ECSE324 : Computer Organization

Computer Technology and Abstractions

Christophe Dubach
Fall 2020

Original slides from Prof. Warren Gross – 2017.
Updated by Christophe Dubach – 2020.

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: 2020/09/14 10:46:19

Disclaimer

Lectures are recorded live and will be 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 book, the course webpage or ask on Piazza for clarifications.

Introduction

Introduction

A brief history of computer technology

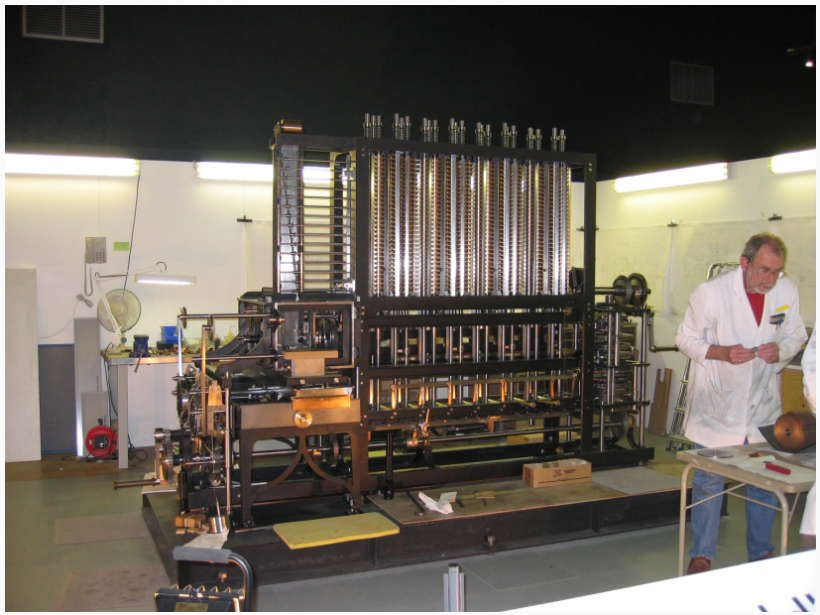
Textbook§1.6,1.7

Mechanical Computers

1822: Difference Engine (Charles Babbage)

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

- Number of parts: 25,000
- Cost: £17,470
  $\cong$ \$ 3mio CAD today
- Working version in 1991 (London Science museum)

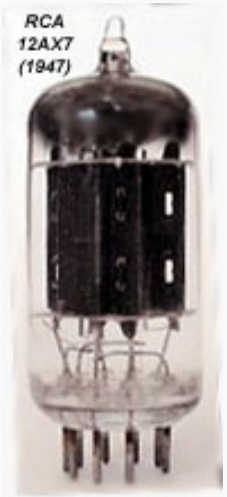


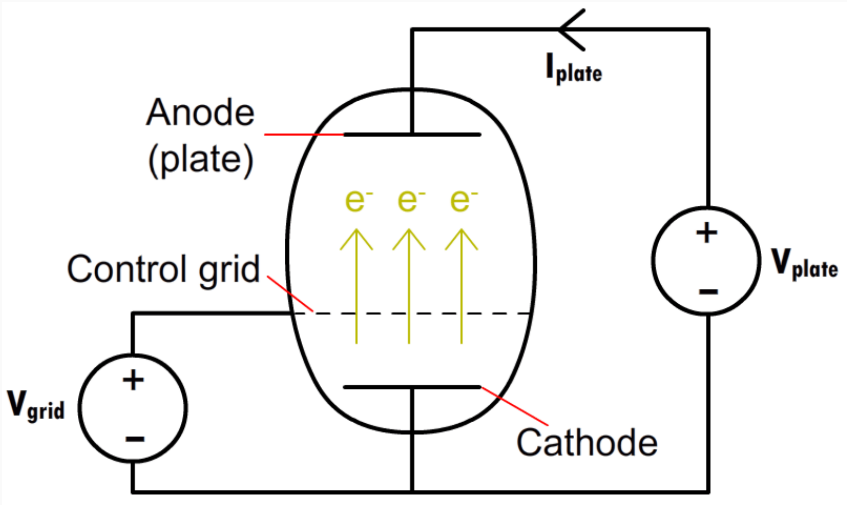
https://youtu.be/KBuJqUf04-w?t=203

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

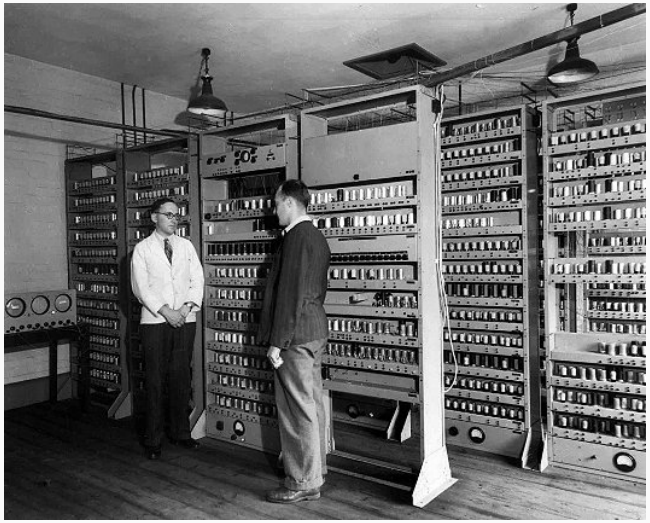
First generation electrical computers: 1940's

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.

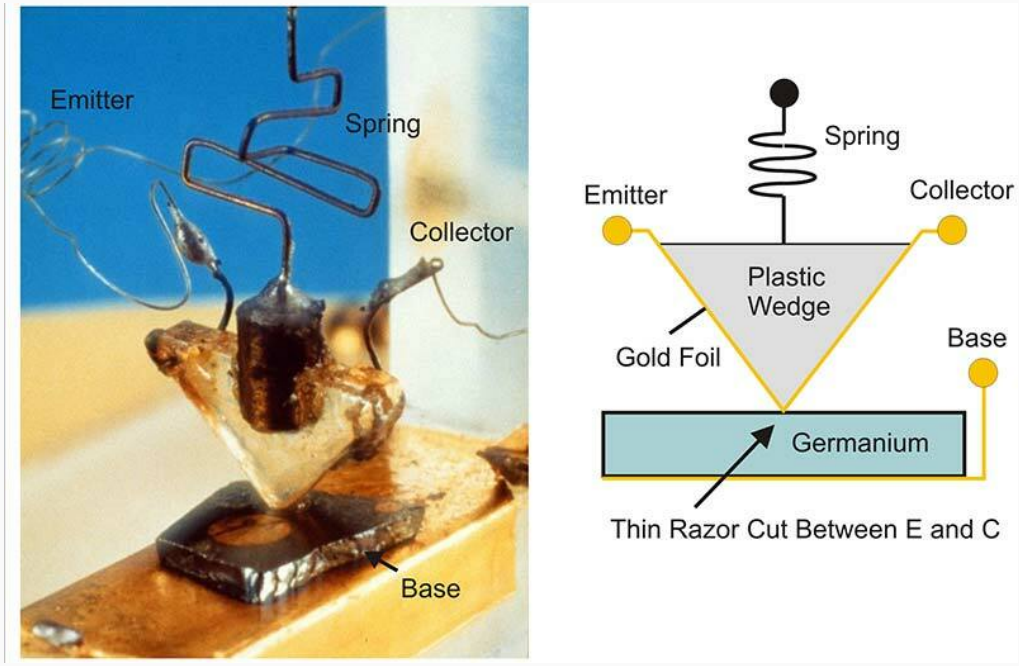
Programs and data stored in same memory !



EDSAC, 1949,
University of Cambridge, UK

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

Transistors: 1948



First transistor, Bell Labs, 1948

source: www.nutsvolts.com

Discrete transistors

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

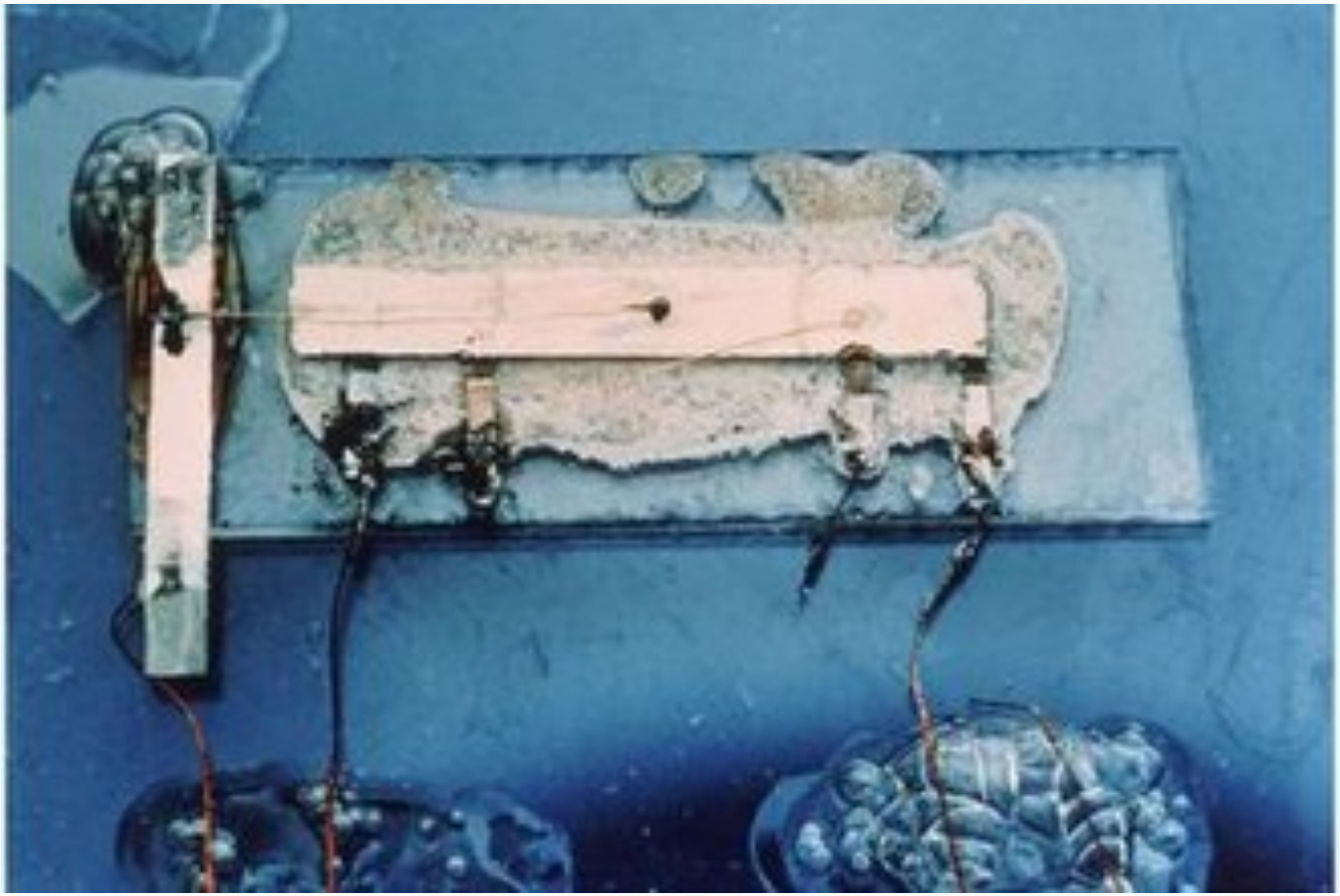
Second Generation (1950s), transistor-based computers

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

First compilers developed: translate high-level program 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

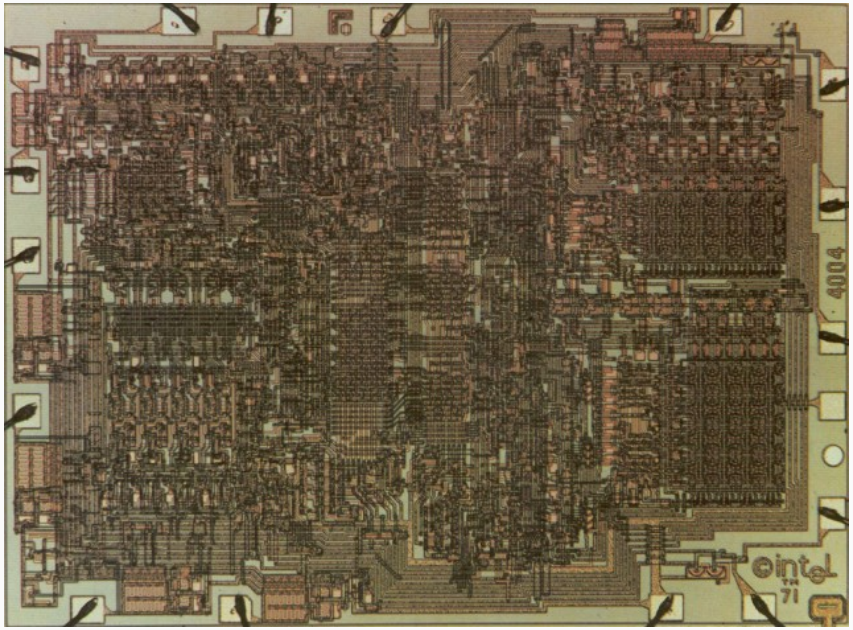
Operating systems allowed several programs to run on a single machine

DEC PDP-8 (1965)

Fourth Generation (1970s): large scale integration

Large Scale Integration

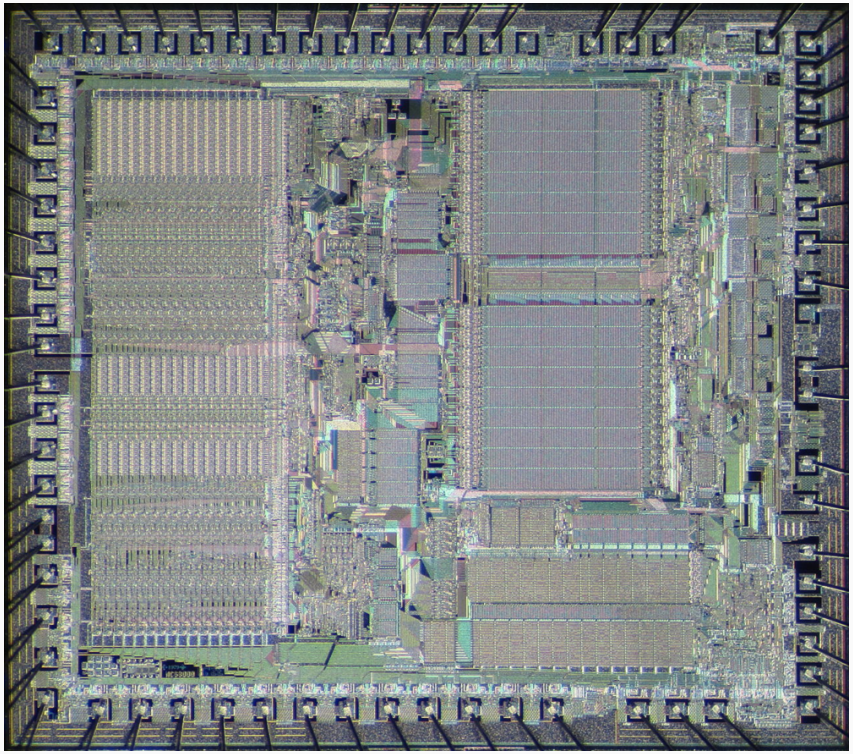
Whole CPU on a single chip (microprocessor)



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

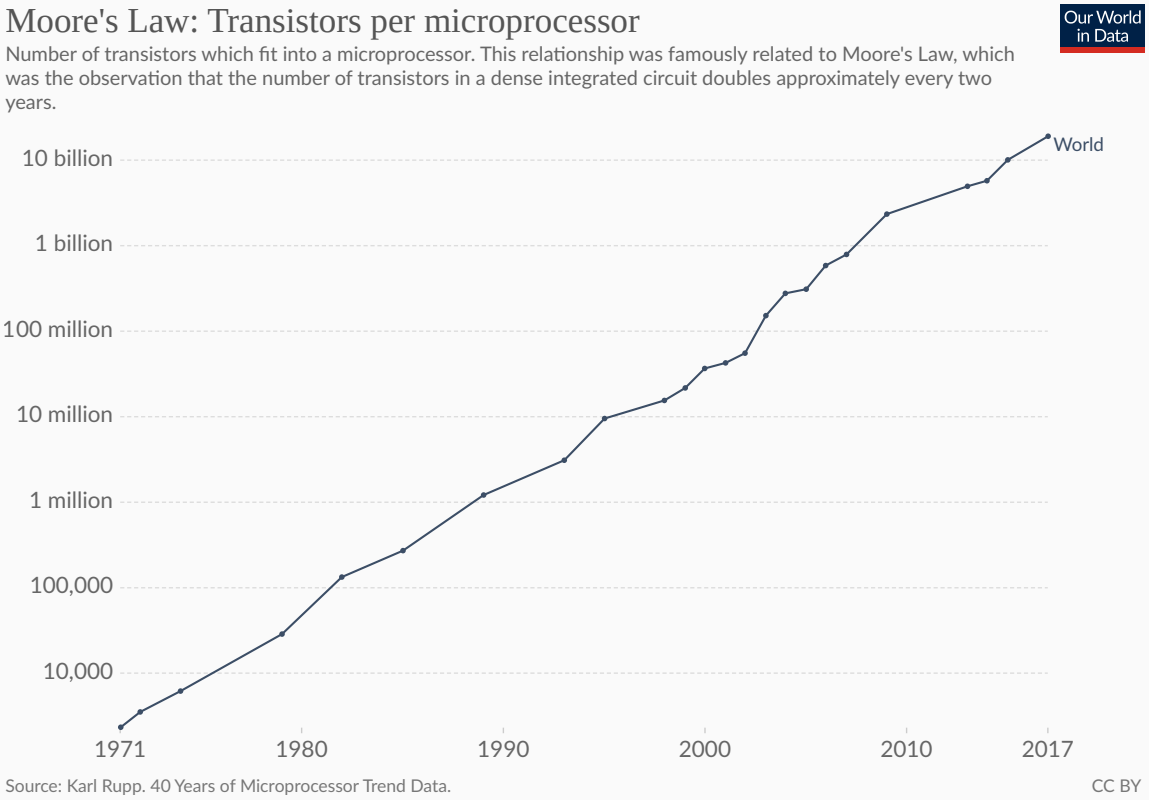
First implementation in 1979:

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

Used in:

- Apple Macintosh
- Sun, Silicon Graphics, Apollo workstations

Moore’s Law



Gordon Moore (1929–), Intel co-founded

source: Intel Free Press / CC BY-SA 2.0

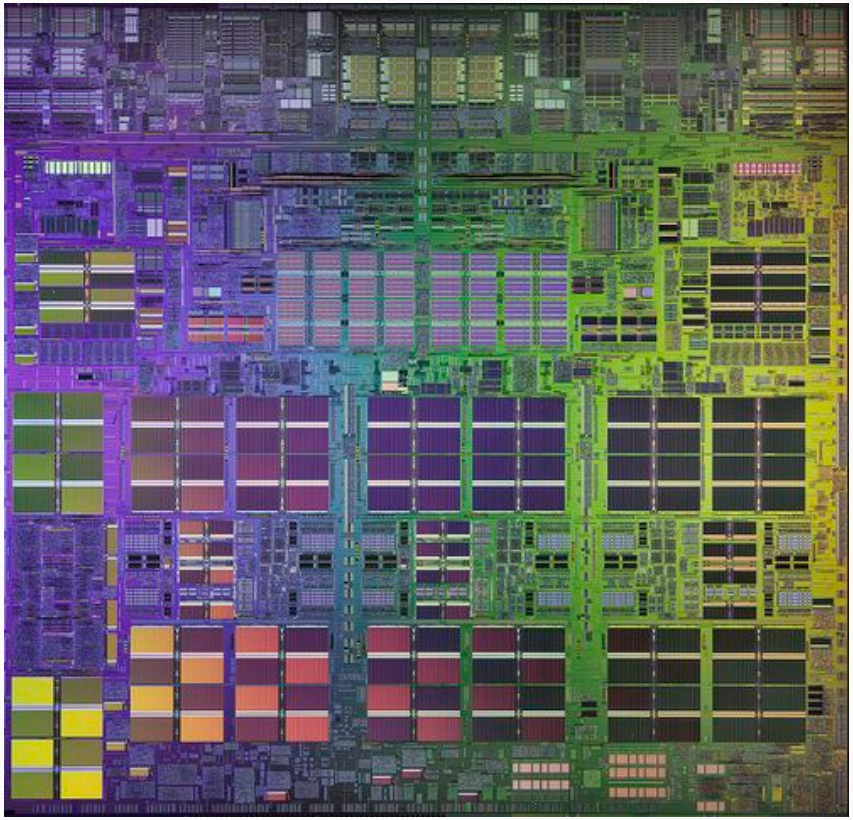
Transistor count doubles every two years.

Moore’s Law = Observation

Computer Organization = how to organize all these transistors

- Make efficient use of them
- Design machine that we can reuse for multiple problems
  $\Rightarrow$ programmable computers $\Rightarrow$ software

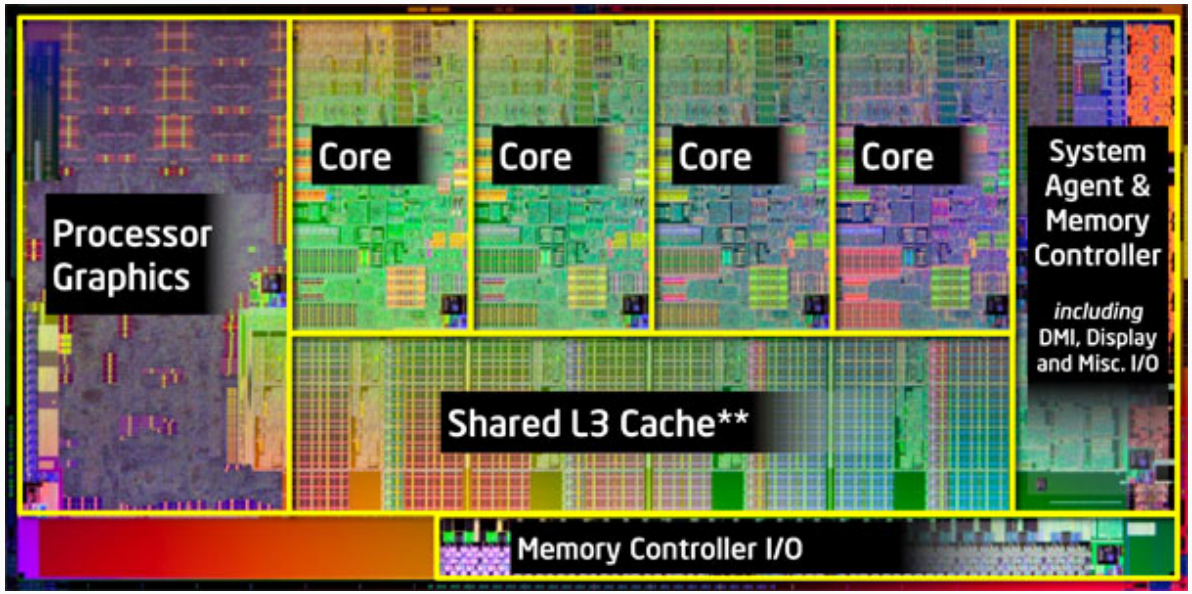
Multicore processors (2000s)



IBM Power4 (2001)

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

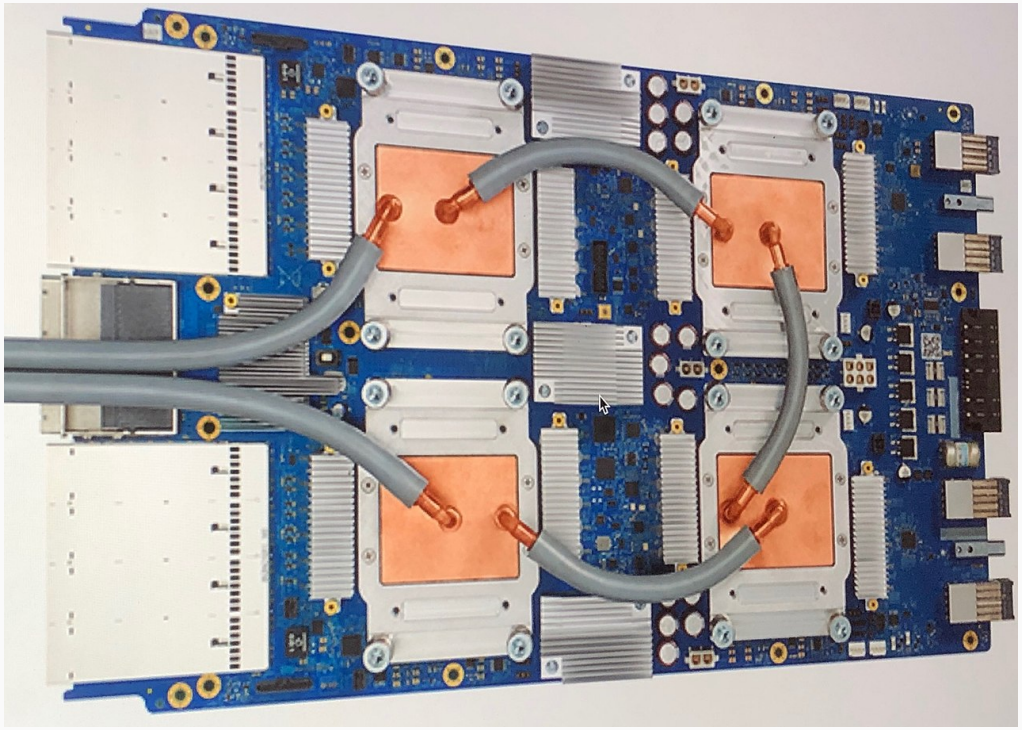
General purpose heterogeneous processors (2010s)



Intel Sandybridge (2011)

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

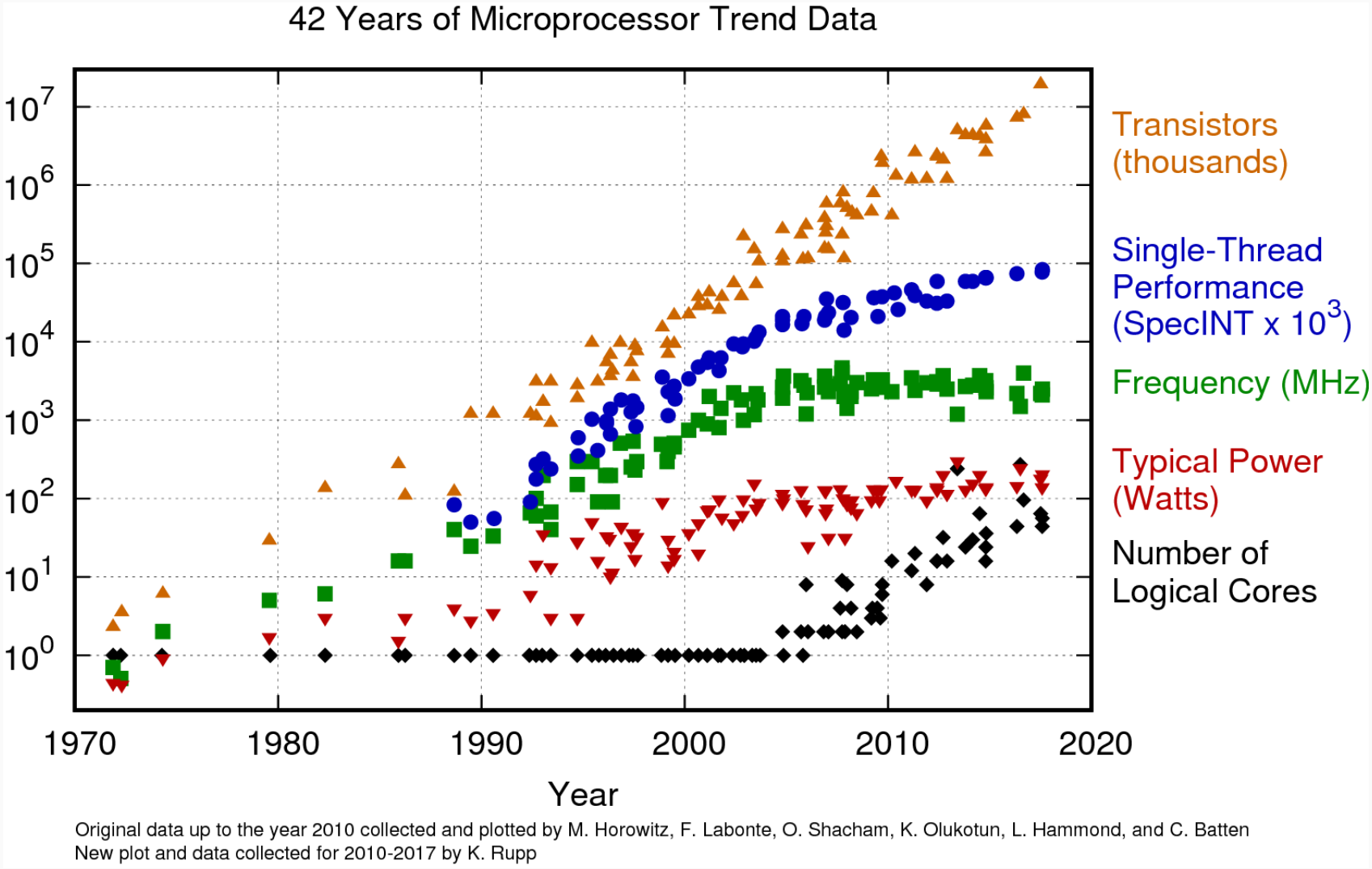
Domain Specific Architectures (2020s)



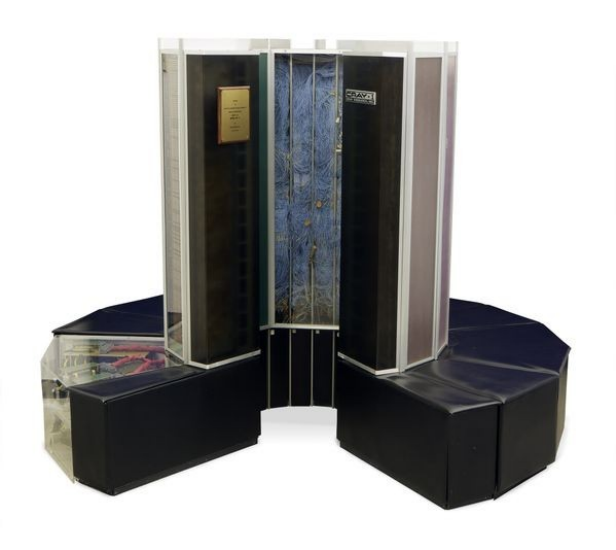
Google TPU (2016)

- Accelerate neural networks
- 8-bit matrix multiplication engine

Historical data



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})
- 150W
- \$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

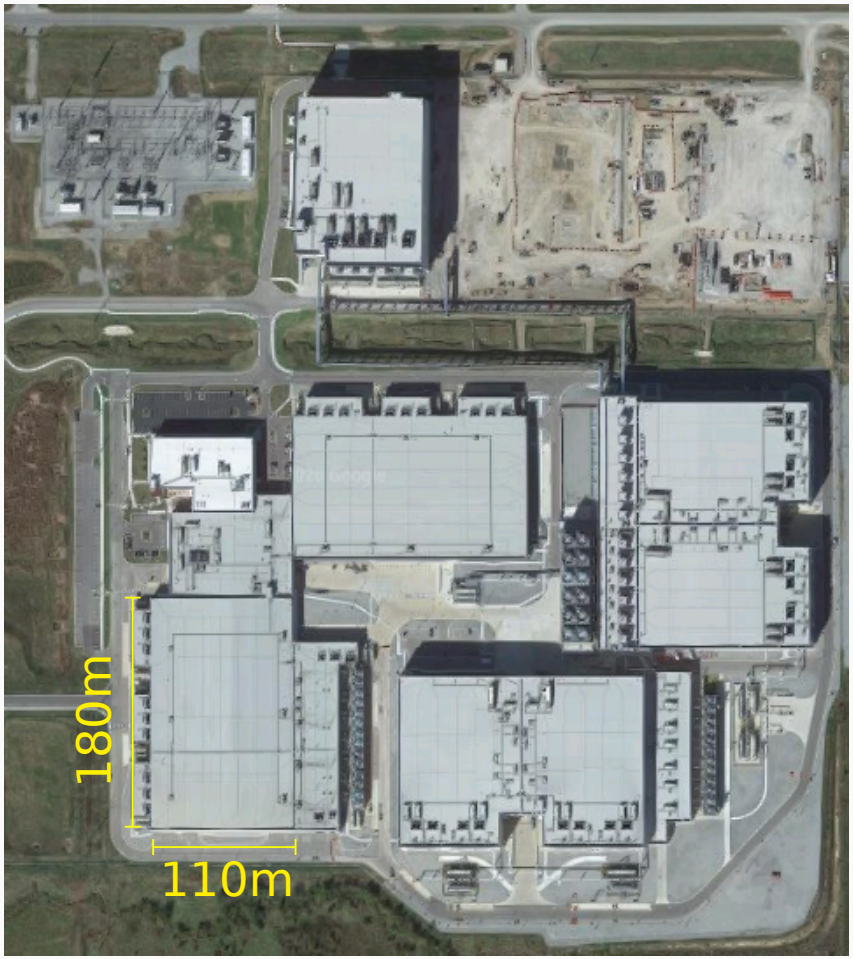
- Computers used for running large programs for multiple users, typically accessed only via a network.
- Price: \$5,000 – \$2M

Cloud Computing

- 10-100K of servers
- housed in large data centres

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

CERN data centre (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: destkop

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

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: 2GB of 667MHz DDR2 SDRAM
- Storage: 64GB SSD
- Input/Output (I/O) devices:
  keyboard, touchpad, screen, speakers, USB port, Wireless/Ethernet

PMD: Personal Mobile Device

- Price: \$100 – \$1000
- All the elements of a computer are there: touchscreen, virtual keyboard, wireless, processors, memory, storage
- Majority of these devices use ARM processors
  $\Rightarrow$ energy-efficient processors (2W)

first iPhone, 2007

first iPad, 2010

Embedder Computers

A computer inside another device

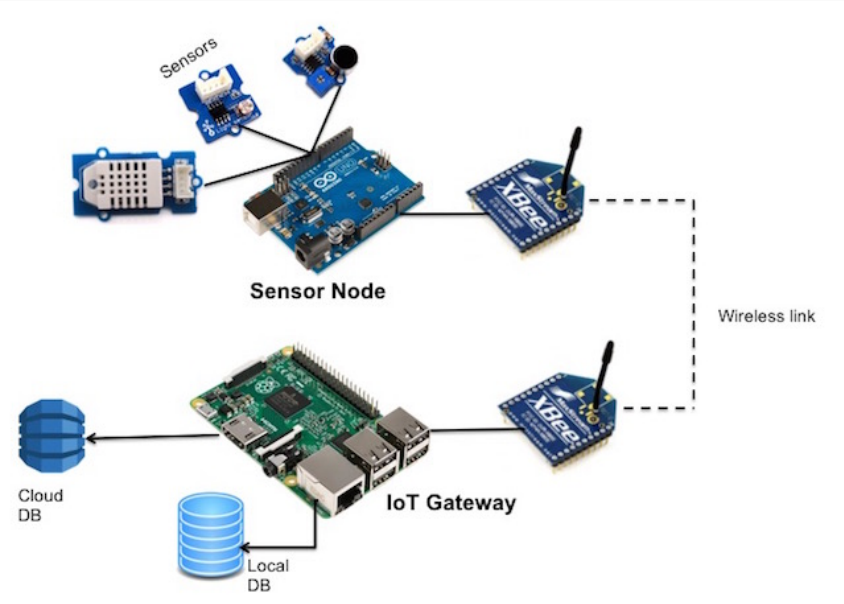
Often referred to as an *embedded system*

- 98% of all processors are in embedded systems
- Users do not necessarily realize they are intereacting with a computer
- *e.g.* 25-50 processors in a typical car
  - Engine management
  - Safety systems
    (ABS, skid control, pedestrian detection)
  - Input: sensors (*e.g.* accelerometer)
    output: mechanical control

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

Internet of Things (IoT)

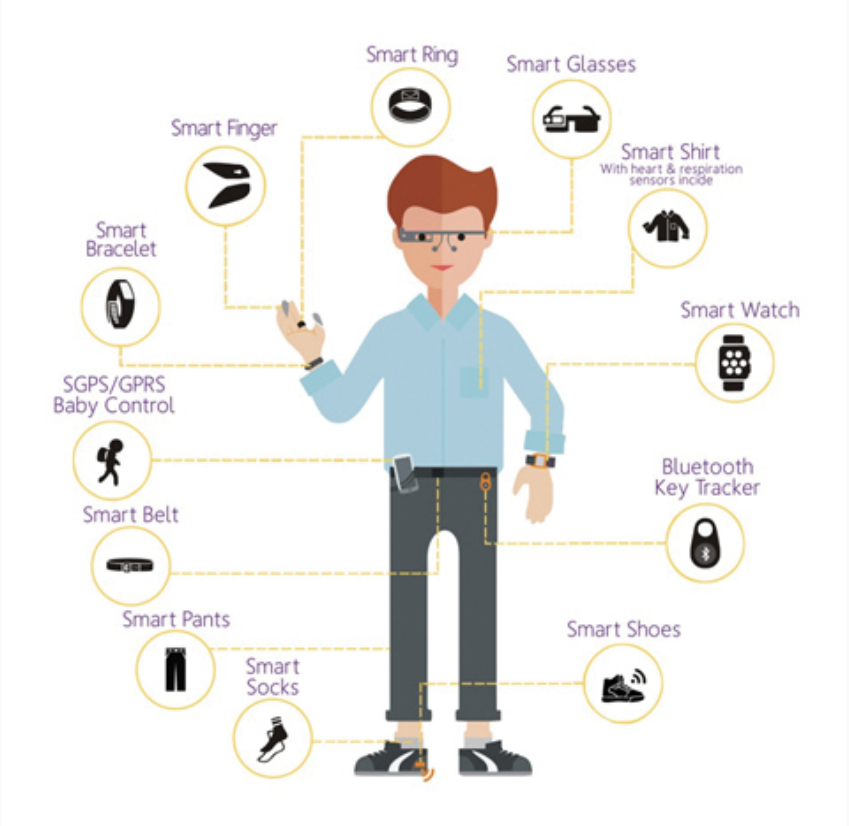
- Embedded systems
- Connected to other devices



source: thenewstack.io

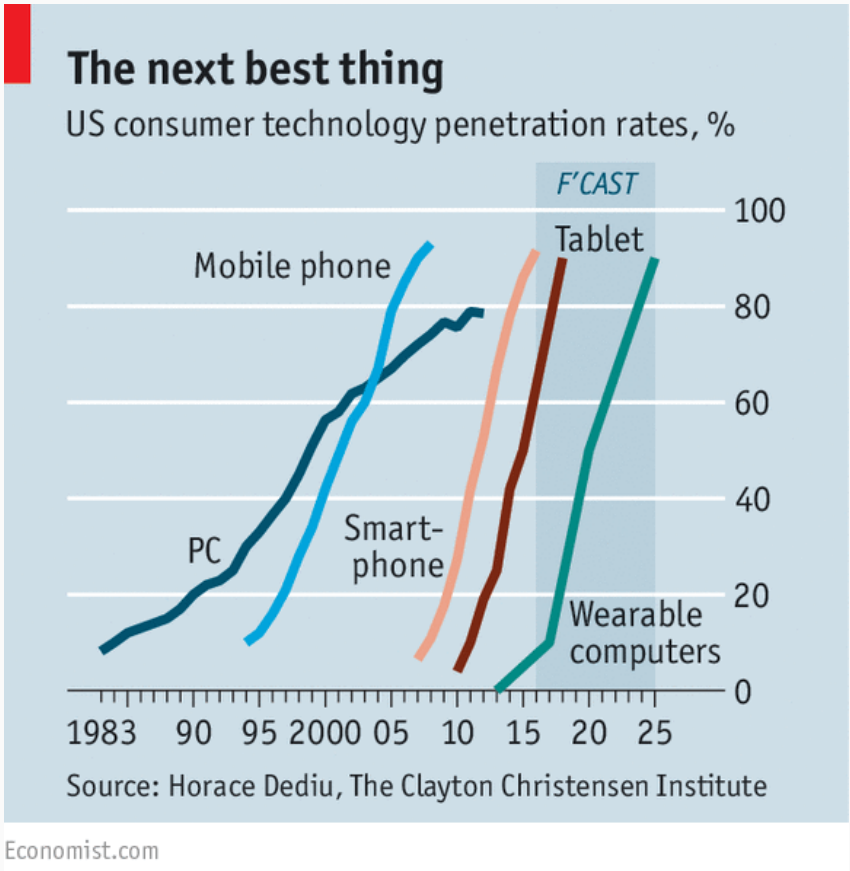
Arduino - Rasberry Pi IoT nodes

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

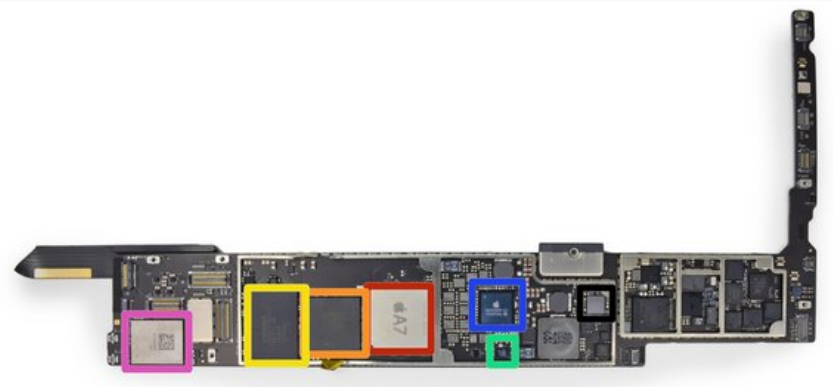
Under the hood

What's under the hood?

source: www.ifixit.com

iPad Air LTE Teardown

Main board ("Mother" board)

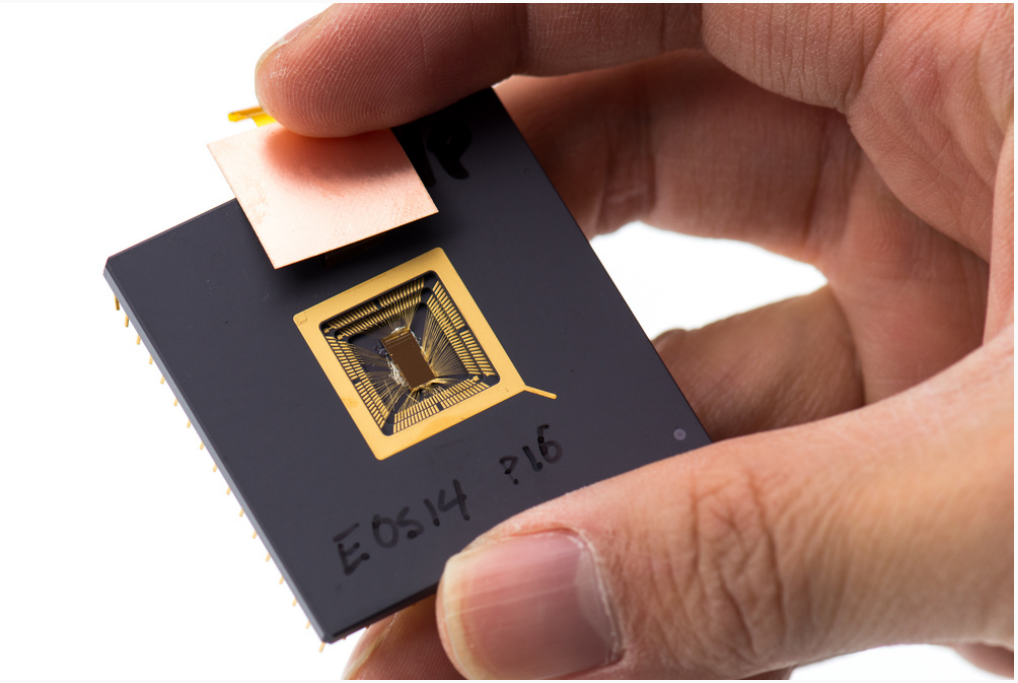


source: www.ifixit.com

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



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

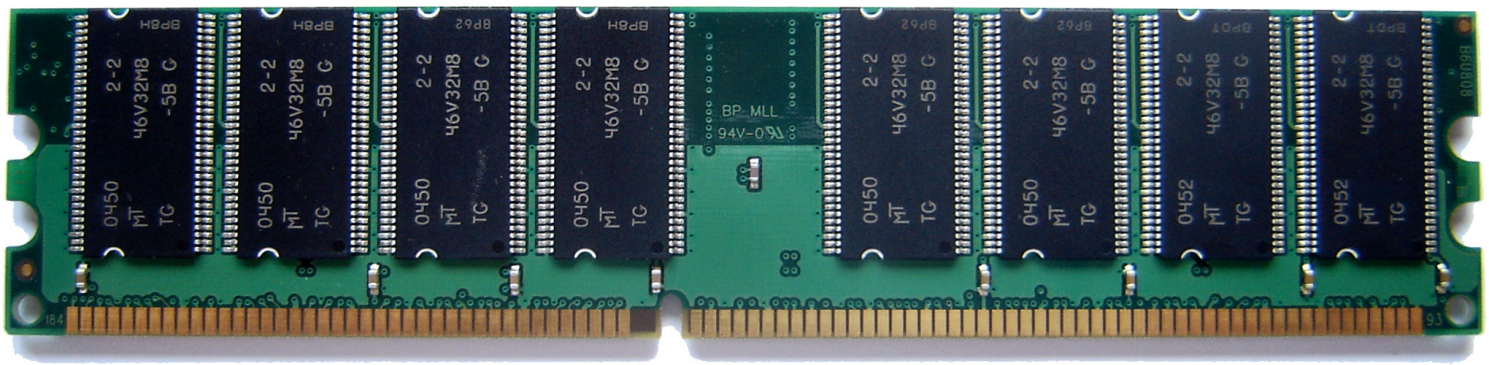
RISC V prototype chip, 2013

Components:

- Processor die: single piece of semiconductor (silicon)
- Processor package: plastic/ceramic housing with gold pin contacts

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

Memory



1GB Ram module

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

This is where the data to be processed is temporarily stored.

Storage

source: Evan-Amos / CC BY-SA 3.0

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

source: https://www.newegg.ca

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

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

This is where data is stored for the long-term.

I/O Devices

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

Use to exchange data with humans or other machines.

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:

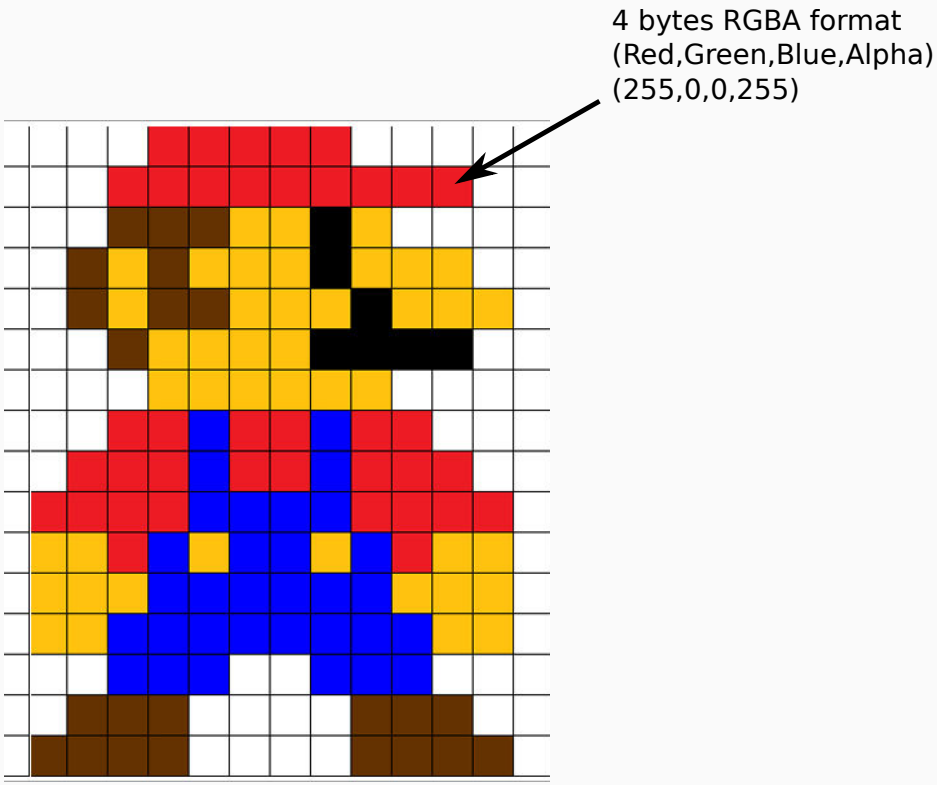
- easy to code
- easy to understand
- easy to port to different machines

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`.

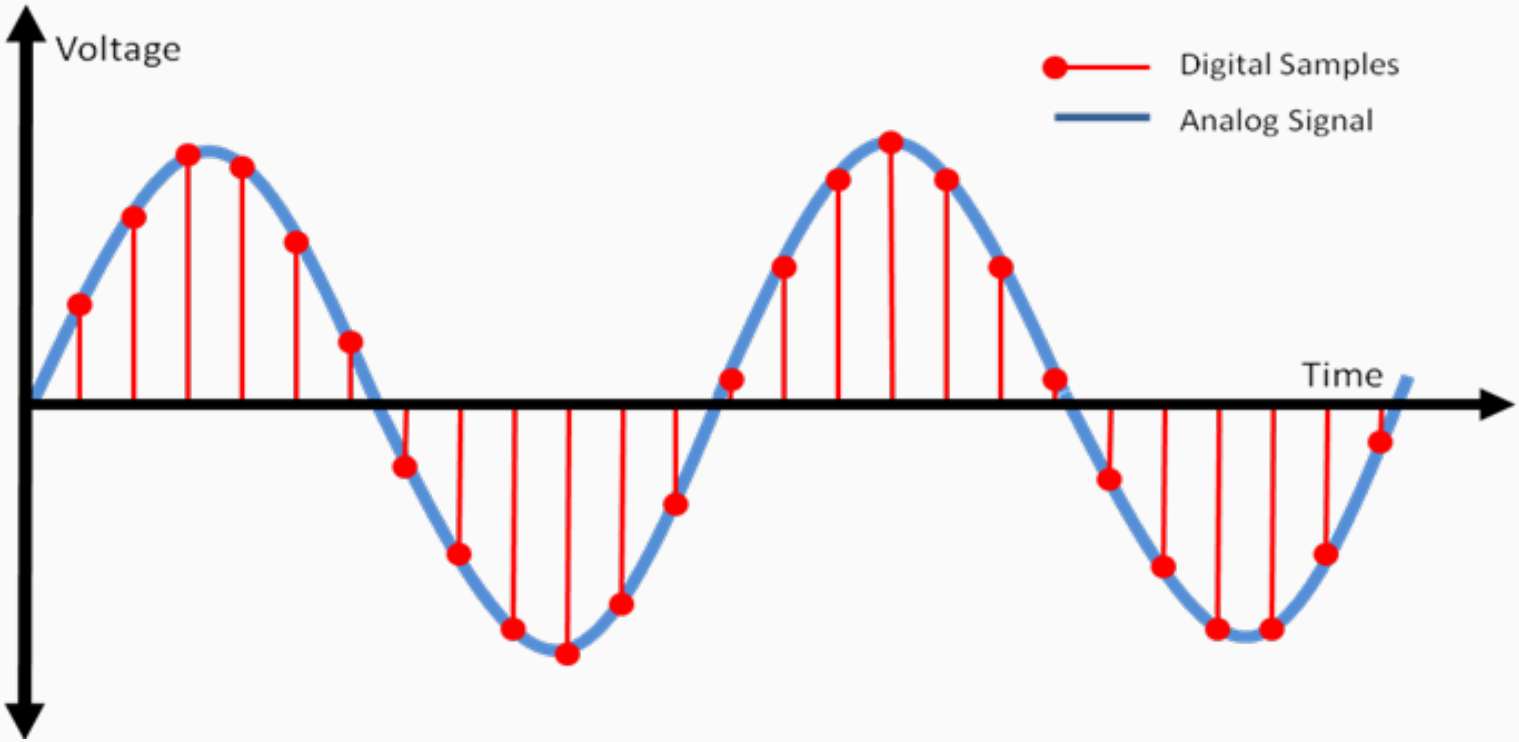
How about images?

- 2D array of picture elements (pixels)
- Each pixel is represented by a set of number indicating the intensity of colours such as red, green and blue.



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 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:

- *Memory* and
- *a clock* (synchronous sequential circuit)

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:

Logic functions, arithmetic, reading and writing memory

The 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

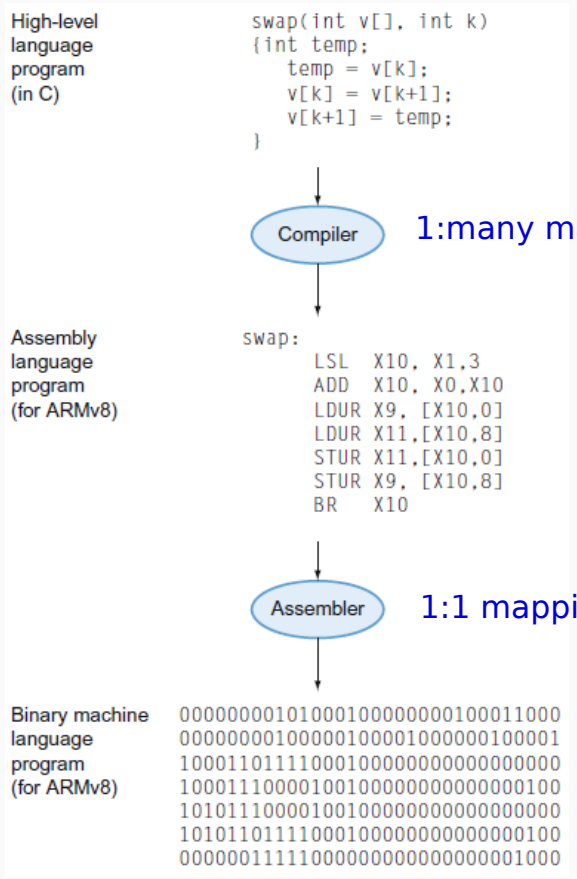
Instructions represent very simple operations: implemented with digital logic.

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



Where are the instructions stored?

Two type of architectures:

- *Hardvard architecture* (1944)
  Instructions and data are stored in separate memory.
- *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 from the outside, internally, Hardvard architecture is used, *i.e.* data/instruction caches).

Binary Integer Arithmetic (Recap)

Textbook§1.4, 1.5

Unsigned integers

Decimal number $D = d_{n-1} d n-2 \ldots d_1 d_0$ where $d_i \in \{0, 1, \ldots, 9\}$

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 $B = b_{n-1} b_{n-2} \ldots b_1 b_0$ 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. $13_{10} = 0000\ 1101_2$

Range of values depends on number of bits n: $V(D) \in [0; 2^n - 1]$

e.g. $n = 8$ bits, maximum value is $2^8 - 1 = 255_{10} = 1111\ 1111_2$

Binary Addition

Decimal addition

$$\begin{array}{rr} & 67 \\ + & 13 \\ \hline & {\scriptstyle 1} \\ & 80 \end{array}$$

Binary addition

$$\begin{array}{rr} & 0100\ 0011 \\ + & 0000\ 1101 \\ \hline & {\scriptstyle 1\ 111} \\ & 0101\ 0000 \end{array}$$

Binary Addition

Decimal addition

$$\begin{array}{rr} & 67 \\ + & 13 \\ \hline & {\scriptstyle 1} \\ & 80 \end{array}$$

Binary addition

$$\begin{array}{rr} & 0100\ 0011 \\ + & 0000\ 1101 \\ \hline & {\scriptstyle 1\ 111} \\ & 0101\ 0000 \end{array}$$

Watch out for overflow!

$$\begin{array}{rr} & 195 \\ + & 141 \\ \hline & {\scriptstyle 1} \\ & 336 \end{array}$$

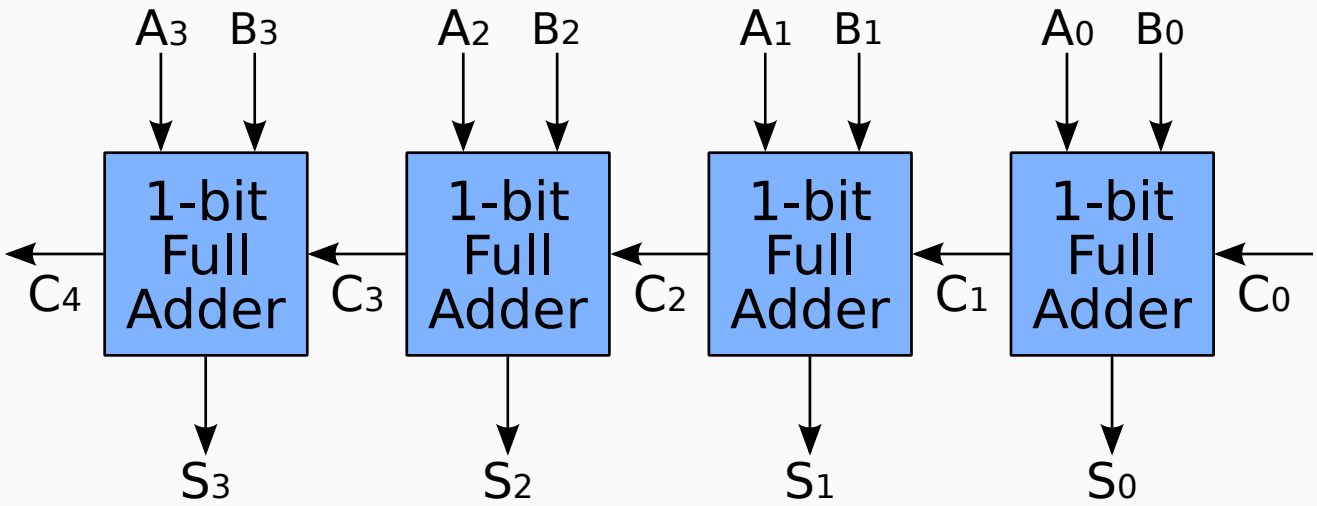
$$\begin{array}{rr} & 1100\ 0011 \\ + & 1000\ 1101 \\ \hline & {\scriptstyle 1\quad 1\ 111} \\ & 0101\ 0000 \end{array}$$

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: $S = A + B$



Signed integer: Sign-and-magnitude representation

Need to encode the sign in the representation.

Sign-and-magnitude

Simplest approach, use the leftmost bit (MSB) to represent the sign, and the remaining bit to represent the magnitude (*i.e.* absolute value).

$MSB = 0 \Rightarrow$ *positive*
$MSB = 1 \Rightarrow$ *negative*

Example for 8 bits:
$+13 = 0000\ 1101$
$-13 = 1000\ 1101$

Signed integer: Sign-and-magnitude representation

Need to encode the sign in the representation.

Sign-and-magnitude

Simplest approach, use the leftmost bit (MSB) to represent the sign, and the remaining bit to represent the magnitude (*i.e.* absolute value).

$MSB = 0 \Rightarrow$ *positive*
$MSB = 1 \Rightarrow$ *negative*

Example for 8 bits:
$+13 = 0000\ 1101$
$-13 = 1000\ 1101$

Problems:

- two representations for zero $= 0000\ 0000 = 1000\ 0000$
- need extra hardware to handle addition of a positive number with a negative one (cannot simply add the numbers together)

Signed Integer: 1's-complement representation

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

$+13 = 0000\ 1101$
$-13 = 1111\ 0010$

Signed Integer: 1's-complement representation

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

$+13 = 0000\ 1101$

$-13 = 1111\ 0010$

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

off by one and

carry out but not overflow

Signed Integer: 1's-complement representation

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

$+13 = 0000\ 1101$

$-13 = 1111\ 0010$

$$\begin{array}{rll} & 0001\ 0000 & = (16_{10}) \\ + & 1111\ 0010 & = (-13_{10}) \\ \hline {\scriptstyle 1}\,{\scriptstyle 111} & & \\ & 0000\ 0010 & = (2_{10}) \end{array}$$

off by one and

carry out but not overflow

Problems:

- Still two representations for zero $= 0000\ 0000 = 1111\ 1111$
- Need to add 1 to the result when adding to a negative number (try as an exercise with $(-2)_{10} + (-2)_{10}$).
- Need to handle overflow correctly

Signed Integer: 2's-complement representation

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

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

Signed Integer: 2's-complement representation

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

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

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

Carry out but no overflow!

Problem:

- Need to detect overflow correctly

Ranges

Assuming $n = 4$ bits integer representations

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 unsigned addition with 2's complement

Occurs when the answer does not fit into the range of the n-bit 2's complement representation: $[-2^{n-1}; 2^{n-1} - 1]$.

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.

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

No carry out, different sign

$\Rightarrow$ overflow!

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

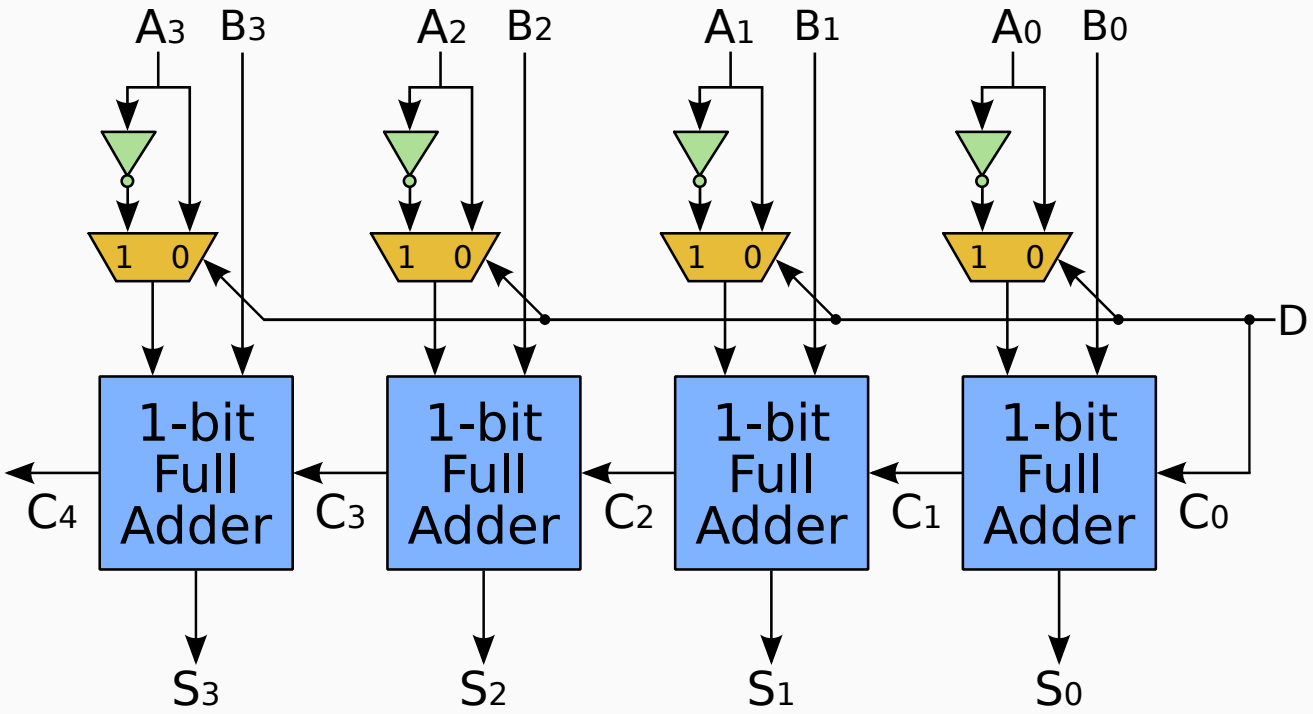
Carry out, different sign

$\Rightarrow$ overflow!

Subtraction

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

In hardware, inverse the bits and add one using the carry-in. D signal 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.

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

Hexadecimal Representation

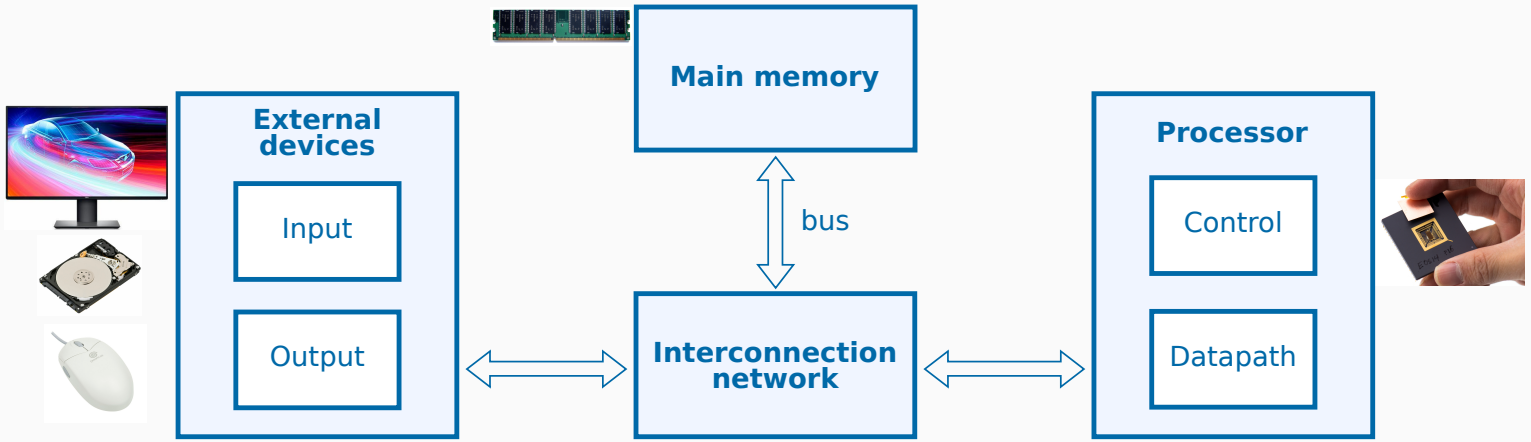
- Binary can be very unwieldy to represent large values: $7748_{10} = 0001111001000100_2$
- So we can use the base-16 hexadecimal (hex) representation. Each hex digit has $16 = 2^4$ possible values and represents 4 bits.
- We can write the above binary number more compactly in base-16 as $1E44_h$
- Get 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

Components (abstractions)



Processor

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

Main Memory

- RAM: Random Access Memory
- stores data and program

External devices

- IO (Input/Output)
- keyboard, hard-disk, display

Interconnection network

- Interconnect
- uses shared buses

Basic computer organization

Textbook§1.2, 1.3, 2.1, 2.2

Memory

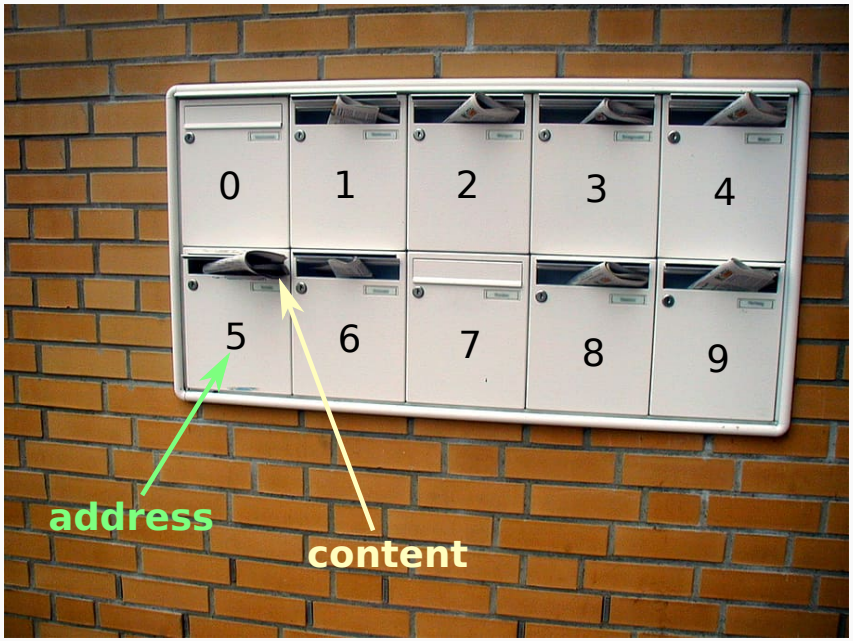
Main memory

Usually, data in a program resides in main memory. Conceptually, 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



- The computer memory is a linear array of byte.
- Each byte in the memory has it's 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 4GB (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*).

Multiple bytes access example

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) = 0x90124FEE

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

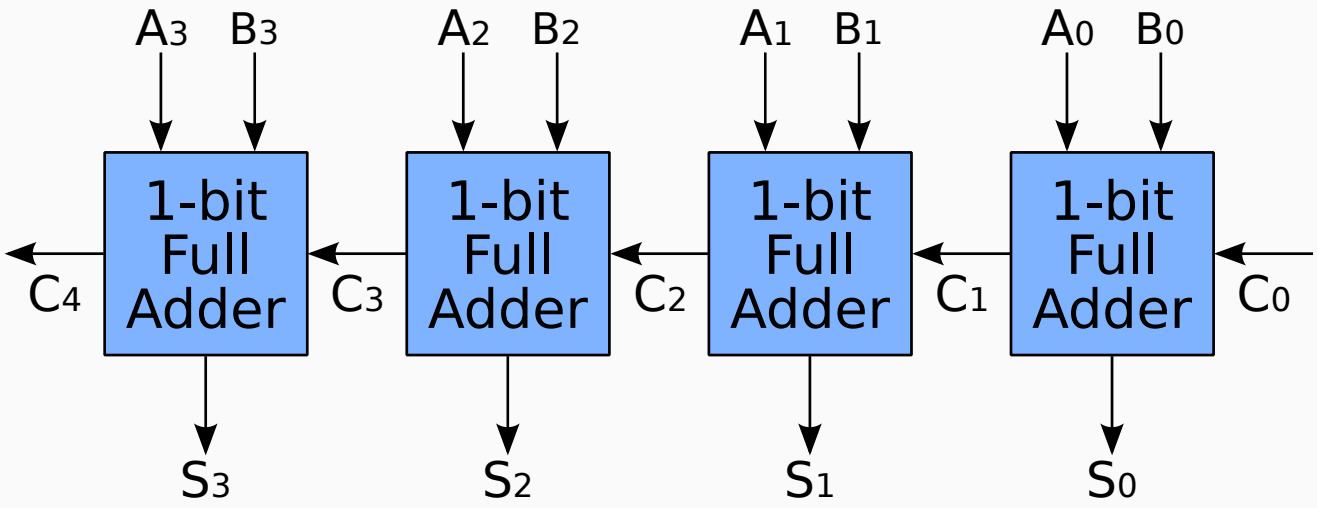
Little Endian : starts with the Least Significant Byte (Little byte).
E.g. Read(0x04) = 0x EE 4F 12 90

Origin of the name

Gulliver's Travels, 1726
(Jonathan Swift).

Why Little-Endian?

One of the reaons: consider the ripple carry adder:



The carry chain starts with the least significant bit. When the first byte-addressable micro-controllers and processors appeared, made it more efficient to implement addition: 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:

$$addr \bmod sze = 0$$

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

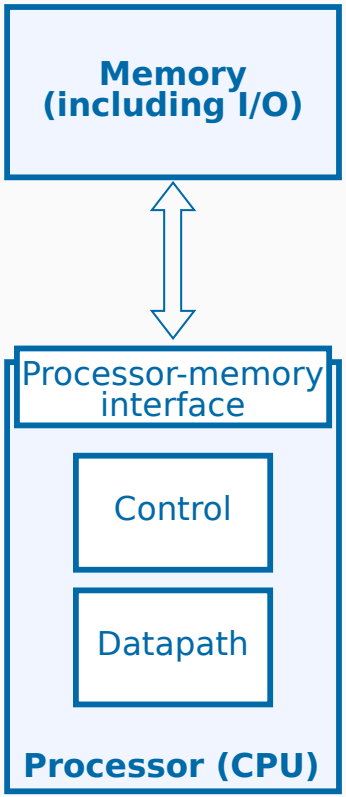
address	value
0x00	0xD1
0x01	0x4B
0x02	0x45
0x03	0xC4
0x04	0x90
0x05	0x12
0x06	0x4F
0x07	0xEE
0x08	0x78
0x09	0x91
0x0A	0x03
0x0B	0x70
0x0C	0xB3
0x0D	0xDA
0x0E	0x7F
0x0F	0xE6

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 interact with the outside world through a *memory interface*.

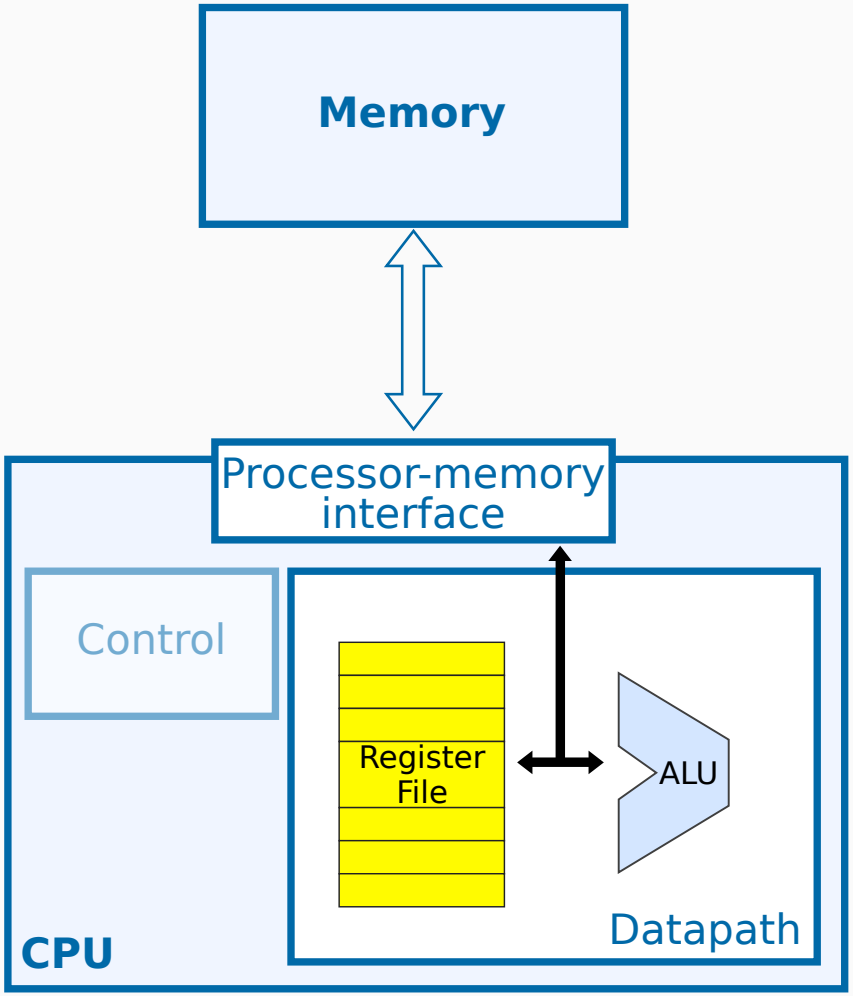
CPU's Control

- The control logic coordinates the execution of the instructions in the datapath.
- Mainly consists of finite state machines

CPU's Datapath

- The datapath is driven by the control logic and processes the data depending on the instruction.

Processor Datapath



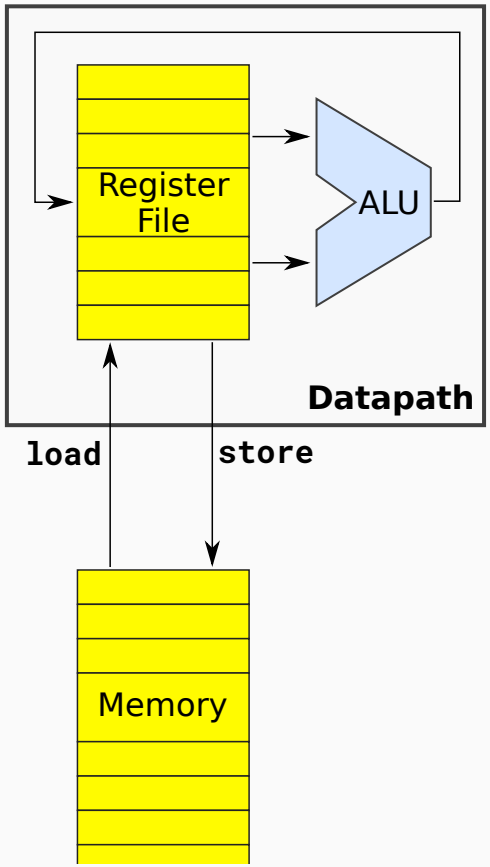
Arithmetic and Logic Unit (ALU)

- Performs operations on data
- *e.g.* add, multiply, shift, and

Register File

- 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
  $\Rightarrow$ simplifies the design of the hardware.
- Use special *load* and *store* instructions for transfers between registers and memory.

CISC vs RISC architecture

CISC = Complex Instruction Set Computer

- Larger number of instructions
- Instructions can access both memory and registers (*e.g.* add)
- Leads to more complex CPU design

RISC = Reduced Instruction Set Computer

- Focus on simple CPU design
- Small number of instructions (*e.g.* may not have division, emulated in software)
- Load/Store architecture
- Focus of this course

RISC Operations

Textbook§2.3

Register Transfer Notation (RTN)

RTN allows us to specify the semantic of an instruction

- `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!

Memory instructions

Memory Load

```
Load R2, ADDRESS
```

reads (copies) 4 consecutive bytes from the memory starting at memory address ADDRESS and writes them as a word into register R2.

Register Transfer Notation: $R2 \leftarrow Mem[ADDRESS]$

Memory Store

```
Store R4, ADDRESS
```

Copies the word stored in register R4 into four consecutive bytes in memory starting from address ADDRESS

Register Transfer Notation: $Mem[ADDRESS] \leftarrow R4$

ALU instructions

e.g. Addition

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

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.

Register Transfer Notation: $\texttt{R4} \leftarrow \texttt{R2} + \texttt{R3}$

Sequence of instructions

Consider the C program, where a, b, c each occupies 4 byte in memory:

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

The equivalent sequence of assembly instructions are:

```
Load  R2, a
Load  R3, b
Add   R4, R2, R3
Store R4, c
```

where a,b and c are the addresses of where the variables are stored.

Running a program

When running the program, the machine executes each instruction sequentially, one after another (or at least pretends of doing so).

Program & Data stored in the same memory

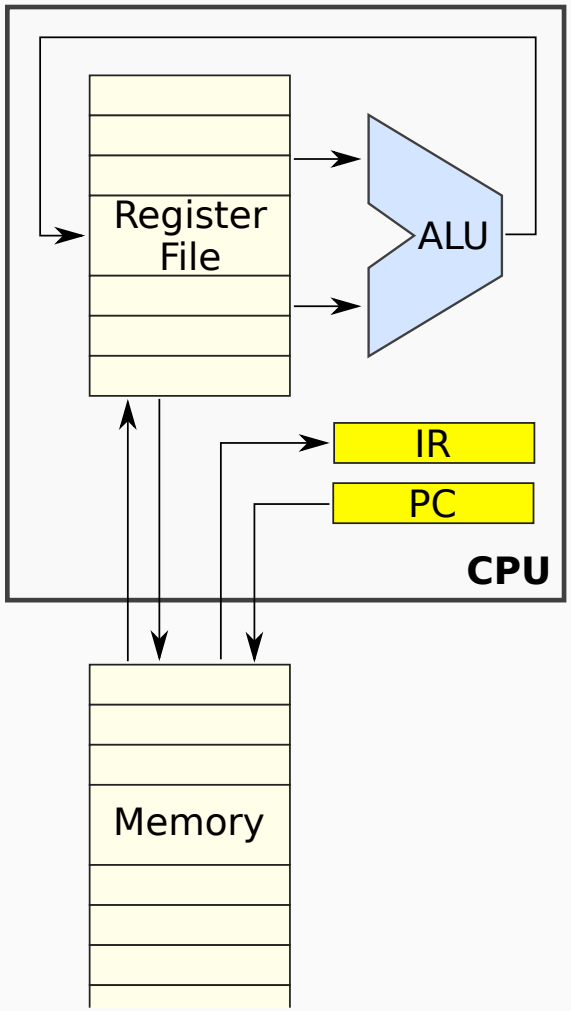
- Instructions are like data, stored in memory.
- On a typical RISC machine, all instructions are of the same length (*e.g.* 4 bytes)
- Instructions are stored in consecutive memory addresses:
  e.g. 0x00, 0x04, 0x08, 0x0C.

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,
show the memory and registers content after each instruction.

Program Counter and Instruction Register



Program Counter (PC)

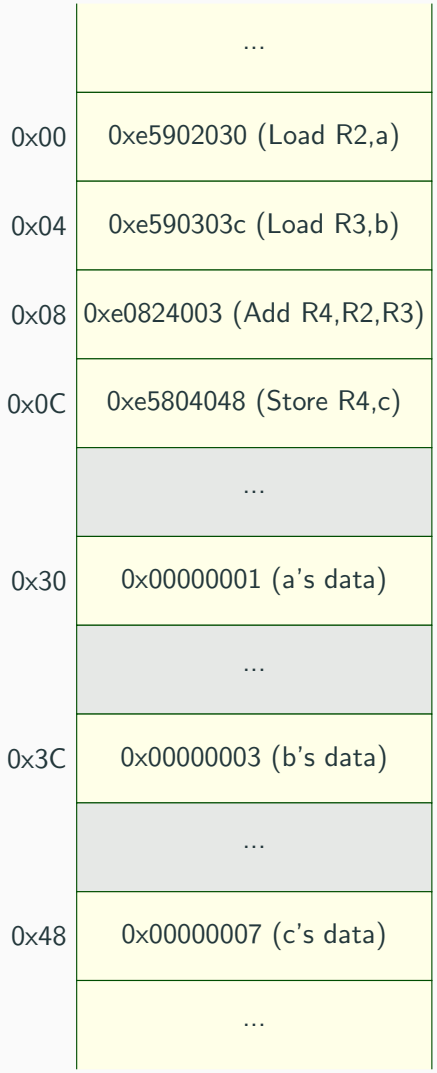
- *points* to the current instruction executing
- PC's content is an *address* in memory

Instruction Register (IR)

- contains the *current instruction* to execute

- When executing, the processor *fetch* the instruction pointed by the PC from memory and stores it into the IR.
- Then, the instruction is *decoded* and the processor *executes* it.
- Finally, the PC is updated, usually incremented by the instruction size.

Executing instructions step by step

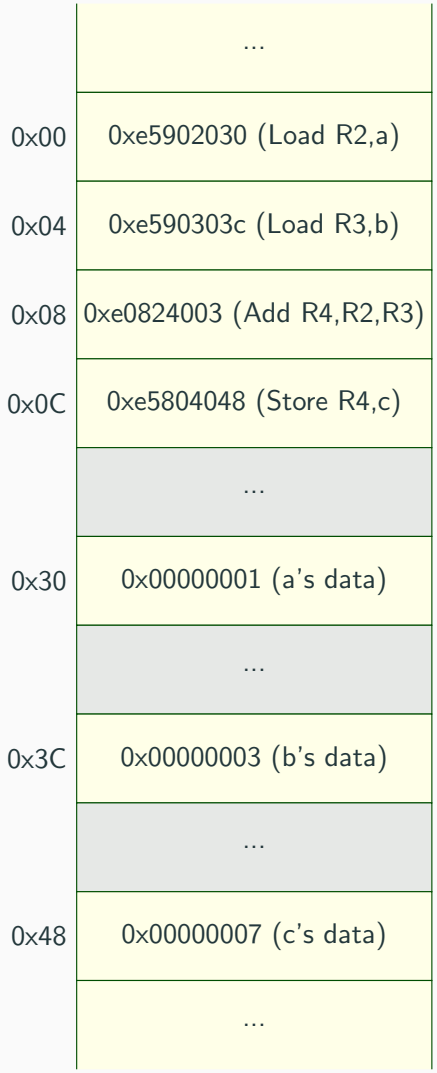


Registers

PC	0x00000000
IR	?
R2	?
R3	?
R4	?

1. `PC = 0x00000000`

Executing instructions step by step

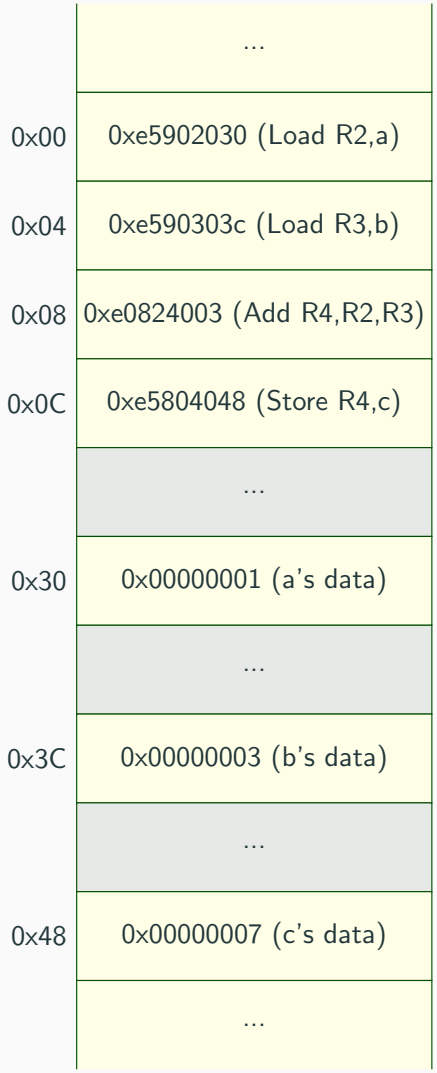


Registers

PC	0x00000000
IR	0xe5902030
R2	?
R3	?
R4	?

1. `PC = 0x00000000`
2. fetch $\Rightarrow$ `IR = 0xe5902030`

Executing instructions step by step

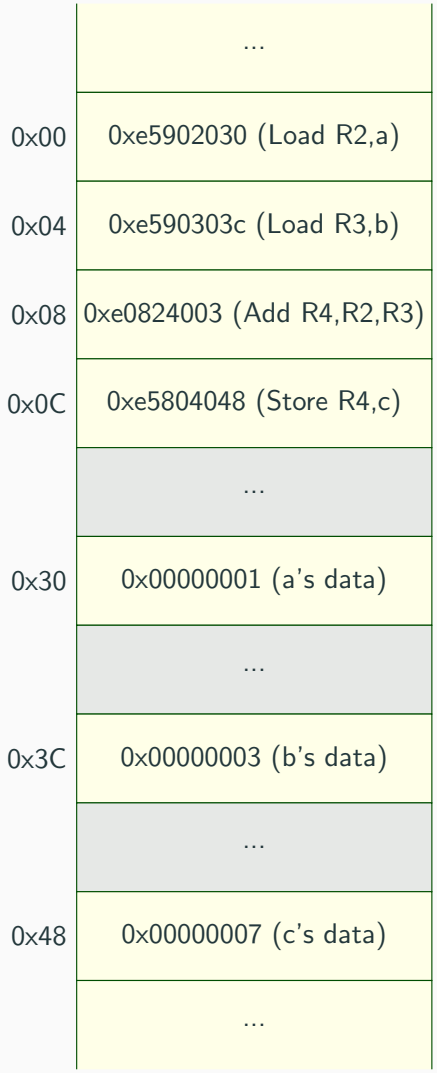


Registers

PC	0x00000000
IR	0xe5902030
R2	?
R3	?
R4	?

1. `PC = 0x00000000`
2. fetch $\Rightarrow$ `IR = 0xe5902030`
3. decode $\Rightarrow$ `Load R2, a`

Executing instructions step by step

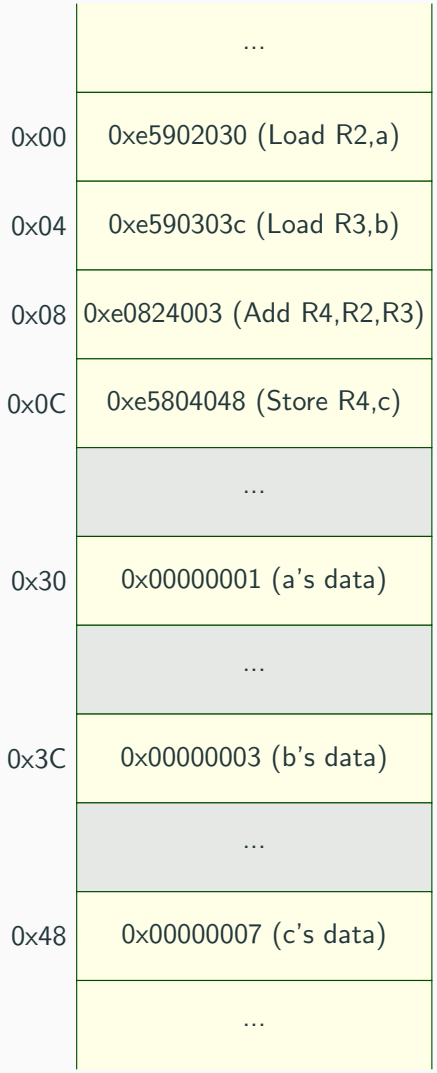


Registers

PC	0x00000000
IR	0xe5902030
R2	0x00000001
R3	?
R4	?

1. `PC = 0x00000000`
2. fetch ⇒ `IR = 0xe5902030`
3. decode ⇒ `Load R2, a`
4. execute ⇒ `R2 = 0x00000001`

Executing instructions step by step

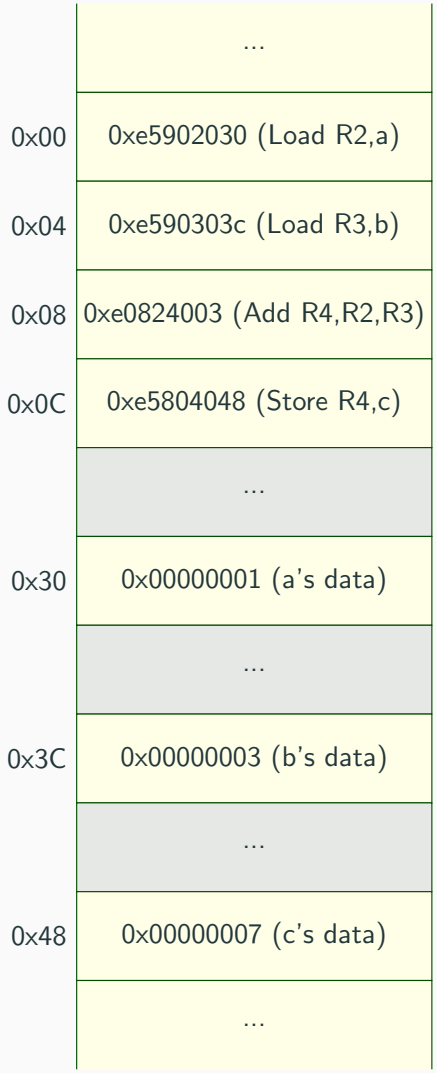


Registers

PC	0x00000004
IR	0xe5902030
R2	0x00000001
R3	?
R4	?

1. `PC = 0x00000000`
2. fetch $\Rightarrow$ `IR = 0xe5902030`
3. decode $\Rightarrow$ `Load R2, a`
4. execute $\Rightarrow$ `R2 = 0x00000001`
5. `PC += 4` $\Rightarrow$ `PC = 0x00000004`

Executing instructions step by step

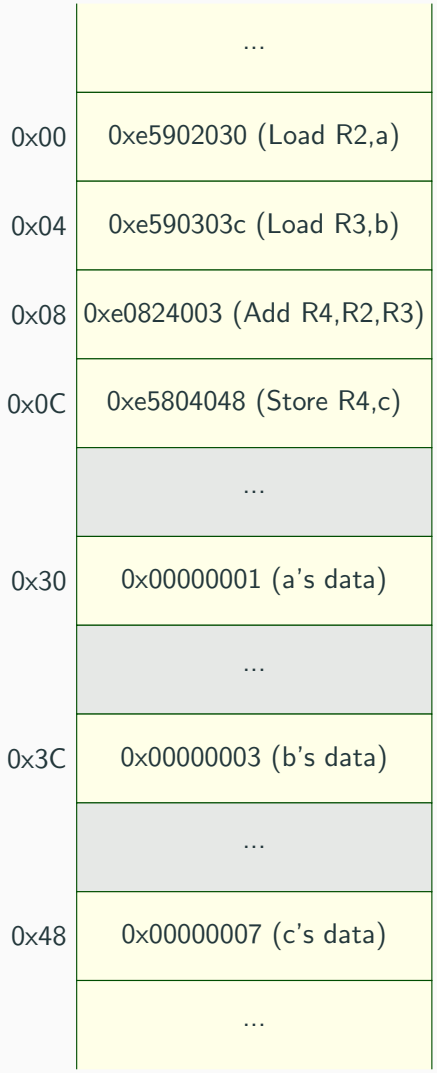


Registers

PC	0x00000004
IR	0xe590303c
R2	0x00000001
R3	?
R4	?

1. `PC = 0x00000000`
2. fetch $\Rightarrow$ `IR = 0xe5902030`
3. decode $\Rightarrow$ `Load R2, a`
4. execute $\Rightarrow$ `R2 = 0x00000001`
5. `PC += 4` $\Rightarrow$ `PC = 0x00000004`
6. fetch $\Rightarrow$ `IR = 0xe590303c`

Executing instructions step by step

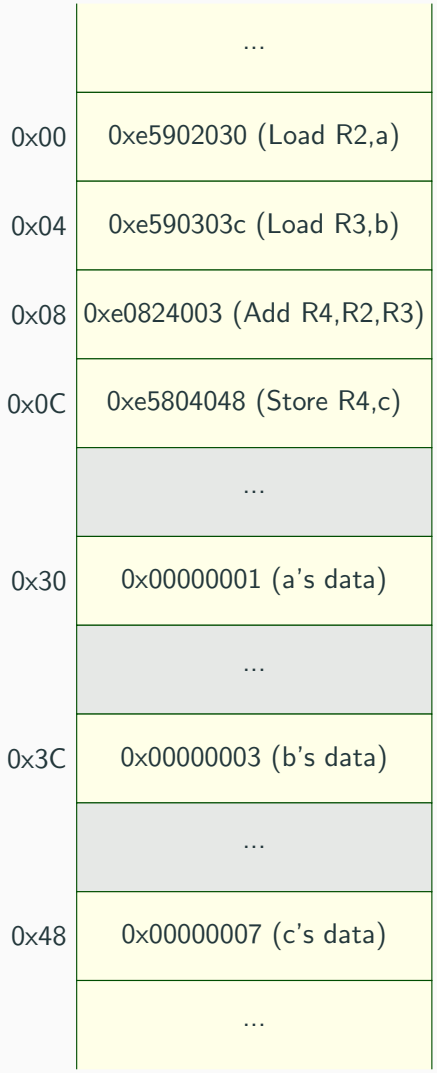


Registers

PC	0x00000004
IR	0xe590303c
R2	0x00000001
R3	?
R4	?

1. `PC = 0x00000000`
2. fetch ⇒ `IR = 0xe5902030`
3. decode ⇒ `Load R2, a`
4. execute ⇒ `R2 = 0x00000001`
5. `PC += 4` ⇒ `PC = 0x00000004`
6. fetch ⇒ `IR = 0xe590303c`
7. decode ⇒ `Load R3, b`

Executing instructions step by step

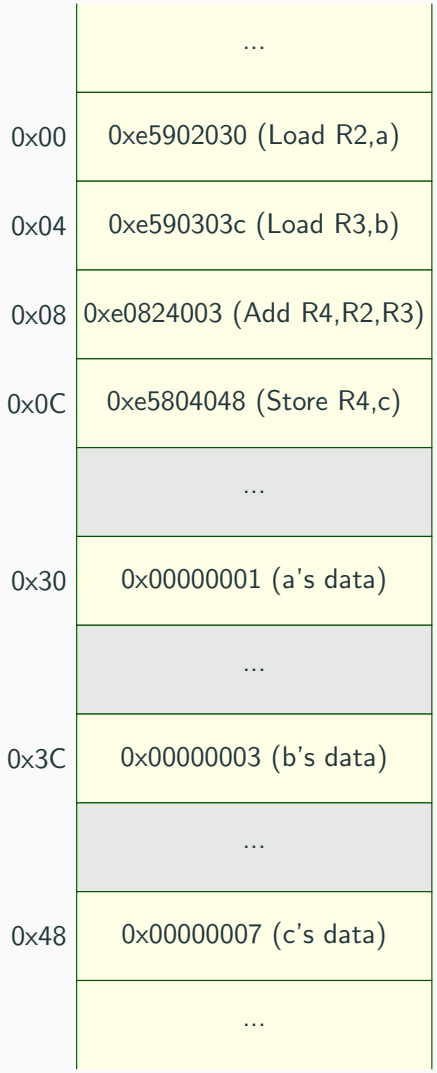


Registers

PC	0x00000004
IR	0xe590303c
R2	0x00000001
R3	0x00000003
R4	?

1. `PC = 0x00000000`
2. fetch ⇒ `IR = 0xe5902030`
3. decode ⇒ `Load R2, a`
4. execute ⇒ `R2 = 0x00000001`
5. `PC += 4` ⇒ `PC = 0x00000004`
6. fetch ⇒ `IR = 0xe590303c`
7. decode ⇒ `Load R3, b`
8. execute ⇒ `R3 = 0x00000003`

Executing instructions step by step

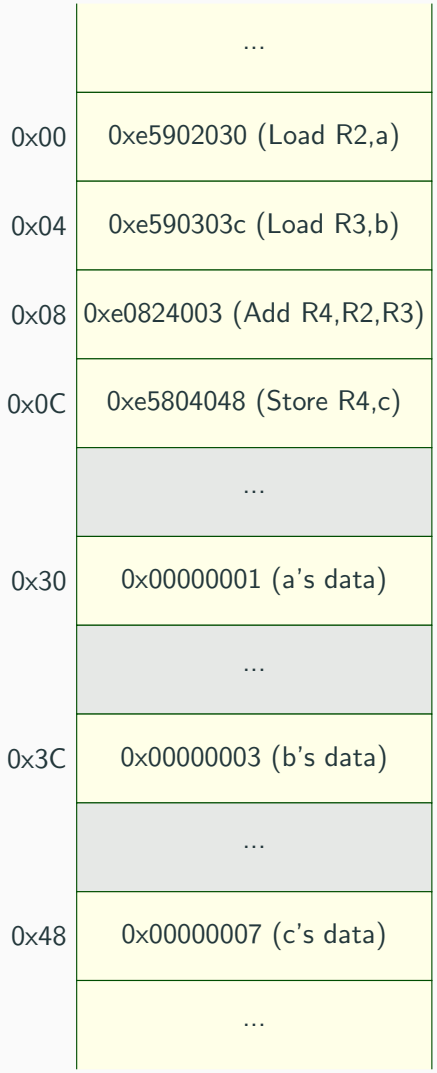


Registers

PC	0x00000008
IR	0xe590303c
R2	0x00000001
R3	0x00000003
R4	?

1. `PC = 0x00000000`
2. fetch ⇒ `IR = 0xe5902030`
3. decode ⇒ `Load R2, a`
4. execute ⇒ `R2 = 0x00000001`
5. `PC += 4` ⇒ `PC = 0x00000004`
6. fetch ⇒ `IR = 0xe590303c`
7. decode ⇒ `Load R3, b`
8. execute ⇒ `R3 = 0x00000003`
9. `PC += 4` ⇒ `PC = 0x00000008`

Executing instructions step by step

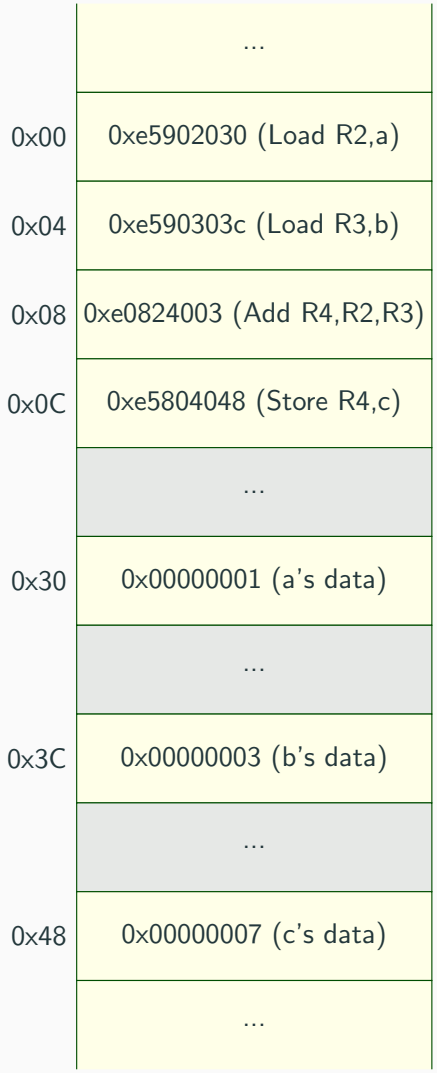


Registers

PC	0x00000008
IR	0xe0824003
R2	0x00000001
R3	0x00000003
R4	?

1. `PC = 0x00000000`
2. fetch ⇒ `IR = 0xe5902030`
3. decode ⇒ `Load R2, a`
4. execute ⇒ `R2 = 0x00000001`
5. `PC += 4` ⇒ `PC = 0x00000004`
6. fetch ⇒ `IR = 0xe590303c`
7. decode ⇒ `Load R3, b`
8. execute ⇒ `R3 = 0x00000003`
9. `PC += 4` ⇒ `PC = 0x00000008`
10. fetch ⇒ `IR = 0xe0824003`

Executing instructions step by step

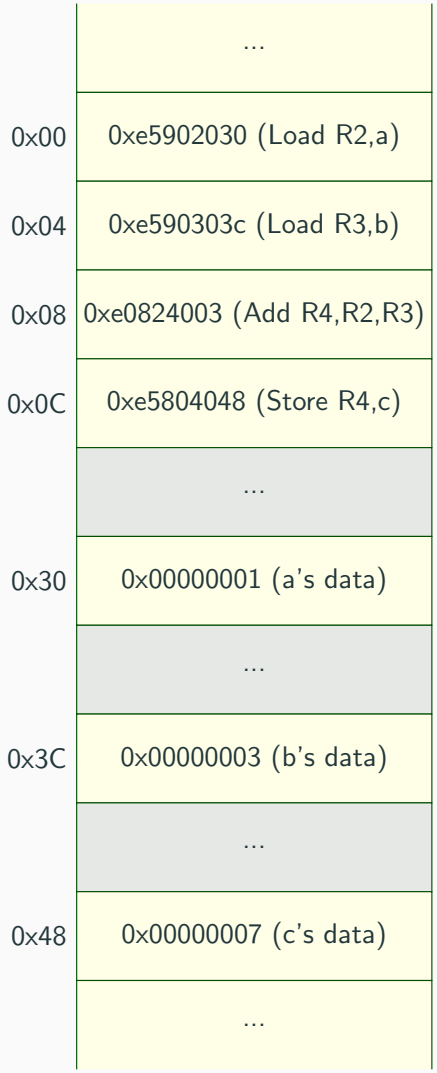


Registers

PC	0x00000008
IR	0xe0824003
R2	0x00000001
R3	0x00000003
R4	?

1. `PC = 0x00000000`
2. fetch ⇒ `IR = 0xe5902030`
3. decode ⇒ `Load R2, a`
4. execute ⇒ `R2 = 0x00000001`
5. `PC += 4` ⇒ `PC = 0x00000004`
6. fetch ⇒ `IR = 0xe590303c`
7. decode ⇒ `Load R3, b`
8. execute ⇒ `R3 = 0x00000003`
9. `PC += 4` ⇒ `PC = 0x00000008`
10. fetch ⇒ `IR = 0xe0824003`
11. decode ⇒ `Add R4, R2, R3`

Executing instructions step by step

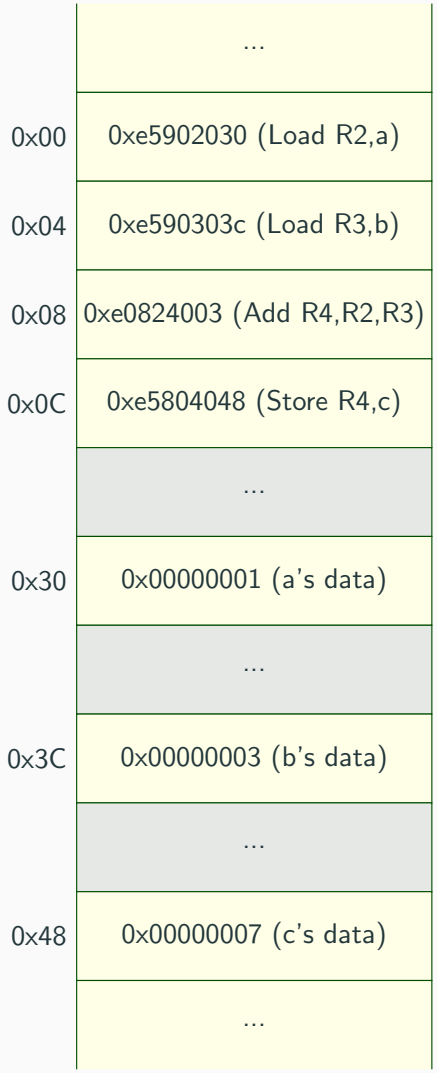


Registers

PC	0x00000008
IR	0xe0824003
R2	0x00000001
R3	0x00000003
R4	0x00000004

1. `PC = 0x00000000`
2. fetch ⇒ `IR = 0xe5902030`
3. decode ⇒ `Load R2, a`
4. execute ⇒ `R2 = 0x00000001`
5. `PC += 4` ⇒ `PC = 0x00000004`
6. fetch ⇒ `IR = 0xe590303c`
7. decode ⇒ `Load R3, b`
8. execute ⇒ `R3 = 0x00000003`
9. `PC += 4` ⇒ `PC = 0x00000008`
10. fetch ⇒ `IR = 0xe0824003`
11. decode ⇒ `Add R4, R2, R3`
12. execute ⇒ `R4 = 0x00000004`

Executing instructions step by step

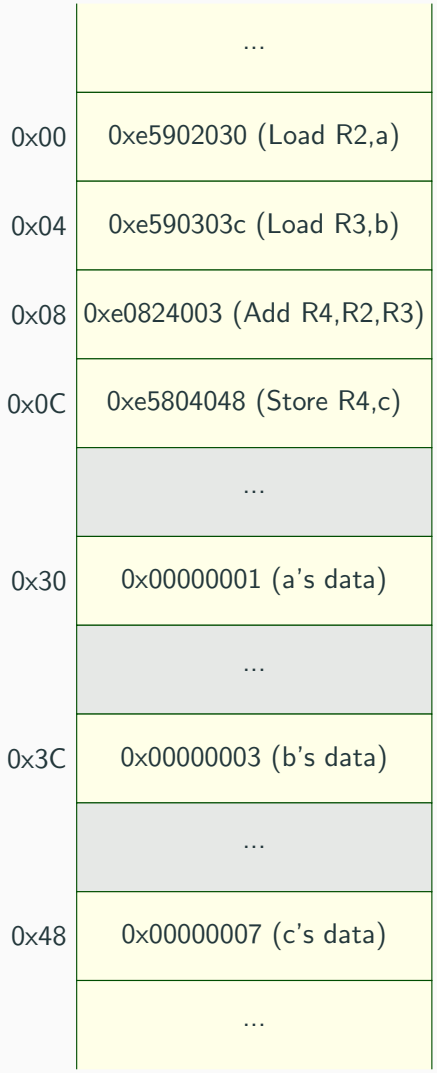


Registers

PC	0x0000000C
IR	0xe0824003
R2	0x00000001
R3	0x00000003
R4	0x00000004

1. `PC = 0x00000000`
2. fetch ⇒ `IR = 0xe5902030`
3. decode ⇒ `Load R2, a`
4. execute ⇒ `R2 = 0x00000001`
5. `PC += 4` ⇒ `PC = 0x00000004`
6. fetch ⇒ `IR = 0xe590303c`
7. decode ⇒ `Load R3, b`
8. execute ⇒ `R3 = 0x00000003`
9. `PC += 4` ⇒ `PC = 0x00000008`
10. fetch ⇒ `IR = 0xe0824003`
11. decode ⇒ `Add R4, R2, R3`
12. execute ⇒ `R4 = 0x00000004`
13. `PC += 4` ⇒ `PC = 0x0000000C`

Executing instructions step by step

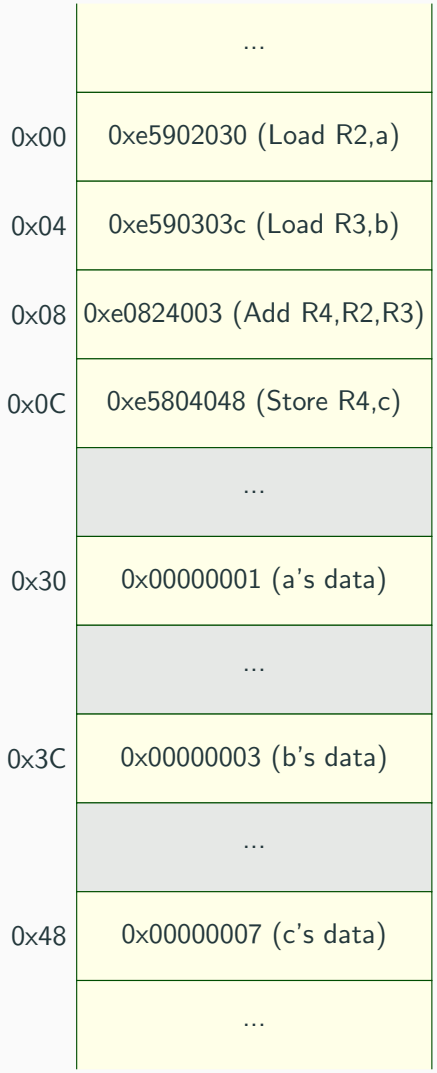


Registers

PC	0x0000000C
IR	0xe5804048
R2	0x00000001
R3	0x00000003
R4	0x00000004

1. `PC = 0x00000000`
2. fetch ⇒ `IR = 0xe5902030`
3. decode ⇒ `Load R2, a`
4. execute ⇒ `R2 = 0x00000001`
5. `PC += 4` ⇒ `PC = 0x00000004`
6. fetch ⇒ `IR = 0xe590303c`
7. decode ⇒ `Load R3, b`
8. execute ⇒ `R3 = 0x00000003`
9. `PC += 4` ⇒ `PC = 0x00000008`
10. fetch ⇒ `IR = 0xe0824003`
11. decode ⇒ `Add R4, R2, R3`
12. execute ⇒ `R4 = 0x00000004`
13. `PC += 4` ⇒ `PC = 0x0000000C`
14. fetch ⇒ `IR = 0xe5804048`

Executing instructions step by step

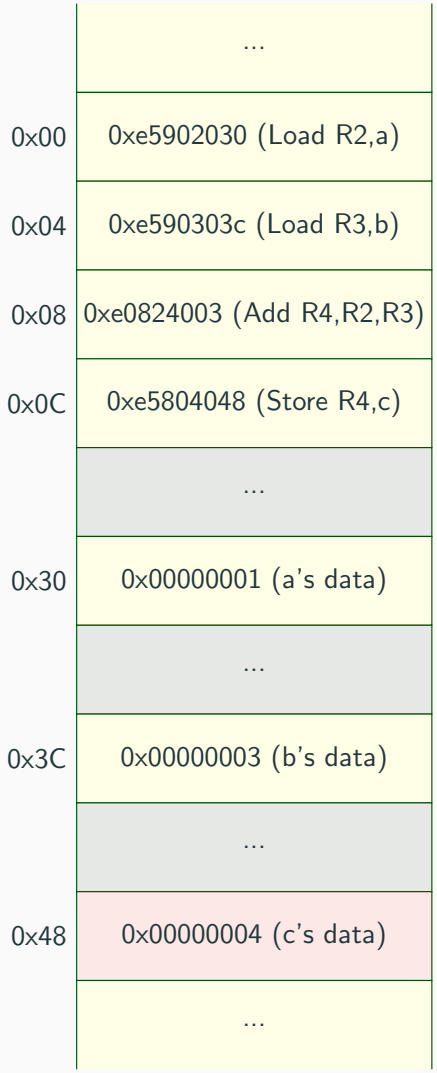


Registers

PC	0x0000000C
IR	0xe5804048
R2	0x00000001
R3	0x00000003
R4	0x00000004

1. `PC = 0x00000000`
2. fetch ⇒ `IR = 0xe5902030`
3. decode ⇒ `Load R2, a`
4. execute ⇒ `R2 = 0x00000001`
5. `PC += 4` ⇒ `PC = 0x00000004`
6. fetch ⇒ `IR = 0xe590303c`
7. decode ⇒ `Load R3, b`
8. execute ⇒ `R3 = 0x00000003`
9. `PC += 4` ⇒ `PC = 0x00000008`
10. fetch ⇒ `IR = 0xe0824003`
11. decode ⇒ `Add R4, R2, R3`
12. execute ⇒ `R4 = 0x00000004`
13. `PC += 4` ⇒ `PC = 0x0000000C`
14. fetch ⇒ `IR = 0xe5804048`
15. decode ⇒ `Store R4, c`

Executing instructions step by step



Registers

PC	0x0000000C
IR	0xe5804048
R2	0x00000001
R3	0x00000003
R4	0x00000004

1. `PC = 0x00000000`
2. fetch ⇒ `IR = 0xe5902030`
3. decode ⇒ `Load R2, a`
4. execute ⇒ `R2 = 0x00000001`
5. `PC += 4` ⇒ `PC = 0x00000004`
6. fetch ⇒ `IR = 0xe590303c`
7. decode ⇒ `Load R3, b`
8. execute ⇒ `R3 = 0x00000003`
9. `PC += 4` ⇒ `PC = 0x00000008`
10. fetch ⇒ `IR = 0xe0824003`
11. decode ⇒ `Add R4, R2, R3`
12. execute ⇒ `R4 = 0x00000004`
13. `PC += 4` ⇒ `PC = 0x0000000C`
14. fetch ⇒ `IR = 0xe5804048`
15. decode ⇒ `Store R4, c`
16. execute ⇒ memory is changed

Branching

How does a machine execute such a C code?

```
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 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 value from memory 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 0x10
0x10:     Move   R2, #13   // R2 = 13
0x14:     Store  R2, b     // store value of R2 into memory
```

- Br = Branch
- BLE = Branch if Lesser or Equal (to zero)
  (all 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 value from memory into R1
          Ble    R1, ELSE  // if R1<=0 branch to ELSE
          Move   R2, #7    // R2 = 7
          Br     END       // branch to END
ELSE:     Move   R2, #13   // R2 = 13
END:      Store  R2, b     // store value of R2 into memory
```

Assuming the first instruction is at address 0x00:

- ELSE = 0x10
- END = 0x14

Loops

How does the machine executes 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
```

What about:

- other loop constructs?
- 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)
- shown a recap on integer arithmetic
- looked at the memory and processor abstractions
- introduced basic RISC operations

The next lecture will:

- present a real processor instruction set (ARMv7)
- show how to write assembly programs in more details