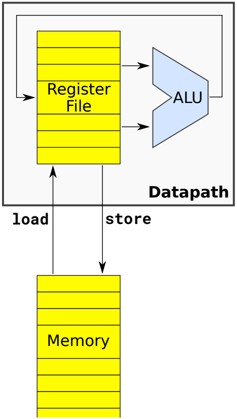
Arithmetic and Logic Unit (ALU)

- Performs operations on data
- *E.g.*, add, multiply, shift, and
- Also used to generate addresses for memory accesses

Register File

- A small number of general-purpose registers used as fast temporary data storage

Load/Store Architecture



Load/Store architecture

Load and store are the only instructions that are allowed to access memory.

- The ALU *only** reads its input data and writes its result from/to registers
  ⇒ simplifies the design of the hardware.
- Use special *load* and *store* instructions for transfers between registers and memory.

** The ALU assists with memory accesses by calculating addresses.*

CISC vs RISC architecture

CISC = Complex Instruction Set Computer (*e.g.*, x86)

- Instructions can be complex (*e.g.*, reciprocal of square root)
- Instructions can access both memory and registers (*e.g.*, add)
- Leads to more complex CPU design
- From the days before good compilers were available

RISC = Reduced Instruction Set Computer (*e.g.*, ARM)

- Simpler CPU design simplifies performance improvement
- Load/store architecture
- Simple arithmetic operations
- Focus of this course

RISC Operations

Textbook§2.3

Register Transfer Notation (RTN)

RTN allows us to specify the semantics of an instruction. Some common notation:

- `Rn` : content of register *n*
- `XX` : content of a specific named register, *e.g.*, *IR* or *PC*
- `Mem[a]` : content of memory at address *a*
- $\leftarrow$: transfer (copy).

Note: the number of bits on both sides of the arrow should be equal!

Instruction set architecture documentation, which describes the operations (*i.e.*, software interface) of a computer, specify such operations in RTN.

Memory Instructions

Memory Load

```
Load R2, ADDRESS
```

Load reads (copies) four consecutive bytes (a word) from the memory starting at memory address ADDRESS and writes them as a word into register R2.

RTN: `R2` ← `Mem[ADDRESS]`

Memory Store

```
Store R4, ADDRESS
```

Store copies the word stored in register R4 into four consecutive bytes in memory starting from address ADDRESS.

RTN: `Mem[ADDRESS]` ← `R4`

ALU Instructions

E.g., Addition

```
Add R4, R2, R3
```

Add adds the contents of registers R2 and R3 and places their sum into register R4. The operands in R2 and R3 are not altered but the previous value in R4 is overwritten.

RTN: $R4 \leftarrow R2 + R3$

Sequence of Instructions

Consider a C program where a, b, c each occupy 4 bytes in memory:

```c
int a;
int b;
int c;
c = a+b;
```

The equivalent sequence of assembly instructions is:

```
Load  R2, A
Load  R3, B
Add   R4, R2, R3
Store R4, C
```

where A, B, and C are the addresses where the variables are stored.

Running a program

When running the program, the machine executes each instruction sequentially, one after another (or at least appears to do so).

Program instructions and data are both stored in memory

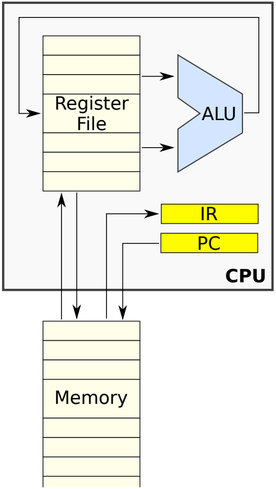
- Instructions are just like data, and are stored in memory
- On a typical RISC machine, all instructions are the same length (*e.g.*, 4 bytes)
- Instructions are stored in consecutive memory addresses: *e.g.*, 0x00, 0x04, 0x08, 0x0C

Address	Content
0x00	Load R2, A
0x04	Load R3, B
0x08	Add R4, R2, R3
0x0C	Store R4, C
	...
(A) 0x30	a’s data
	...
(B) 0x3C	b’s data
	...
(C) 0x48	c’s data
	...

Exercise

Given the following initial values: a=1, b=3 and c=7,
what are the memory and register contents after each instruction?

Program Counter and Instruction Register



Program Counter (PC)

Holds the *address* of the current instruction

Instruction Register (IR)

Holds the *current instruction* to execute

- First, the processor loads (*fetches*) the instruction pointed to by the PC from memory and stores it in the IR.
- The instruction in the IR is then *decoded*, and the processor *executes* it.
- Finally, the PC is updated; usually PC is incremented by the instruction size.

Executing Instructions, Step by Step

	…
0x00	0xe5902030 (Load R2, A)
0x04	0xe590303c (Load R3, B)
0x08	0xe0824003 (Add R4, R2, R3)
0x0C	0xe5804048 (Store R4, C)
	…
0x30	0x00000001 (a's data)
	…
0x3C	0x00000003 (b's data)
	…
	0x00000007 (c's data)
0x48	0x00000004 (c's data)
	…

Registers

PC	0x000000000x000000040x000000040x000000080x000000080x000000
IR	?0xe59020300xe59020300xe590303c0xe590303c0xe08240030xe0824
R2	?0x000000010x00000001
R3	?0x000000030x00000003
R4	?0x000000040x00000004

1. PC ⇐ 0x00000000
2. fetch: IR ⇐ MEM[PC] = 0xe5902030
3. decode: **Load R2, A**
4. execute: R2 ⇐ MEM[A] = 0x00000001
5. PC ⇐ PC+4 = 0x00000004
6. fetch: IR ⇐ MEM[PC] = 0xe590303c
7. decode: **Load R3, B**
8. execute: R3 ⇐ MEM[B] = 0x00000003
9. PC ⇐ PC+4 = 0x00000008
10. fetch: IR ⇐ MEM[PC] = 0xe0824003
11. decode: **Add R4, R2, R3**
12. execute: R4 ⇐ R2+R3 = 0x00000004
13. PC ⇐ PC+4 = 0x0000000C
14. fetch: IR ⇐ MEM[PC] = 0xe5804048
15. decode: **Store R4, C**
16. execute: MEM[C] ⇐ R4 = 0x00000004

Branching

What if we want to change what code we execute based on program conditions?

```
if (a>0)
  b = 7;
else
  b = 13;
```

The machine usually increments the **PC** by 4 after each instruction. We need a generic mechanism to change the **PC**.

Branch Instructions

- *Conditional* branch:
  Changes the PC to a specific address *if a condition* is true.
- *Unconditional* branch:
  Always changes the PC to a specific address.

C code

```
if (a>0)
  b = 7;
else
  b = 13;
```

Assembly code

```
Address   Instruction      Comment
0x00:     Load    R1, A    // load a from MEM[A] into R1
0x04:     BLE     R1, 0x10 // if R1<=0, branch to address 0x10
0x08:     Move    R2, #7   // R2 = 7
0x0c:     Br      0x14     // branch to address 0x14
0x10:     Move    R2, #13  // R2 = 13
0x14:     Store   R2, B    // store R2 into MEM[B] (location of b)
```

- `Br` = Branch
- `BLE` = Branch if Lesser or Equal (to zero)
  (Other variants: `BEQ`, `BNE`, `BLT`, `BGT`, `BLE`, `BGE`)

Assembly Labels

Using explicit addresses is cumbersome when writing assembly.

- Addresses of instructions are likely to change when editing a program!

Solution: use assembly *labels* which associate an address with a name.

```
Label     Instruction        Comment
          Load    R1, A      // load a from MEM[A] into R1
          BLE     R1, 0x10   // if R1<=0, branch to address 0x10
          Move    R2, #7     // R2 = 7
          Br      0x14       // branch to address 0x14
ELSE:     Move    R2, #13    // R2 = 13
END:      Store   R2, B      // store R2 into MEM[B] (location of b)
```

Assuming the first instruction is at address `0x00`:

- `ELSE = 0x10`
- `END = 0x14`

Loops

How does the machine execute loops?

```
int sum = 0;
int i   = 0;
while (i-10 < 0) {
  sum = sum+i;
  i = i+1;
}
```

Again, use branch instructions:

```
         Move   R2, #0      // sum=0
         Move   R1, #0      // i=0
LOOP:    Sub    R3, R1, #10 // R3=i-10
         Bge    R3, END     // if (i-10>=0) branch to END
         Add    R2, R2, R1  // sum = sum+i
         Add    R1, R1, #1  // i = i+1
         Br     LOOP        // branch back to LOOP
END:     Store  R1, I       // store i in memory
         Store  R2, Sum     // store sum in memory
```

More Loops and Conditions

- What about other loop constructs?
- What about other conditions?

```
int sum = 0;
for (int i=0; i<10; i++)
  sum = sum+i;
```

Convert to known constructs/comparisons:

- turn loops into equivalent while loops
- turn conditions into comparison with zero

Equivalent while loop with comparison with zero:

```
int sum = 0;
int i   = 0;
while (i-10 < 0) {
  sum = sum+i;
  i = i+1;
}
```

Conclusion

This set of lectures has:

- Introduced computers and their history
- Looked at basic abstractions (*e.g.*, data, instructions)
- Reviewed integer arithmetic
- Looked at the memory and processor abstractions
- Introduced basic RISC operations

The next set will:

- Present a real processor instruction set (ARMv7)
- Show how to write real assembly programs in more detail