ENSC254 – Intro to Computer Architecture

© 2021 -- Fabio Campi and Craig Scratchley
School of Engineering Science
Simon Fraser University
Burnaby, BC, Canada

Why ensc254?

1) ***If you want to be (good) software engineers, you need to have knowledge and visibility of the piece of HW you are going to instruct***

2) ***If you want to be system (HW/SW) engineers, you want to be able to instruct your machines taking into account low-level timings and interactons that are not visible from a purely SW perspective***

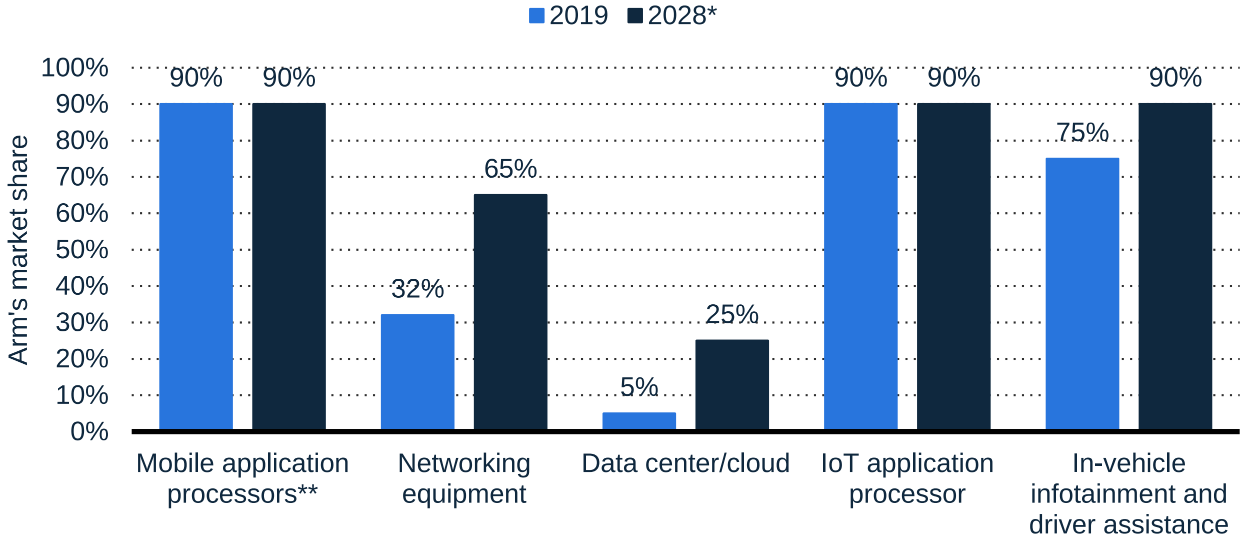
3) ***If you want to be HW designers, you want to build machines that are as programmable as possible so they can be reused in many different contexts***

4) ***If you eventually want to own your business or manage people, you want to know both the core aspects of a Computing system and the interactions that make it effective***

5) ***If you want to be a marketing person, you want to understand just enough of the two worlds in order to appreciate the added value of a given product***

Our Case Study -> Why ARM ?

- ARM has large market shares in many markets:
  - ***https://group.softbank/system/files/pdf/ir/financials/annual_reports/annual-report_fy2020_01_en.pdf***



- Example: “**Apple Sells Macs With Its Own [ARM] Chips**”
  - **Apple's 'M2' Next-Gen Mac Chip Enters Mass Production, Expected to Debut in Redesigned MacBooks Later This Year** April 27, 2021
- Microsoft SurfacePro X uses Microsoft’s own ARM-based processor (Fall 2019)

Pros and Cons of studying ARM

- ☺ A large, large portion of jobs in software design are on ARM
- ☺ A large, large portion of jobs in embedded systems are on ARM
- ☺ Most jobs in Digital Electronics design involve dealing with an ARM core

☺ There are surely more types of ARM Cores than people in this (virutal ?) room now

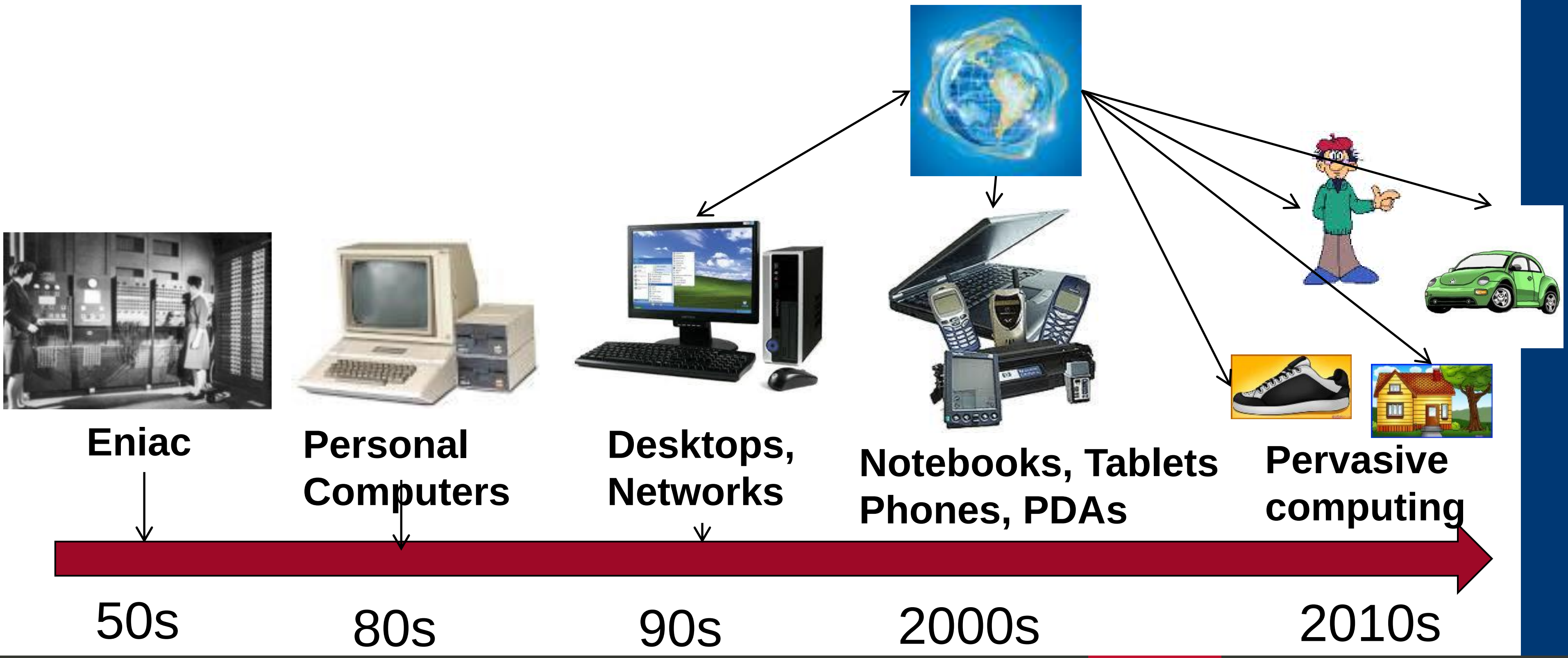
- ☹ ***For a variety of reasons, ARM has many exceptions and differences with respect to a standard processor architecture. So it is NOT NECESSARILY the best starting point if simplicity was the goal***
  - ☹ We will have to learn a little bit of computer architecture theory before we have the basis to compare ARM with the alternatives.
  - ☺ If you put in effort, you'll do fine.

- ☺ Note: in this course we will work with 32-bit ARM, not Thumb or 64-bit.

What is a Computer?

Computer: *device that can be* ***programmed*** *to carry out a set of arithmetical / logical operations.*

Since the sequence of operations can be ***"readily"*** *changed, the computer can solve more than one kind of problem.*



Eniac

Personal Computers

Desktops, Networks

Notebooks, Tablets Phones, PDAs

Pervasive computing

50s

80s

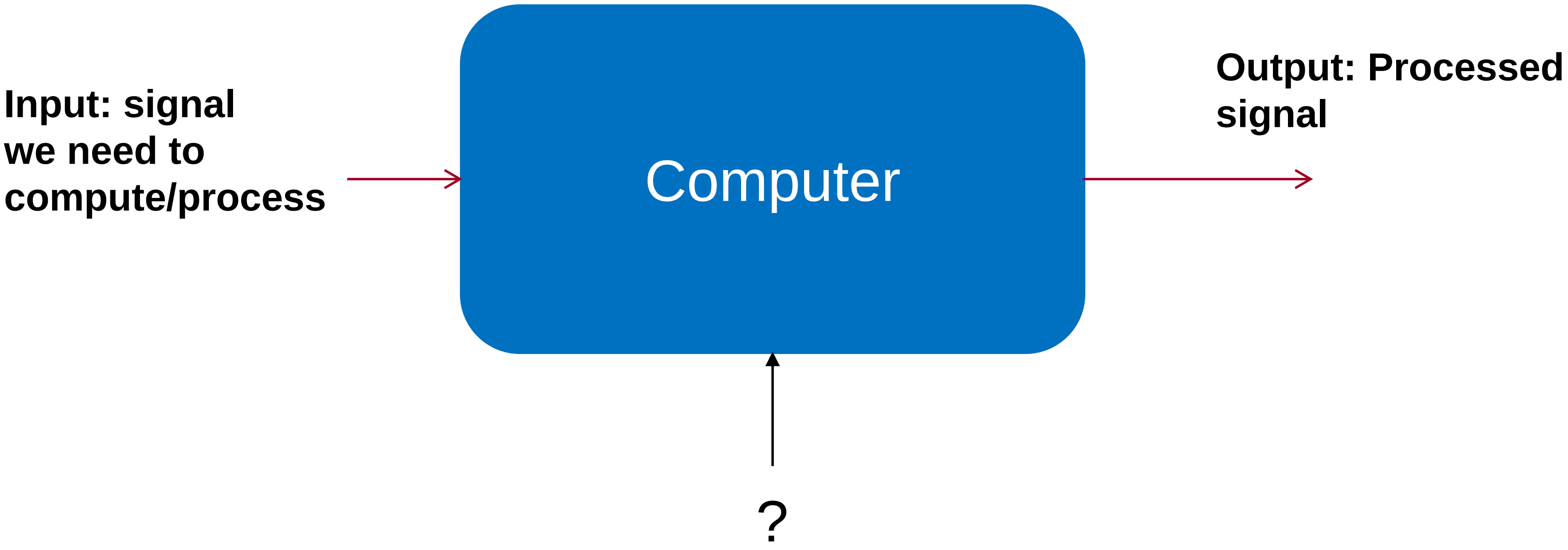
90s

2000s

2010s

What is a computer ?

For users, a computer is a device augmenting the quality of human life. For developers, a computer builds on a set of operations computed by a physical processor, possibly fast & portable



Input: signal we need to compute/process

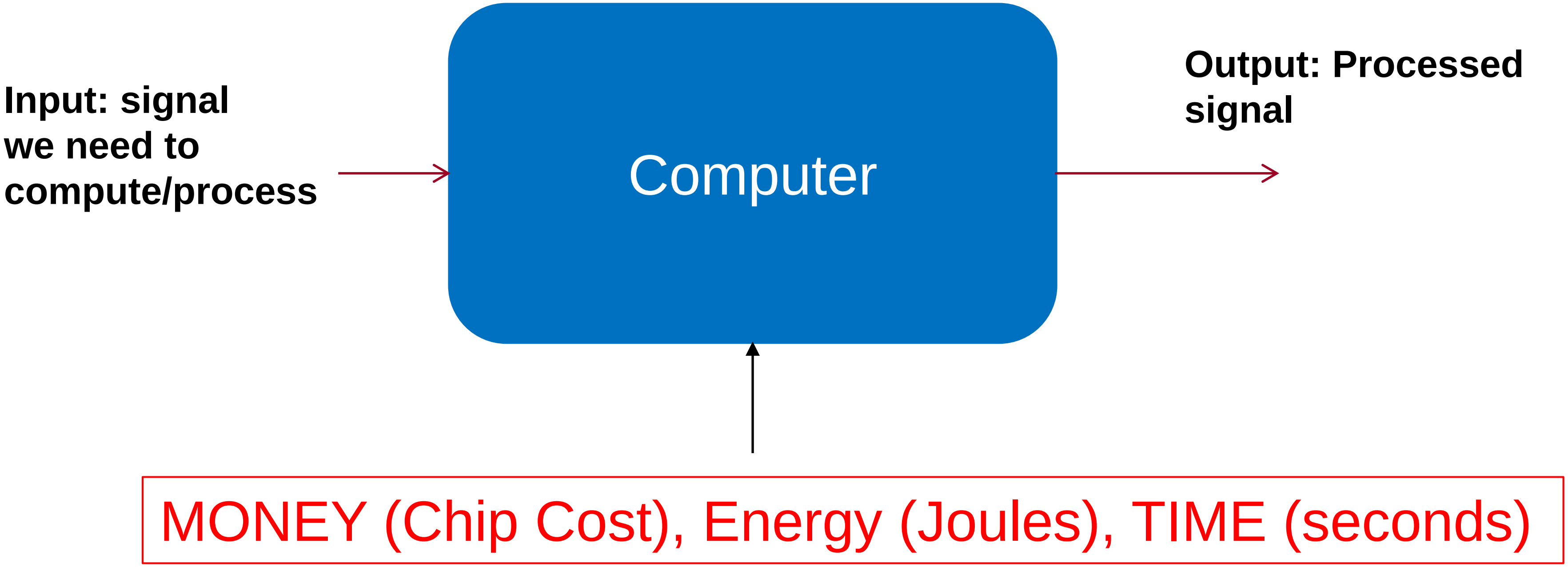
Computer

Output: Processed signal

?

What is a computer ?

For users, a computer is a device augmenting the quality of human life. For developers, a computer builds on a set of operations computed by a physical processor, possibly fast & portable



Input: signal we need to compute/process

Computer

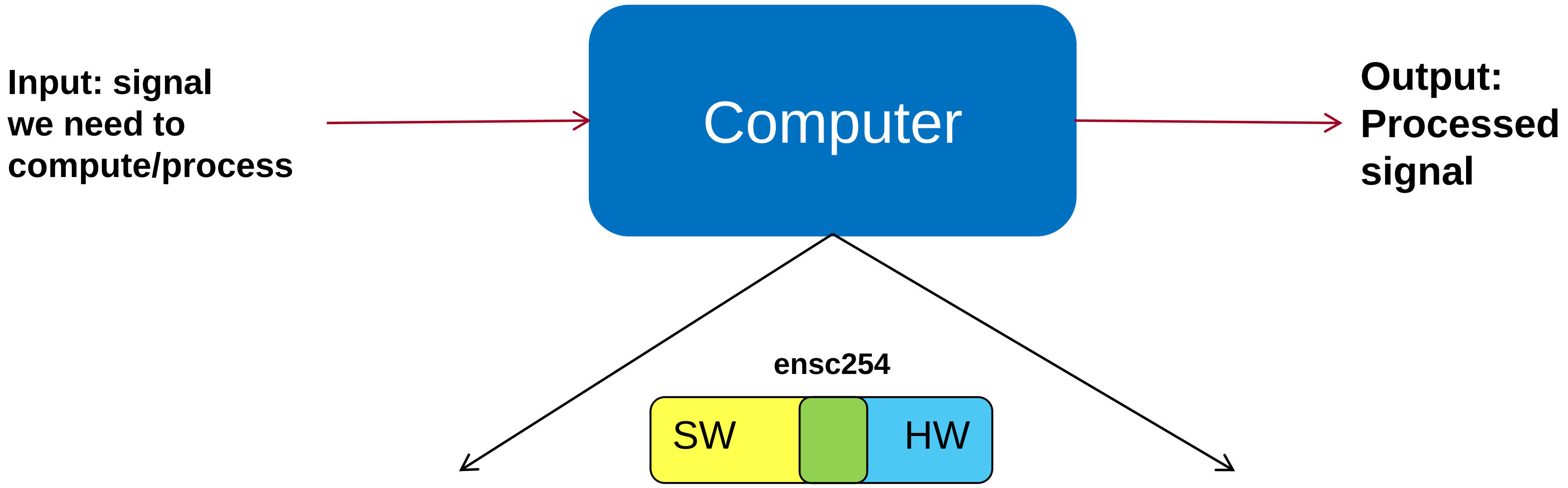
Output: Processed signal

MONEY (Chip Cost), Energy (Joules), TIME (seconds)

Computer Engineering vs Computer Science

- A Computer Engineer is very AWARE of the processor's hardware, he may even want to change it
- A Computer Scientist wants to VIRTUALIZE it and hide it

A Computer Engineer has ***important Cost, Time, and Energy Constraints*** that the computer scientist does not worry about so much



Input: signal we need to compute/process

Computer

Output: Processed signal

ensc254

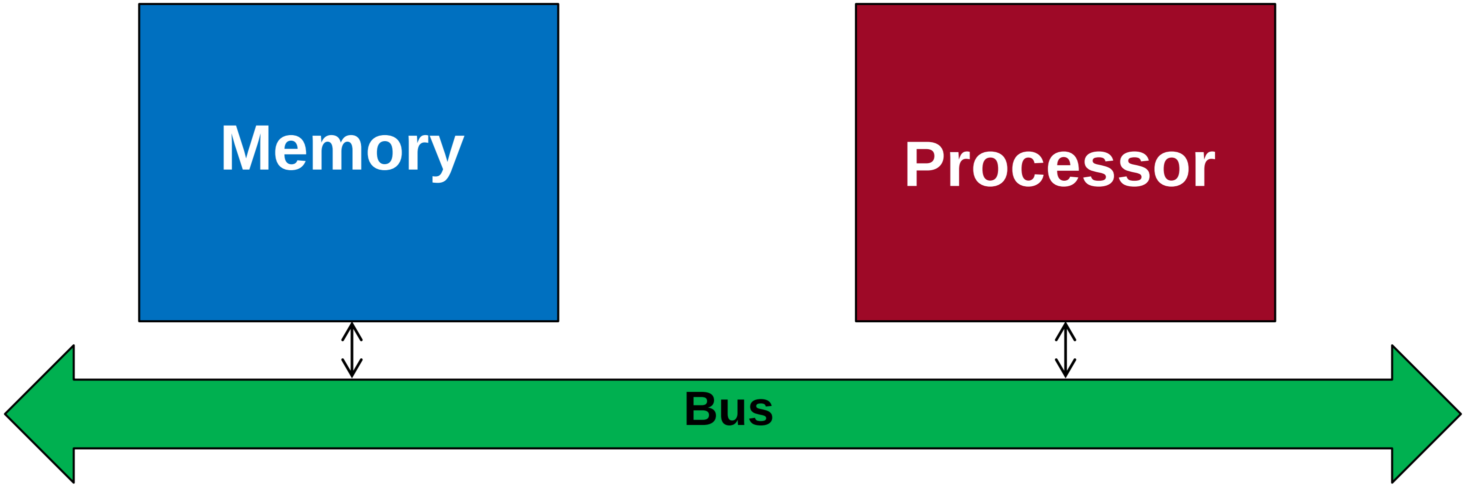
Computer Architecture

- **computer architecture** is a set of disciplines that describes a computer system by specifying its parts and their relations

- ***A computer is a device containing electrical switches (silicon Transistors) that can combine to provide Arithmetical/Logical computations***
- *Computer Architecture* is the way that such Arithmetical / Logical operations are organized in order to be as effective as possible

- ***A computer does not have to be necessarily programmable. Actually, if it is not programmable it may be more effective.***
- ***But it is so hard and costly to build a computer, that we want to make it programmable to reuse it in as many contexts as possile***
- For this reason, here we focus on programmable computers, and the best way to program them

Computer Architecture



- Since we want a computer to be programmable, in a simple form a computer will be composed of
  - A Memory, storing:
    1. Data to be Computed
    2. Instructions on how to compute them
  - A block performing computation – Processor (or historically Central Processing Unit – CPU)
  - The connection between Processor and Memory is called a Bus

Computer Architecture

- The ***Architecture*** of a given ***Computer*** is determined by
  - The ISA (Instruction Set Architecture)
  - The Microarchitecture
  - The System Design

ISA: Instruction Set Architecture

- The ***Architecture*** of a given ***Computer*** is determined by
  - The ISA (Instruction Set Architecture)
  - The Microarchitecture
  - The System Design

The ISA describes **which** operations, including arithmetical / logical operations, the computer is capable of computing.

The ISA includes, for each instruction, the operation name, operands, and the way the instruction is packaged in Binary format so it can be decoded by the Processor

MicroArchitecture

- The ***Architecture*** of a given ***Computer*** is determined by
  - The ISA (Instruction Set Architecture)
  - The Microarchitecture
  - The System Design

The Microarchitecture of a computer describes the way the ISA is realized. *Each ISA can be realized in many different ways depending on how the electrical circuits are organized* to perform the desired instructions. The microarchitecture is mostly focused on the Processor structure. That will be discussed in ensc450.

Microcode is a layer of internal hardware-level commands that implement one higher-level instruction, realizing internal state machine sequencing in some processors.

System Design

- The ***Architecture*** of a given ***Computer*** is determined by
  - The ISA (Instruction Set Architecture)
  - The Microarchitecture
  - The System Design

System Design includes all of the other hardware components within a computing system external from the Processor. These include:

- Data paths, such as buses and switches
- Memory, Cache and Memory controllers
- External Data processing or data transfer devices such as interrupt control, network interfaces, Direct Memory Access (DMA), etc.
- Low level software (Firmware) .

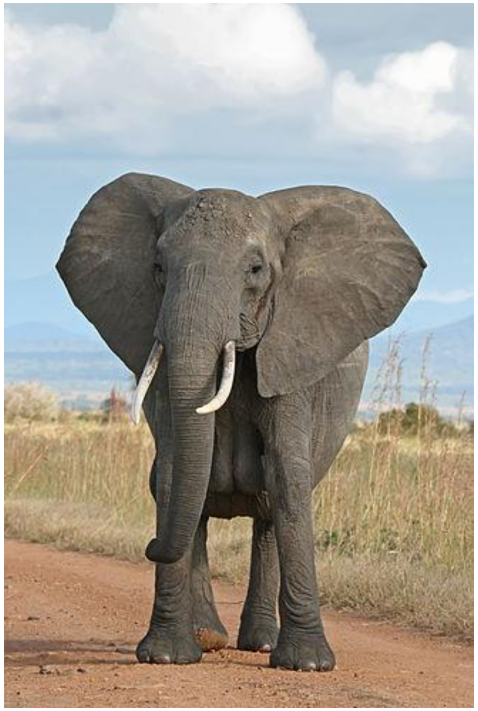
Firmware

- In electronic systems, ***firmware*** is the set of computer instructions and data that reside as read-only software on a given device. As a result, firmware usually cannot be modified during normal operation of the device. Typical examples of devices containing firmware are *embedded systems, computers, computer peripherals, mobile phones, and digital cameras*, etc. The firmware contained in these devices provides the lowest-level control programs for the devices.

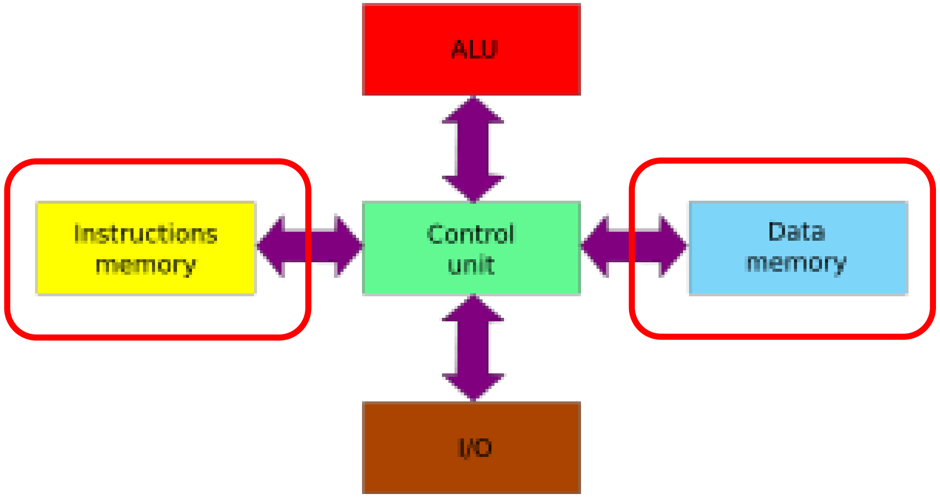
- The terminology "FIRM" means, as opposed to *hard*ware and *soft*ware, that firmware is not going to be changed except in case of explicit upgrades (or downgrades) following a special procedure.

- *An* ***EMBEDDED SYSTEM*** is a computer system that is intrinsically part of a larger physical system. EMBEDDED SYSTEMS represent an increasing portion of the computer market today

Memory / CPU Interface: Harvard Model



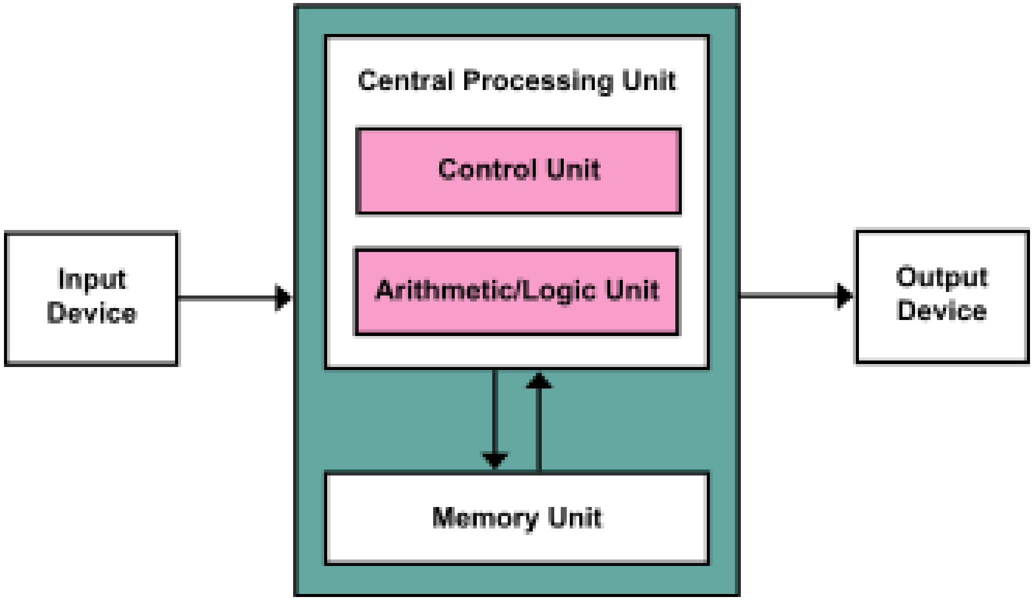
Source:Wikipedia



- The Harvard architecture is based on concurrent access to Instruction and Data Memory by means of separate and independent channels
- One of the first programs to run on the Harvard Mark I was initiated on 29 March 1944[2] by John von Neumann.

Memory / CPU Interface: Von Neumann Model

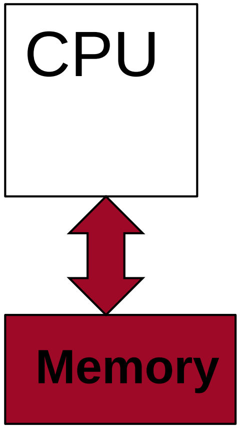
Source:Wikipedia



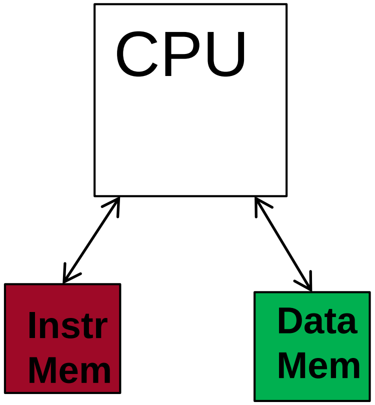
- Connection between Processor (CPU) and memory often has been based on the von Neumann model (John von Neumann, Princeton, 1945)
- One Memory Unit holds both instruction and data for computation
- When CPU performance is an issue, the *von Neumann* model may result in a bottleneck between Data and Instruction access

Wake-up Call: Determine which of the following is a Harvard Machine

A)

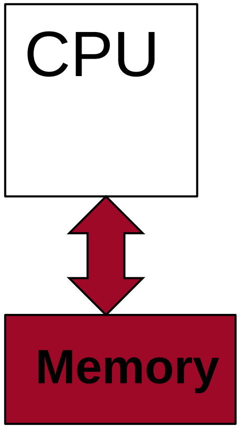


B)

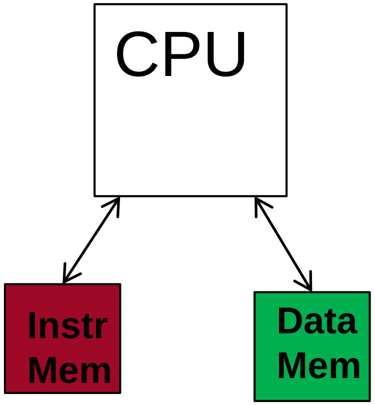


Wake-up Call: Determine which of the following is a Harvard Machine

A)



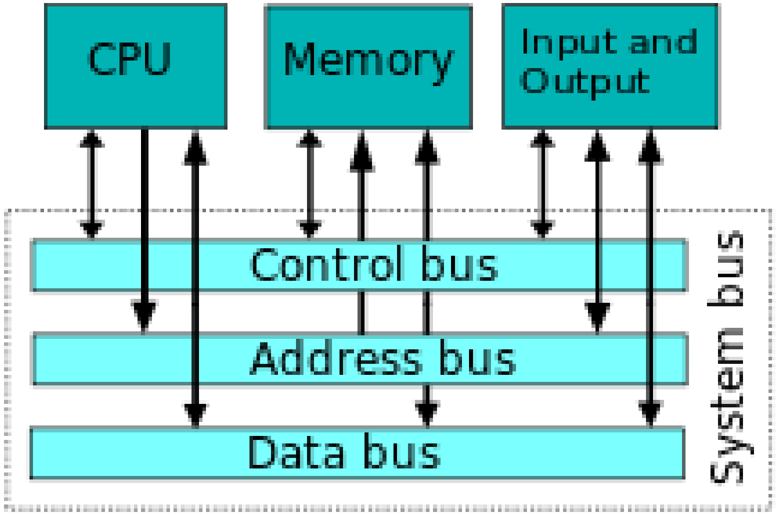
B)



b) A harvard architecture is a CPU
That has concurrent and separate access to
Data and instruction memory

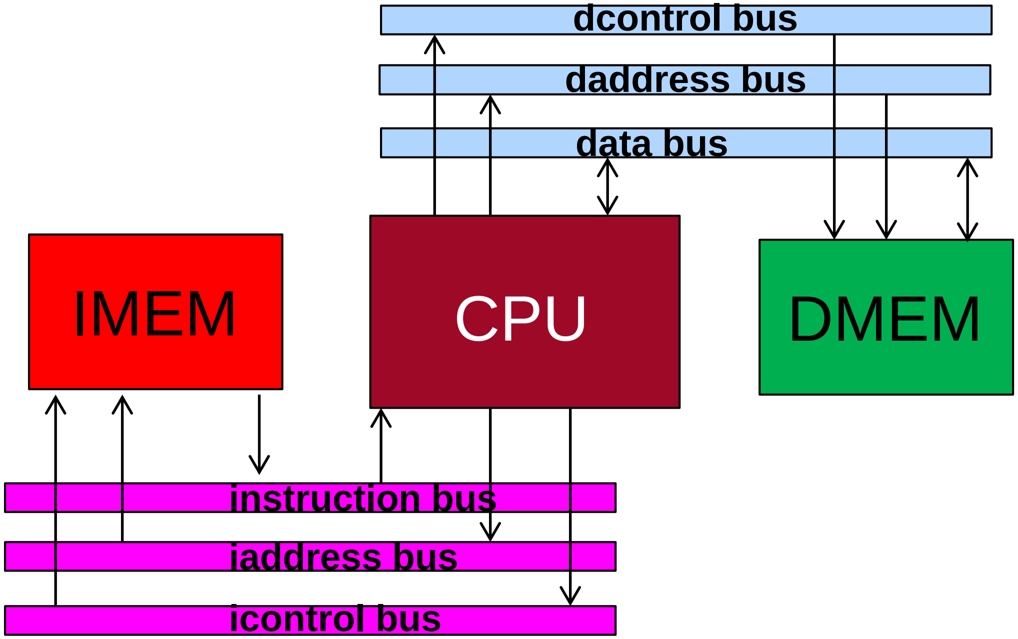
Bus Architecture

Source:Wikipedia



- A Memory Bus is typically organized with the following components:
  - *Address Bus*, generated by the CPU, contains the address of the memory location that needs to be Read/Written
  - *Control Bus*, generated by the CPU, contains the operation that needs to be performed (Read or Write)
  - *Data Bus*, Bidirectional, may be wrtten by the CPU in case of memory write operation or by Memory/IO in case of memory read

Harvard Bus Architecture



Harvard architecture has dedicated Instruction and Data buses

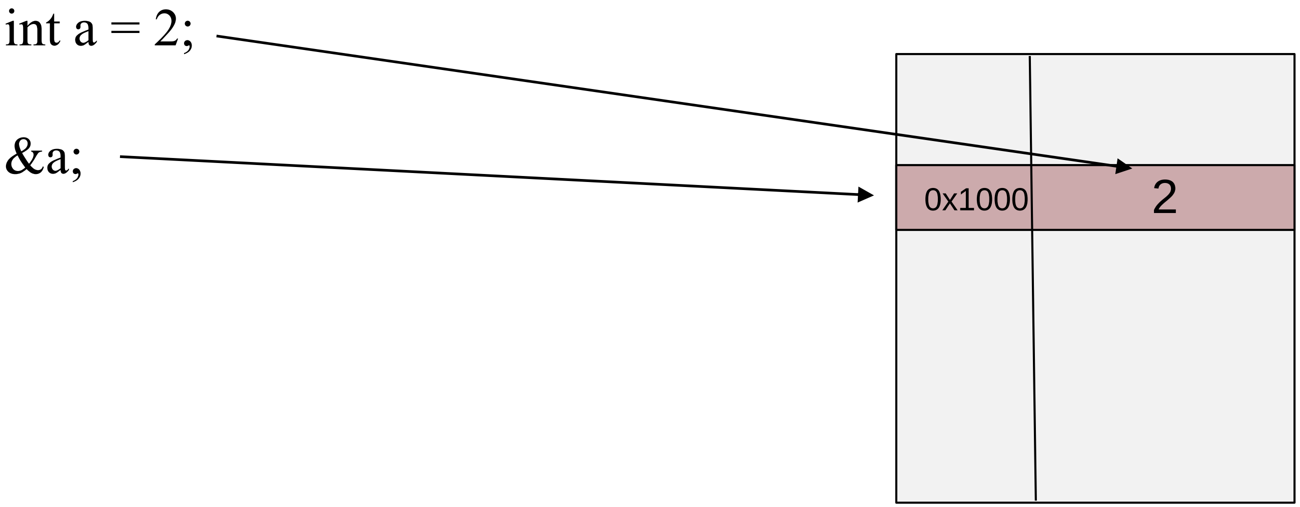
Memory Hierarchy

- **The von Neumann Processor sees memory as a single, whole, unique entity (A set of numeric Addresses).**
- **Remember: an address is a numeric location**
- But a physical memory is not necessarily single: ***memory can be defined as wherever a computer reads and/or stores data***
  - Large memories can contain lots of data / instructions but can be very slow, so such memories can require the Processor to wait a long time every time it fetches Data or an instruction
  - Small memories can be as fast as the Processor, but do not hold enough instruction/data for many computations
  - Peripherals can implement communication to external systems that are part of the overall «memory» concept

Memory is organized in a hierarchy of different devices with different degrees of size and access time.

Memory Hierarchy (2)

- **The Processor sees memory as a single, whole, unique entity (A set of numeric Addresses).**
- **Remember: an address is a numeric location**

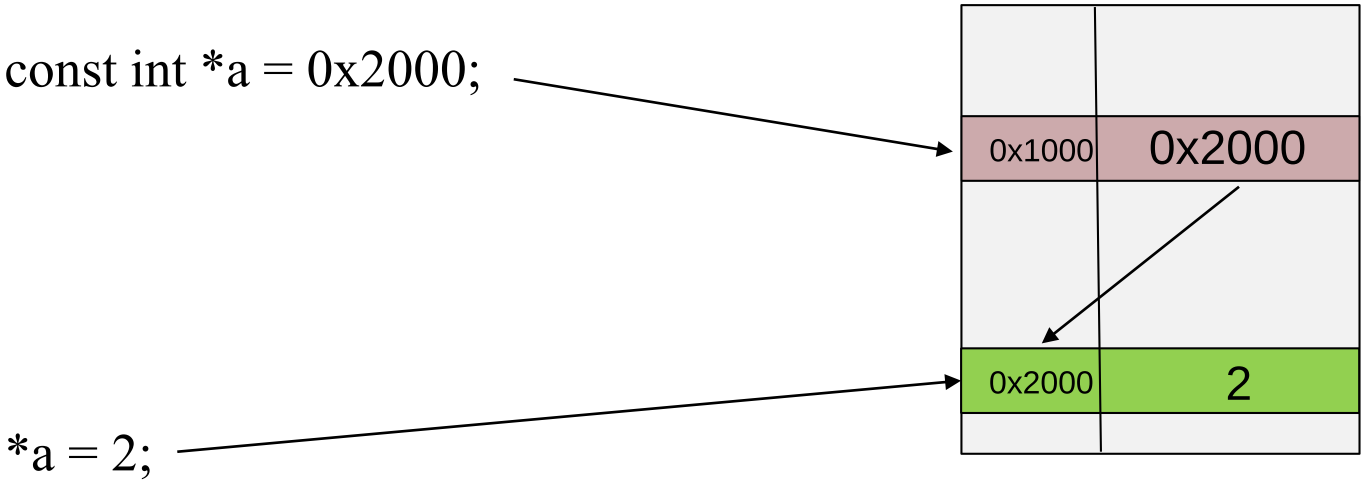

int a = 2;

&a;

THE PROCESSOR HAS NO NOTION OF HOW THE LOCATION IS ULTIMATELY IMPLEMENTED PHYSICALLY

Memory Hierarchy (2)

- **Each location in the ADDRESSING SPACE may be in a different physical device, with different access times and strategy. The ADDRESSING SPACE is a logical abstraction of the world surrounding the Processor.**



```
const int *a = 0x2000;
```

```
*a = 2;
```

0x1000	0x2000
0x2000	2

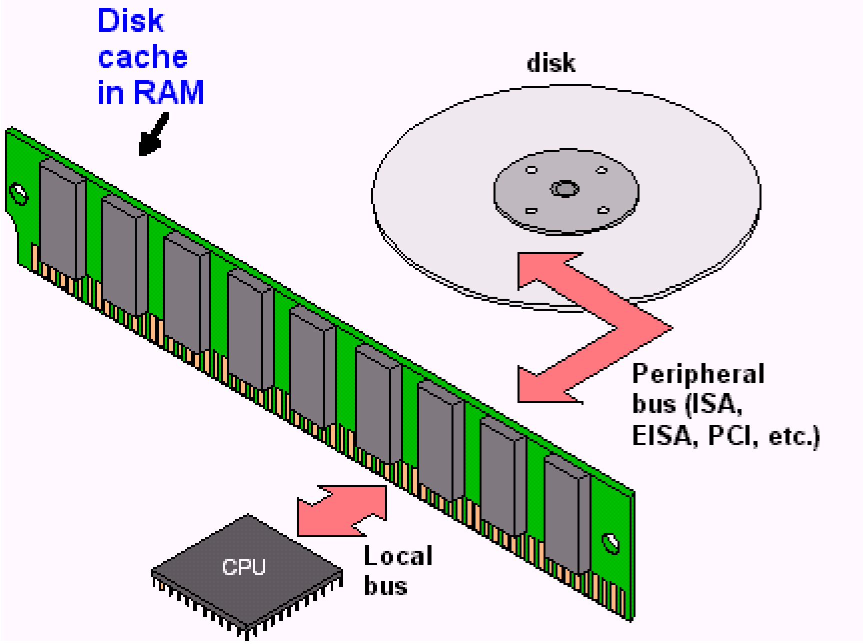
The addressing space can refer to many different physical devices:

- Onchip Memory (Flash, EEPROM, SRAM)
- Off-Chip Memory (SRAM, DRAM, Flash)
- Peripherals (UART, USB, Ethernet, I2C, CAN, …..)
- In case of Operating Systems, they can implement a VIRTUAL memory system (we'll discuss it later)

Memory Hierarchy (3)

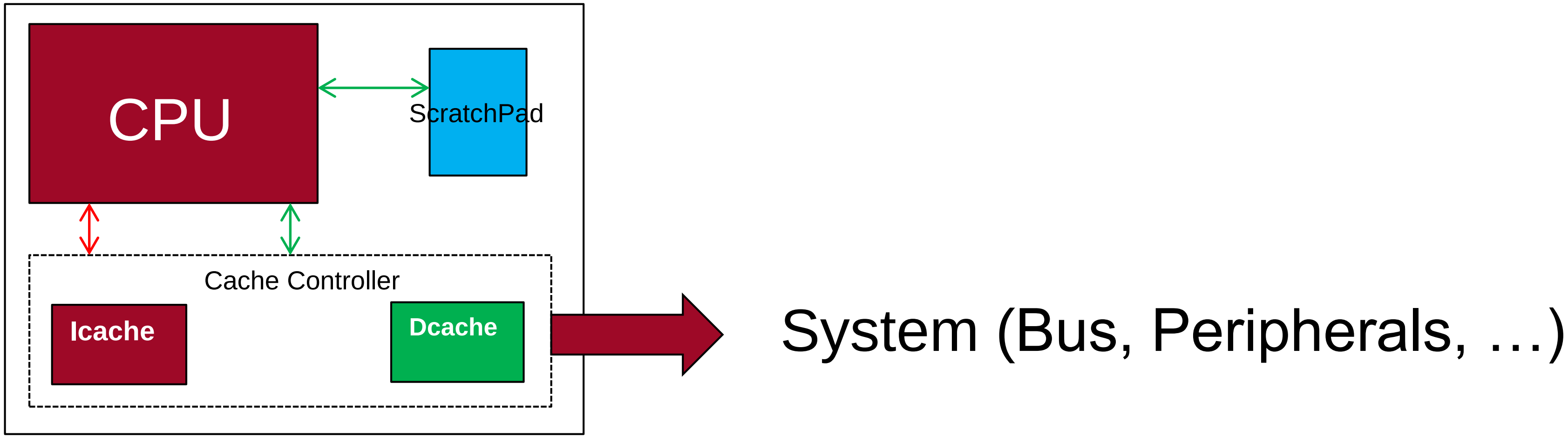
- Memory is organized in a hierarchy of different devices with different degrees of size and access time
- Examples:
  - Small, fast memories are on the same IC as the processor
  - Large memory blocks can be on a different IC or even different board (DRAM)
  - Non-Volatile memory can be on a different IC or even on magnetic media

From Computer Desktop Encyclopedia
© 1999 The Computer Language Co. Inc.



THIS HIERARCHY IS NOT NECESSARILY VISIBLE TO THE PROCESSOR, which only cares about FLAT ADDRESSES

Cache and Scratchpad memories



- Often, the CPU needs a small local memory where it can perform critical computation. If this memory is controlled by the CPU itself, it is usually defined ***SCRATCHPAD***
- Filling the scratchpad is a waste of time: we can build a system that *automatically* stores -- locally to the CPU -- data and instructions that are considered relevant for the CPU at a certain time. In this case, the local memory is called a ***CACHE***
- ***For modified Harvard architectures, it is quite customary to have an Instruction Cache and a Data Cache coupled to the CPU. The cache controller is the logic managing transfers between caches and the rest of the system. Sizes of caches vary greatly, but in the range 256K to 8M bytes has recently been common.***

Computer Architecture: RISC vs CISC

Reduced instruction set computer, or **RISC**, is a Processor design based on the insight that simplified instructions can provide higher performance if this simplicity enables much faster execution of each instruction. Programs require a significantly higher number of instructions to complete a task, but Processor architecture is simpler and more efficient

Complex instruction set computer, or CISC is a Processor design where a single instruction executes several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) and/or is capable of multi-step operations or addressing modes.

Tradeoff in RISC: Making operations simpler, you increase instruction processing rate to compensate for the growth in instruction count

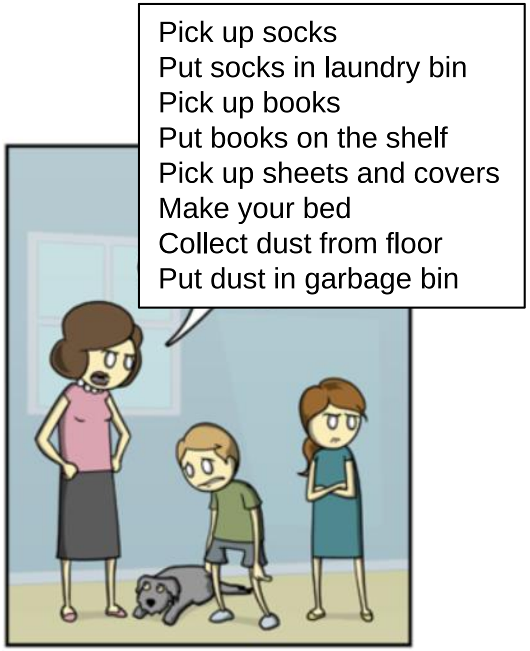
Silly Example: RISC vs CISC

Go clean up your room

CISC Instruction:

Short program

Assumes powerful decoding and computation capabilities by the processor



RISC Instruction:

Long program

Requires little decode and processing capability. Requires much more instruction memory, but processing machine can compute instructions faster

Wake up call

What would you think can execute instructions faster in principle ?

[✓ A.] A RISC processor
B. A CISC processor

What would you think can consume the smaller energy per instruction?

[✓ A.] A RISC processor
B. A CISC processor

This one’s harder: Memory attached to what would you think can consume the higher power?

[✓ A.] A RISC processor
B. A CISC processor

Wake up call

What would you think can execute instructions faster in principle ?

A. A RISC processor
B. A CISC processor

What would you think can consume the smaller energy per instruction?

A. A RISC processor
B. A CISC processor

This one's harder: Memory attached to what would you think can consume the higher power?

A. A RISC processor
B. A CISC processor

“Classic” RISC features

Differentiation between CISC and RISC is not formal, but RISC machines have several “classic” features:

1. One instruction is computed per cycle (Beware of the pipeline concept => will see it in a later part of the course)
2. All instructions have the same size and a more-fixed format
3. “Reduced” use of microcode
4. **Memory Data is explicitly accessed with specific *Load / Store* instructions**. All other operations use only registers in a register file

Feature Creep

- ***(From Wikipedia) Feature creep*** *is the expansion or addition of new features in a product during its lifetime. Extra features go beyond the basic function of the product and so can result in over-complication rather than simple design. Viewed over a longer time period, extra or unnecessary features seem to creep into the system, beyond the initial goals.*
  - Academics, and students, like precise definitions and clear rules
  - Industry likes that too when they advertise, but then they face the harsh reality of life, and in a panic they are very inclined to dirty workarounds rather than rethinking concepts from scratch
  - The idea of “Reduced” instruction set has been “Workarounded” many many times in the last 30 years depending on the needs of manufacturers and their markets:
    - Example: ARM2 had 46 instructions, ARM9 has several hundred
- Still, most of the concepts outlined before hold, ***ESPECIALLY the LOAD/STORE paradigm*** that is the most typical feature of any RISC machine

Notable RISC Processors

UCBerkley RISC-I and RISC-II projects (1980-1985) -> DLX

Stanford University – MIPS (1984-1986)

SPARC (SUN MicroSystems) derived from RISC-II

1983 Acorn + VLSI built ARM1

Apple-IBM-Motorola (AIM) PowerPC in 91

Tensilica XTENSA in early 2000

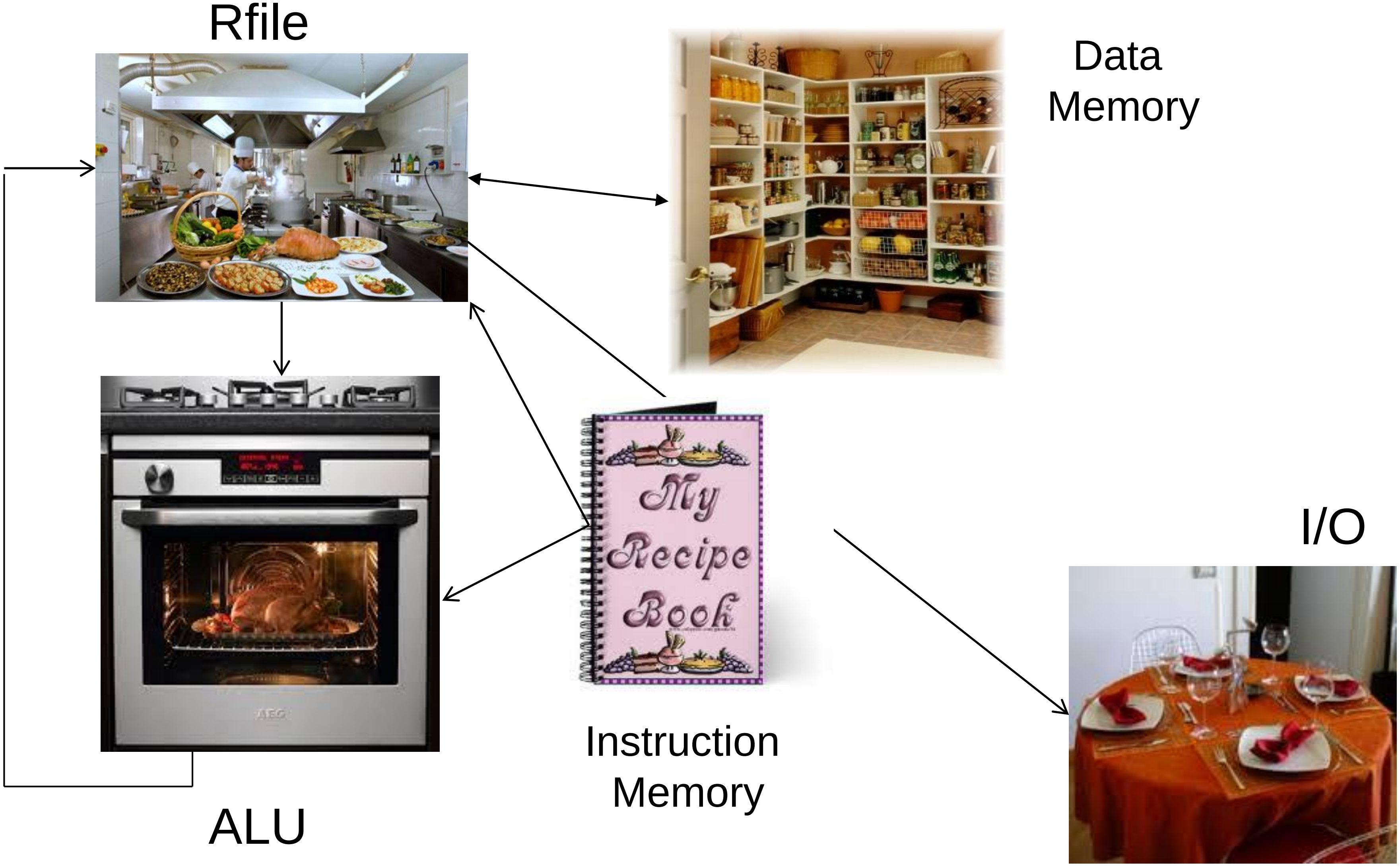
UCBerkley is currently running RISC-V in Berkley with focus on instruction encoding and Multi-Core processors

ARM – Little bit of history

- 1985 In UK Acorn was a Desktop computer manufacturer. They partnered with University of Manchester and VLSI technology (SF, Cal) to design Acorn Risc Machine (ARM1) : 25K transistors, 4 MHZ
- 1987 ARM2 had a MMU, 30K transistors and could run at 18MHz
- 1989 ARM3, 25MHZ
- 1989 ARM starts a partnership with Apple, that was looking for a low-power device for the pioneering Newton PDA. ARM ltd is established, with headquarters in Cambridge, UK
- *1990 ARM chooses to be a **FABLESS** company, licensing its architecture to vendors rather than buying silicon from them.* ***In this way, they avoided the huge cost increase imposed by technology evolution.***
- 1993 Release of the ARM7TDMI: *Thumb, Debug, Multiplier, In Circuit Emulation*
- 2001 Release of the APPLE i-Pod based on ARM7TDMI
- 2007 Release of APPLE i-Phone based on ARM11 at 620MHz
- 2010 Release of APPLE i-Pad based on ARM Cortex A8 at 1GHZ
- 2013 STMicroelectronics releases a version of the ARM Cortex A9 running at 3.2GHz
- 2021 Apple's 'M2' Next-Gen Mac Chip Enters Mass Production

In the meanwhile, several versions of the ARM architecture have been used in all sectors of digital electronics: Wireless communications, Automotive, tablets, Digital TVs, Networks, motor control, discrete microcontrollers

Processor Architecture in a NutShell



Rfile

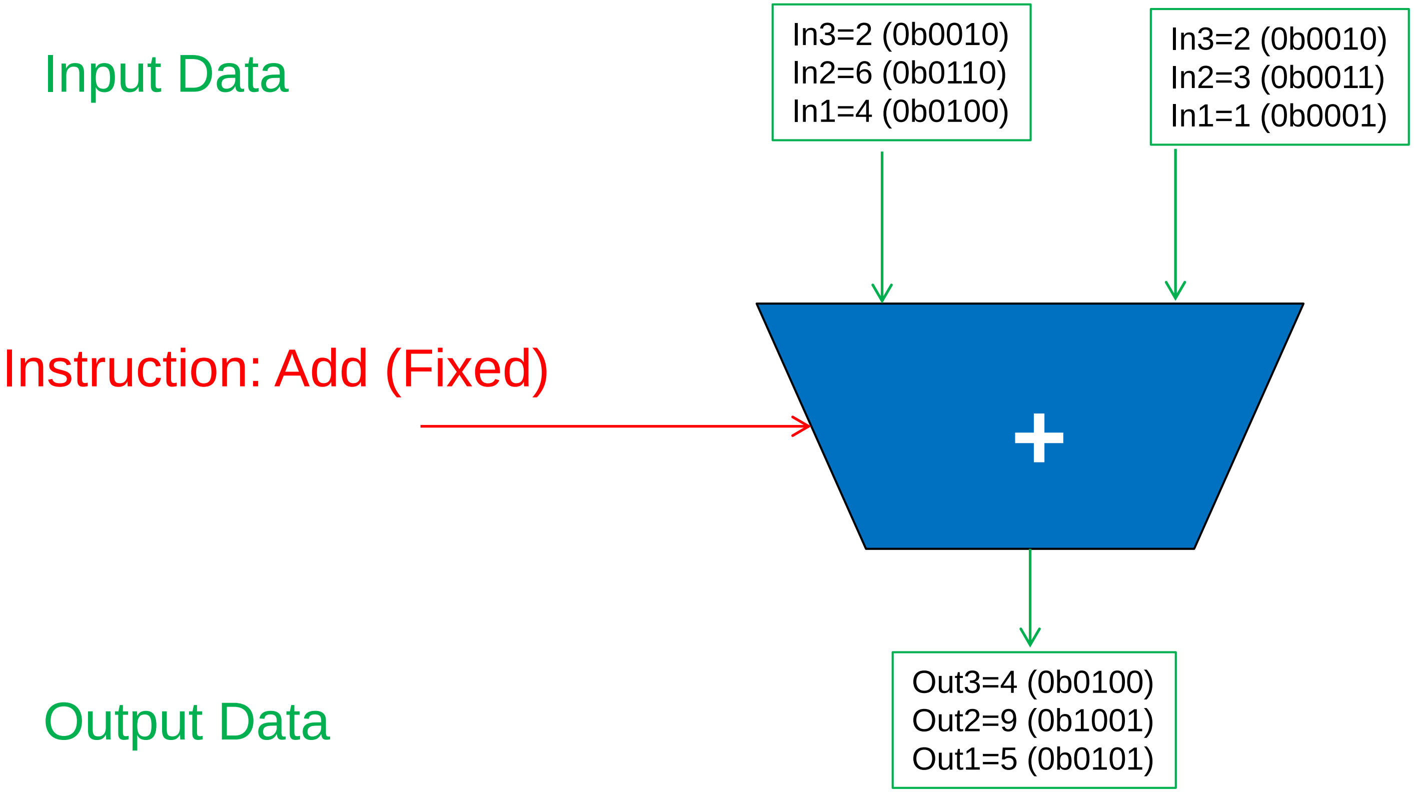
Data Memory

ALU

Instruction Memory

I/O

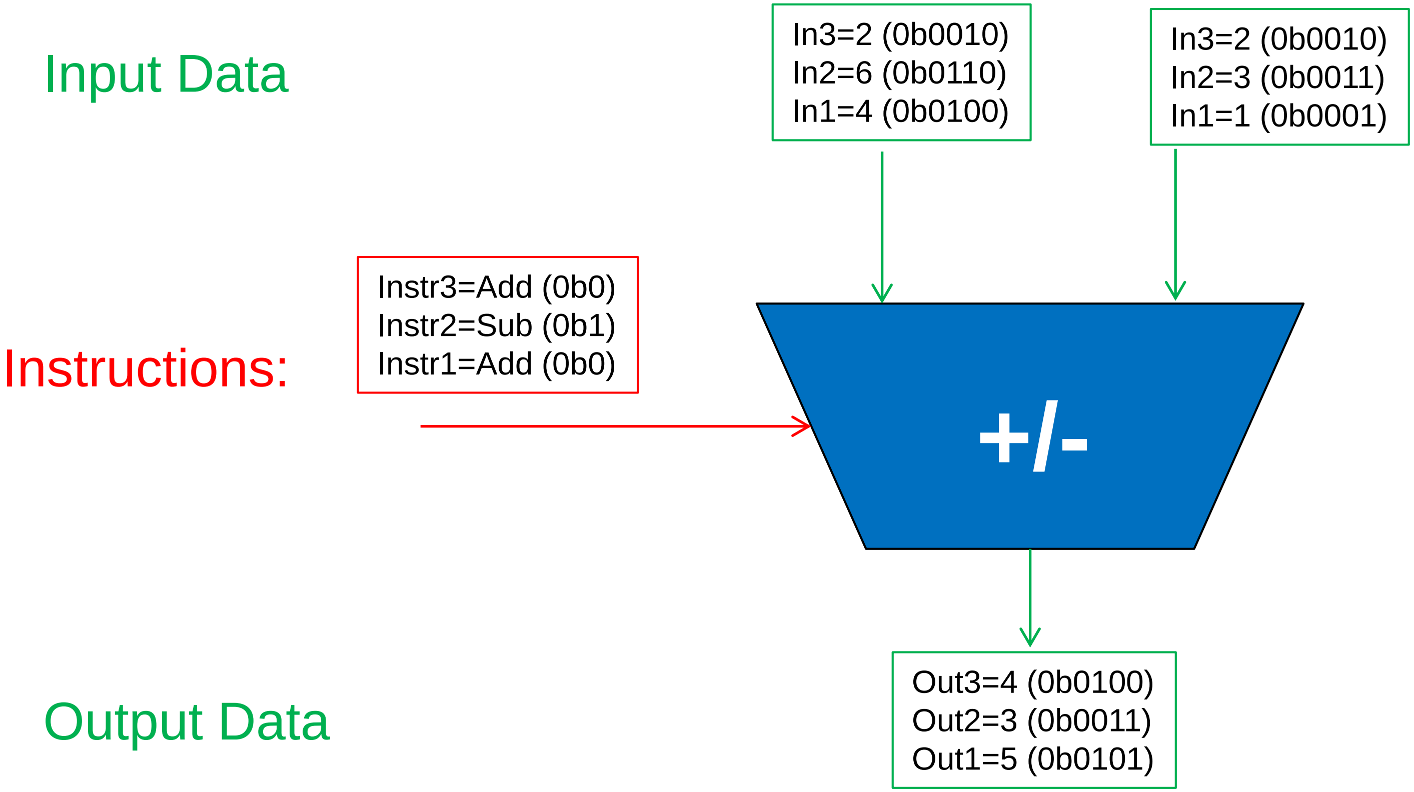
RISC CPU Architecture (1)



A CPU is a device performing operations on Data. In an Harvard architecture, Data and Instructions come from two different directions.

Our first basic example of Processing Unit is an Adder

RISC CPU Architecture (2)



Input Data

In3=2 (0b0010)
In2=6 (0b0110)
In1=4 (0b0100)

In3=2 (0b0010)
In2=3 (0b0011)
In1=1 (0b0001)

Instructions:

Instr3=Add (0b0)
Instr2=Sub (0b1)
Instr1=Add (0b0)

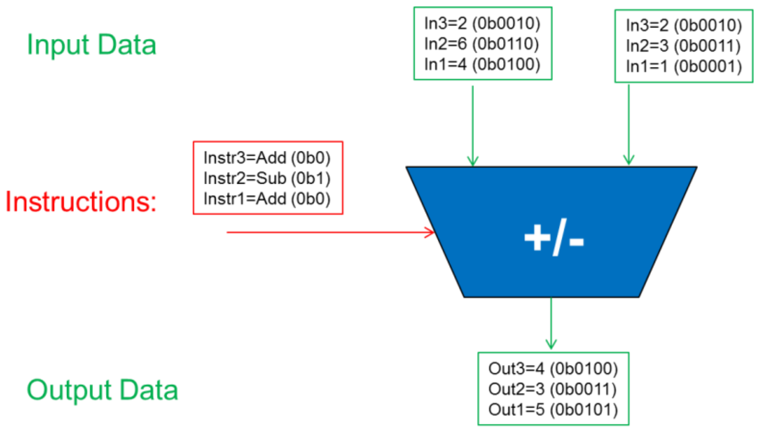
Output Data

Out3=4 (0b0100)
Out2=3 (0b0011)
Out1=5 (0b0101)

Since performing only additions can be a bit of a limitation, we extend the concept to support Addition AND Subtraction.

We use bits to express the desired computation. Our PU is now PROGRAMMABLE!

RISC CPU Architecture (3) : The ALU



Input Data

In3=2 (0b0010)
In2=6 (0b0110)
In1=4 (0b0100)

In3=2 (0b0010)
In2=3 (0b0011)
In1=1 (0b0001)

Instructions:

Instr3=Add (0b0)
Instr2=Sub (0b1)
Instr1=Add (0b0)

+/-

Output Data

Out3=4 (0b0100)
Out2=3 (0b0011)
Out1=5 (0b0101)

Op Type	Operation (Example)
Arithmetical	Addition, Addition Unsigned Subtraction, Subtraction unisgned
Logical	And, or, not, xor
Comparison	Equal, not equal, greater, lower, greater equal, lower equal
Shift	Shift left, shift right

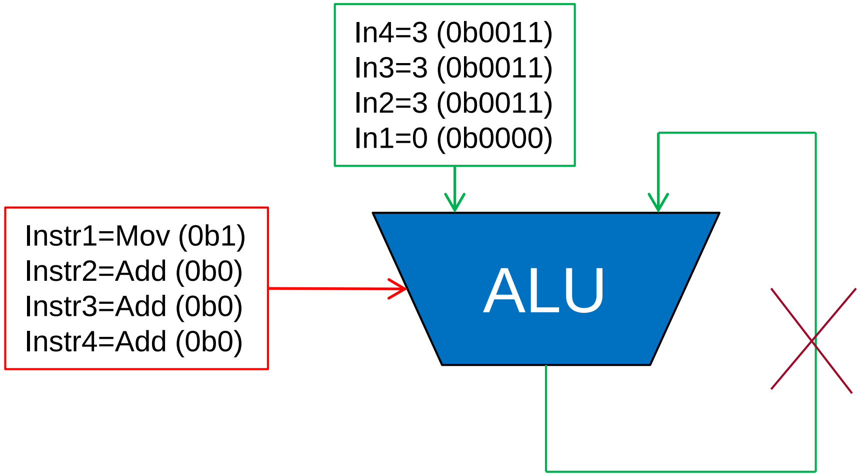
Programmable processing units in computers handle many operations: arithmetical, logical, comparison, and in some cases also shift operations

A programmable block that performs such operations is called ALU (Arithmetical / Logical Unit). It is the heart of every CPU, the part where “Computation” take place

The ALU is a strictly SERIAL device, that performs one operation at a time (better said PER CYCLE).

The hardware complexity of an ALU (hence power, timing and area) is strongly related to the number of bits of its inputs. Common CPU units today feature 32 or 64-bit ALUs.

RISC CPU Architecture (4) : Temporary Variables

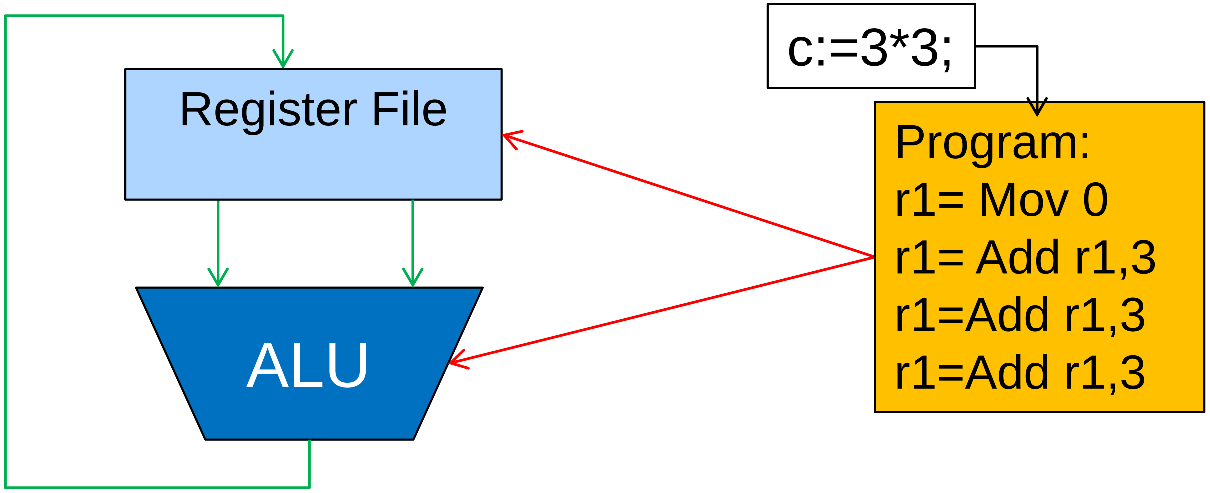


Program:
Res= Mov 0
Res= Add res,3
Res=Add res,3
Res=Add res,3

- Suppose that we want to use our brand new ALU to perform a simple multiplication: Out=3*3
- Most cores have a specific PU that supports multiplication, but for the moment let us suppose that is not possible. ***Could we run our computation iteratively with 4 consecutive ALU Operations, reusing our own outputs?***

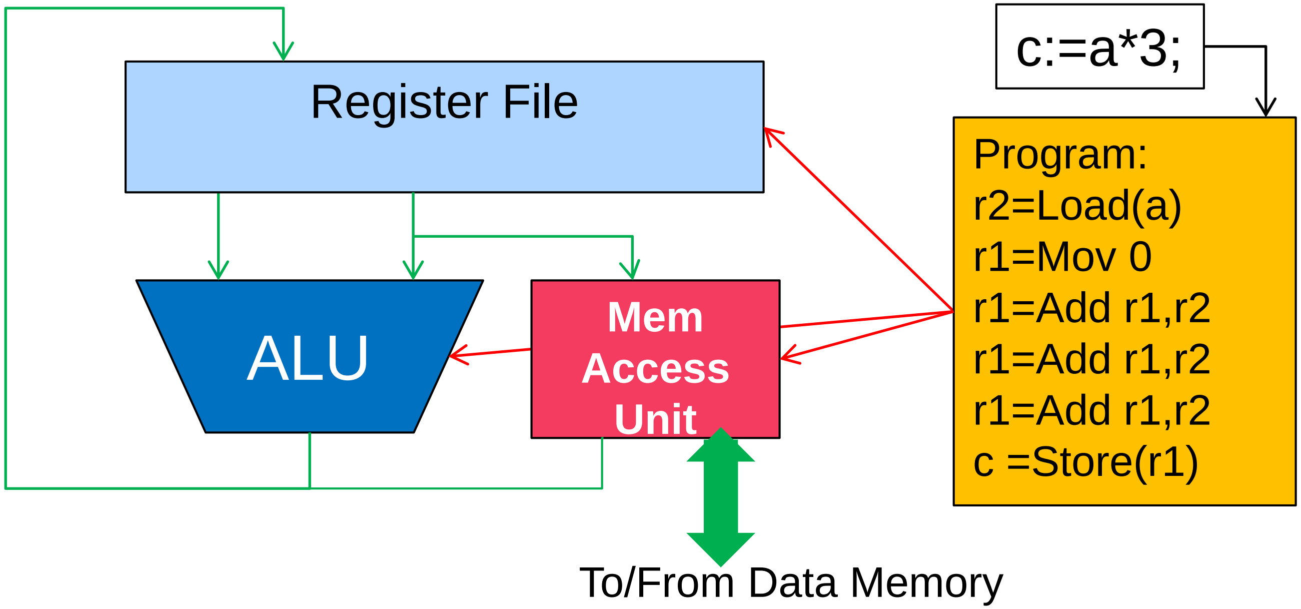
THIS IS NOT VIABLE! The ALU needs to compute ONE Single operation at a time (PER CYCLE). So we need some place to store RES, in order to reuse it in the following operation

RISC CPU Architecture (4) : Register File



- ***We call REGISTER FILE a small local memory***, with few locations tightly coupled to the ALU. The register file is used in all CPUs to store local values for quick usage by the ALU
    - Register files have ~ 8 to 128 registers, with same bit width as the ALU
- *Using RFILE as a source/destination for ALU operations is so efficient that ALL RISC ARCHITECTURES ARE LOAD/STORE: NO ALU OPERATION IS ALLOWED to go directly into memory,*
- **Only specific operations, LOAD and STORE, access memory and load data into the register file, where the data is accessible to the ALU**

RISC CPU Architecture (4) : Memory Access Unit

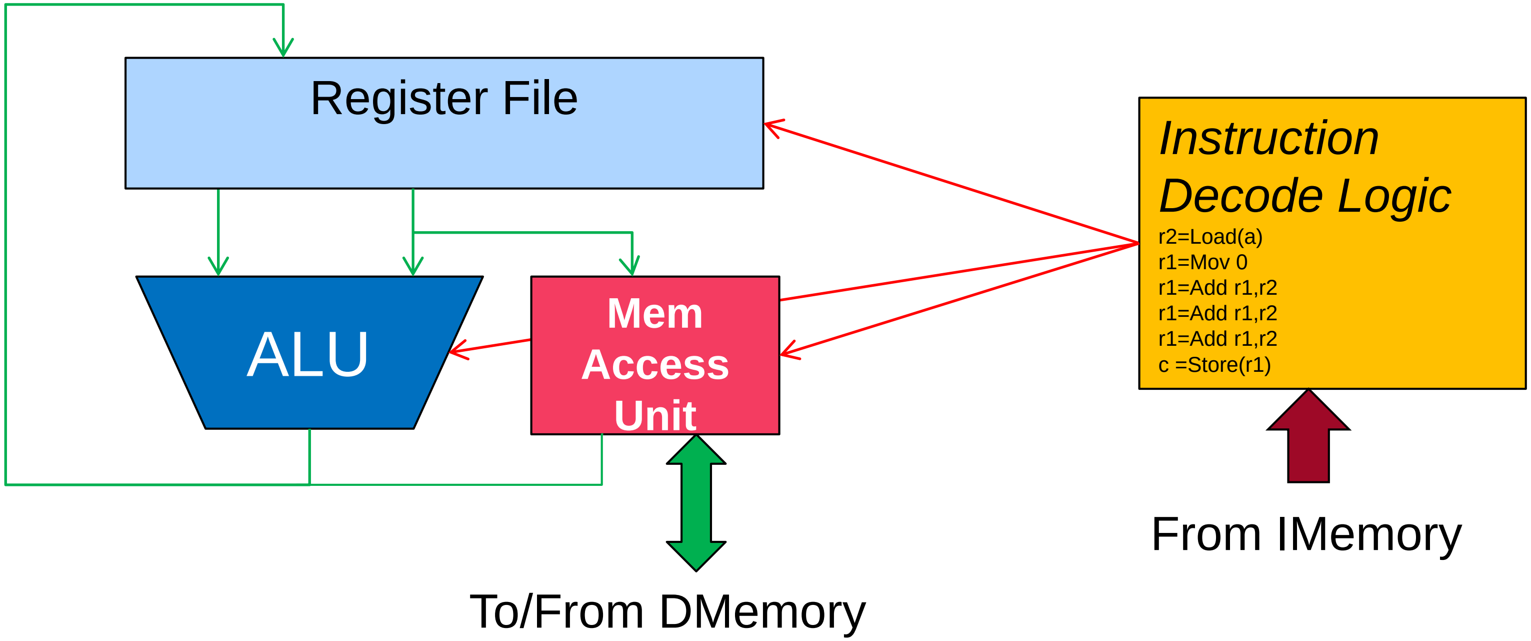


Program:
r2=Load(a)
r1=Mov 0
r1=Add r1,r2
r1=Add r1,r2
r1=Add r1,r2
c =Store(r1)

Op Type	Operation (Example)
load	Load word, Load half-word, Load byte
store	Store word, store half-word, store byte

- At the beginning of our computation (Reset), the register file will be empty.
- At the end of our computation, the result must be written in memory.
- ***The LOAD and STORE operations control a hardware Memory Access Unit that will take care of transferring Register File values TO / FROM the memory.***
- The CPU itself doesn’t want or need to know how memory is organized, if there are any Caches, storage peripherals, etc. For processors, memory is a hopefully large, homogeneous set of addresses where it can LOAD and STORE Data

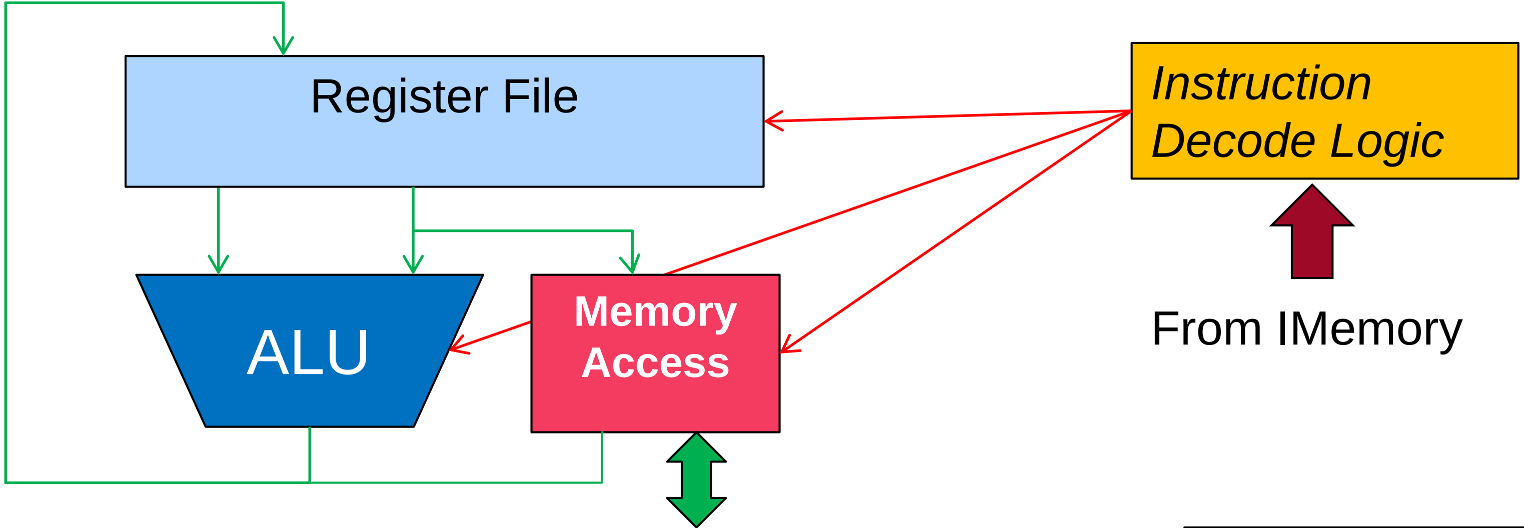
CPU Architecture (5) : Instruction Decode



r2=Load(a)
r1=Mov 0
r1=Add r1,r2
r1=Add r1,r2
r1=Add r1,r2
c =Store(r1)

We define *Instruction Decode Logic* the part of the CPU that reads binary words that represent the instructions, and produces control signals for the various parts of the architecture

CPU Architecture (6) : Program Flow control



To/From Dmemory
And peripherals

Program:

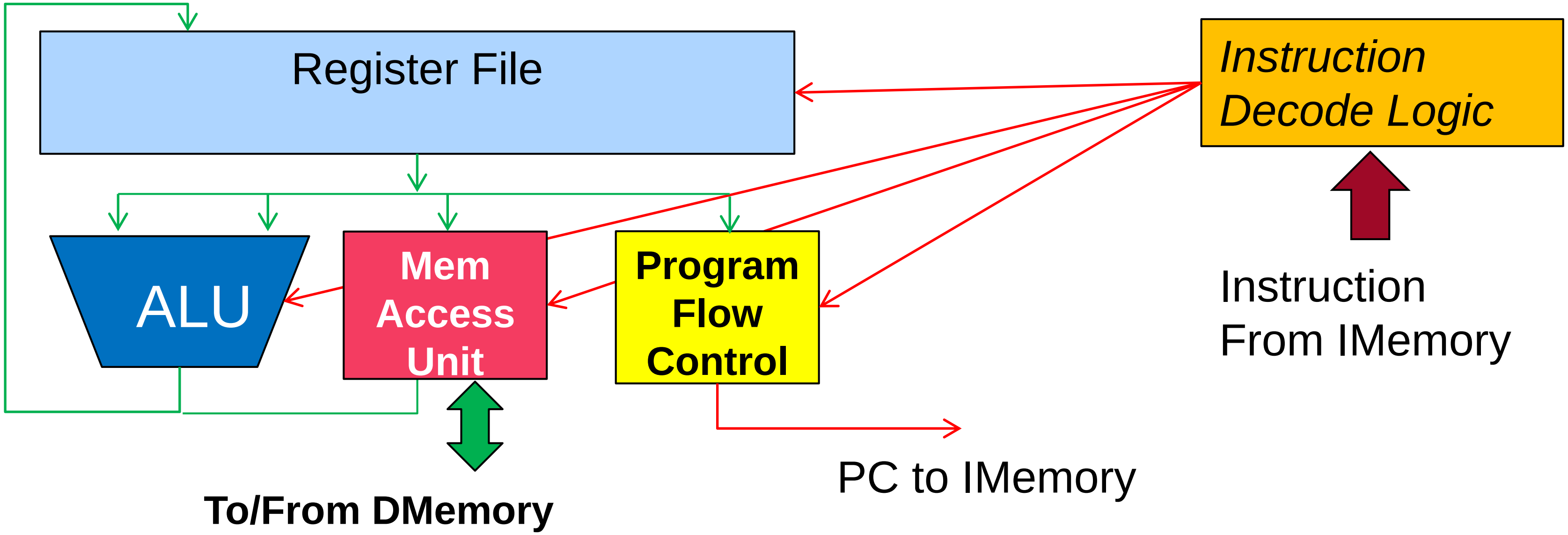
```
    r2=Load(a)
    r3=Load(b)
    r1=Mov 0
Loop:
    r1=Add r1,r2
    r3=Sub r3,1
    Bne Loop
    c =Store(r1)
```

So far we have assumed that the program could flow sequentially one instruction after the other. This is not always the case, due to function calls, loops (for/while), branches (if/switch)

As an example, let us suppose we want to perform a generic multiplication: then we will have to loop on the value of the second operand.

Bnz (Branch Not equal Zero) will force the program to jump back to the label loop until the last operation is equal to 0

CPU Architecture (7) : Program Flow control

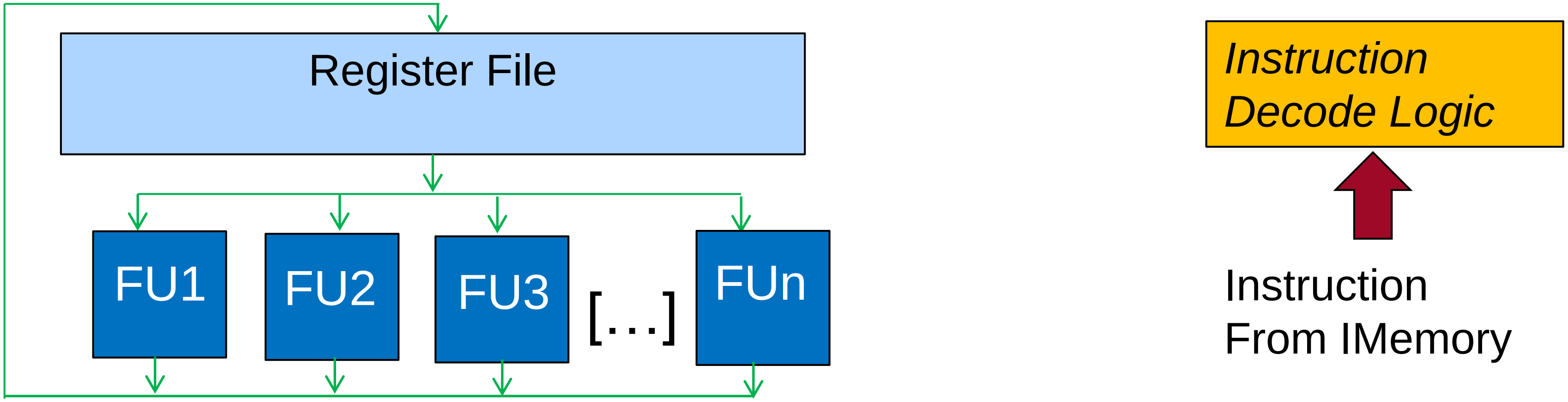


The program flow control logic is the hardware block that will manage control flow operations, that is operations that change the flow of the program and hence the way that the Instruction Memory is addressed.

We call ***PROGRAM COUNTER (PC)*** the address of the Instruction that is currently being read *(FETCHED)* from instruction memory.

The program flow control is dedicated to the calculation of the program counter of the next instruction

CPU Architecture (8) : Overall view



In general, we can represent any Processing unit based on the RISC concept as a system with a Register File holding temporary data, an Instruction Decoding logic translating the binary encoding of the ISA, plus a set of ***function units*** (for example ALU, SHIFTER, MULTIPLIER, FLOATING POINT COPROCESSOR, ….). Between those units, there would be a Dmem handling unit accessing Data memory for load/stores, and an Imem handling unit computing current Program Counter for Instruction Memory access.

Usually, CPUs are strictly sequential, so one instruction at a time (per cycle) is fetched in the architecture. CPUs handling more than one instruction at a time are called Superscalar.