FIT1047 - Week 3

Central Processing Units

This week is about the "brains" of digital computers, the *Central Processing Units*. We will introduce the basic *language* that they understand (called *machine code*), and construct an entire CPU from basic logic gates.

Recap

In weeks 1 and 2 we saw

- Number systems, binary
- Boolean logic
- Basic gates

Now let's put them together to build a computer.

Overview

- CPUs
- Machine code and assembly language
- Combinational Circuits
  - adding, multiplexing, decoding
- Arithmetic/Logic Units (ALUs)

CPUs

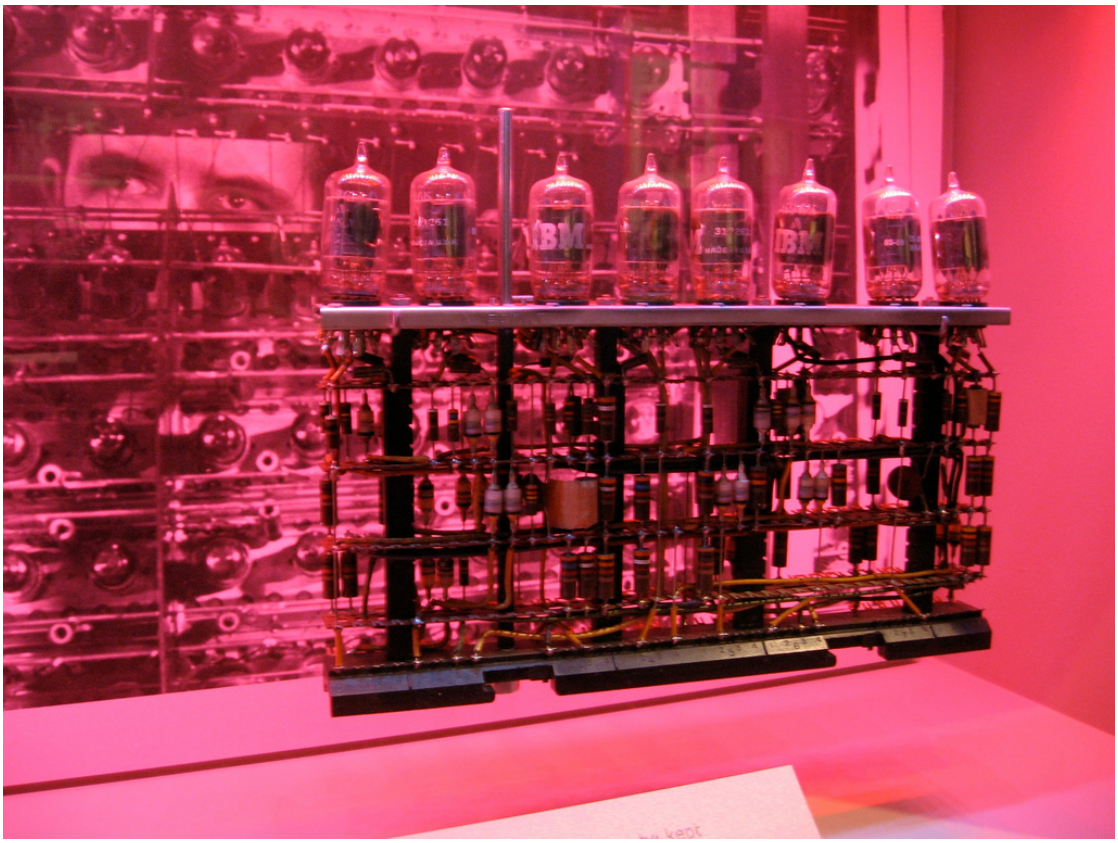
A *central processing unit* is the **"brain"** of a computer.

- built out of **logic gates**
- executes **instructions**
- connected to **memory** and **I/O devices**

Today we will first take a quick look at the *instructions* that a CPU executes, and then at some of the basic *circuits* out of which CPUs are built.

Some CPUs

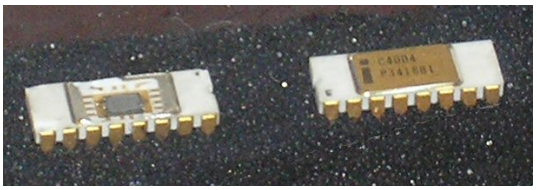
IBM 700 series vacuum tubes and resistors



Source: Wikipedia

These vacuum tubes, combined with resistors, can be used to construct simple logic gates such as NOR and NAND, from which all other gates can then be built.

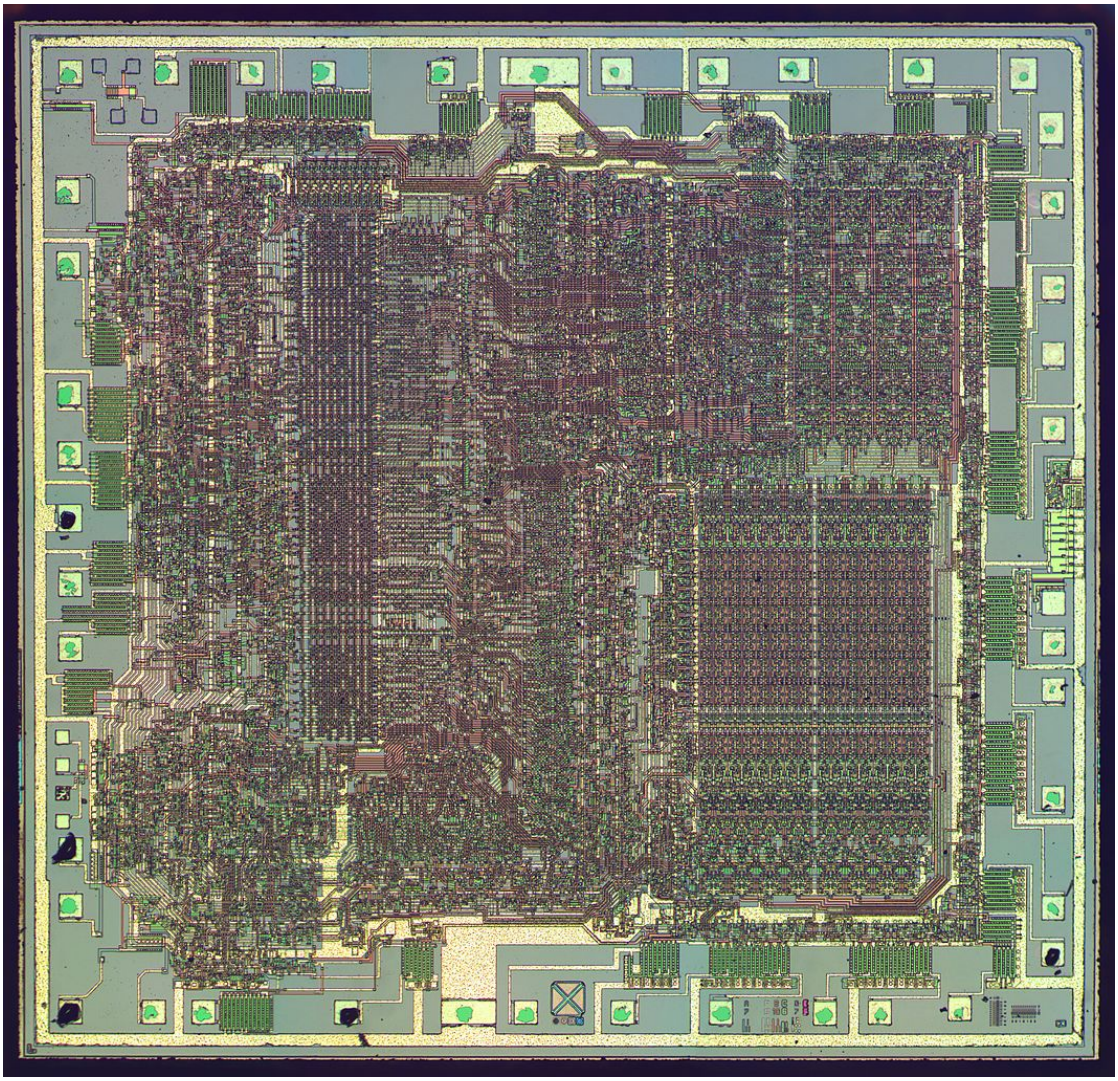
Intel 4004



Source: Wikipedia

Intel's first commercially available microprocessor. It was originally designed for building calculators.

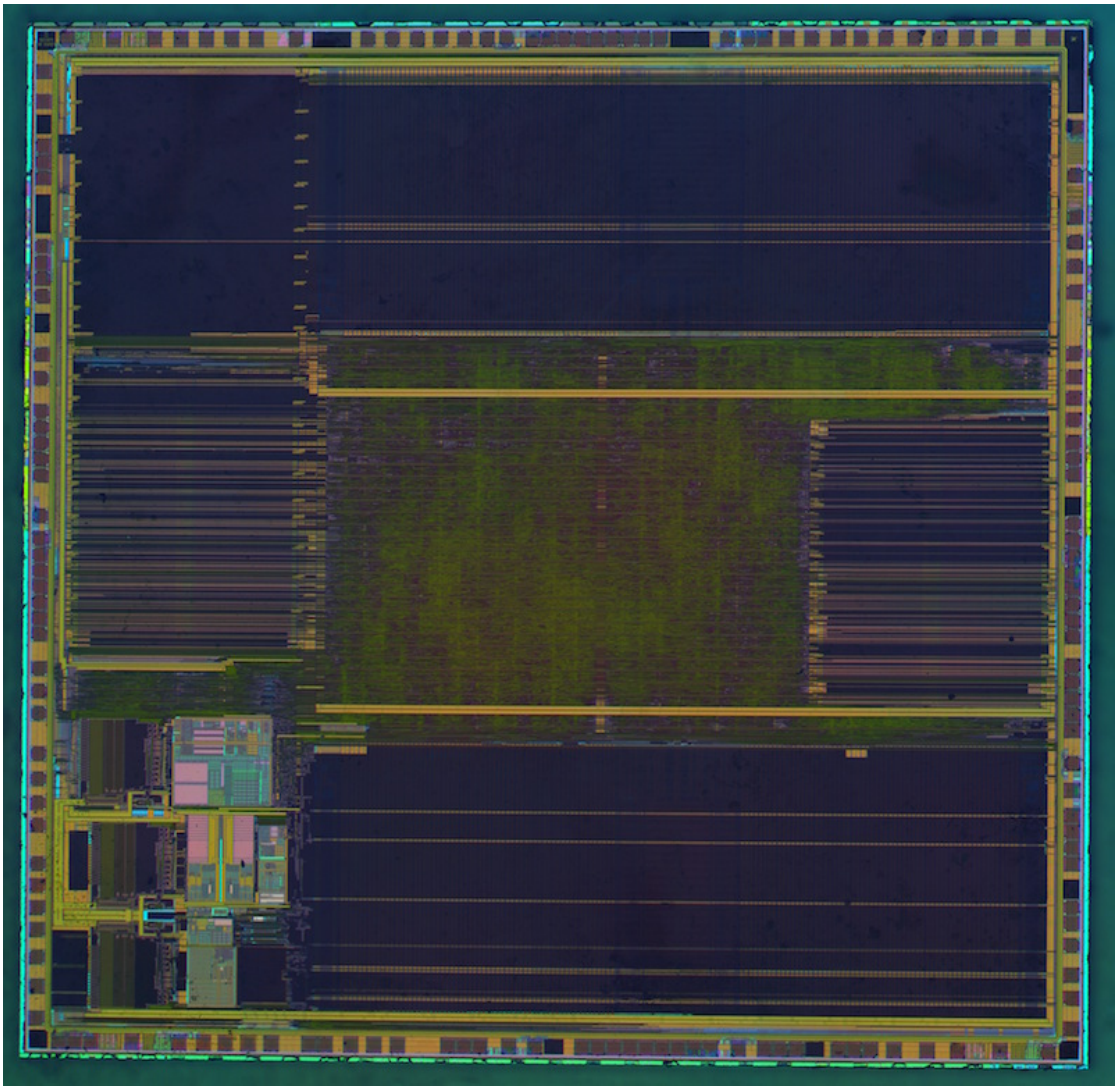
Z80A



Source: Wikipedia

The processor used in a number of home computers in the 1980s.

ARM Cortex

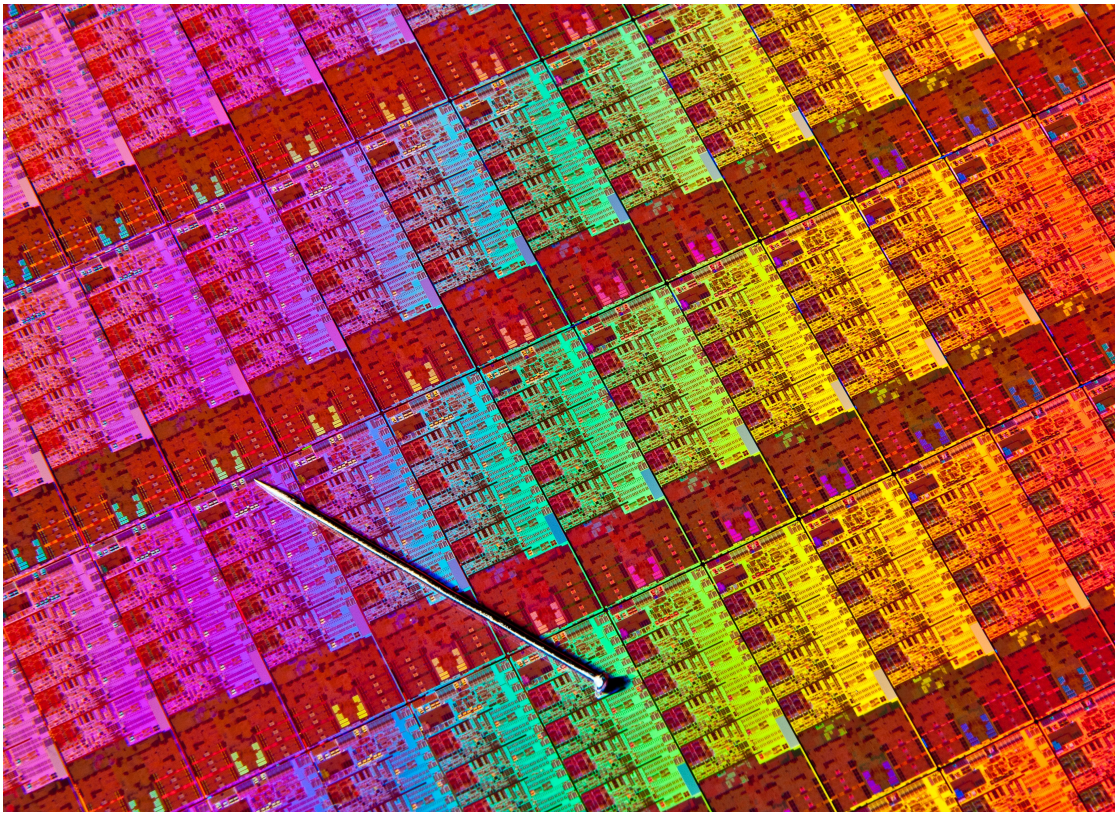


Source: Wikipedia

ARM CPUs are the most common processors for smartphones and tablets, but they are also used in all kinds of other small devices. You can buy a simple ARM-based computer (such as the Raspberry Pi) for around $10.

ARM doesn't actually produce these processors. They just license the design to other companies.

Intel Haswell



Source: Wikipedia

This is a series of Intel chips introduced in 2013 and found in many PC processors (e.g. Core i3, i5, i7 series). On this picture you can actually see many processors on a *wafer* (a slice of silicon, typically 6 or 12 inches in diameter).

Programs

In order to understand how a CPU works and how it is constructed from simple logic gates, we first need to understand *what it is supposed to do*. At a very high level, CPUs are there to execute the programs that we want to run. So let's take a look at some programs in different programming languages.

This is the classic *Hello World* program in the Java programming language:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

This is a Python program that greets you with your name:

```
import sys
name = sys.argv[1]
print 'Hello, ' + name + '!'
```

This is a program in *MiniZinc*, a constraint modelling language. It solves a simple logic puzzle.

```
int: n;
array[0..n-1] of var 0..n: s;
constraint forall(i in 0..n-1) (
  s[i] = sum(j in 0..n-1)(s[j]=i)
);
solve satisfy;
```

So how are these very different programs executed by a CPU? Well, it turns out that **none** of them are.

None of these can be executed directly by the CPU!

They are *compiled* or *interpreted*.

The CPU can only execute **machine code**.

A **compiler** takes a program in a language like Java or C++ and translates it into a lower level language. E.g. in the case of C++, it translates it directly into machine code that can be executed by the CPU. In other cases, such as Java, it translates it into so called *byte code*, which in turn is interpreted.

An **interpreter** executes, or interprets, the instructions written in an interpreted programming language such as Python, without first translating them into machine code. The interpreter is itself a program, usually written in a compiled programming language (so it is running directly) on the CPU.

There are all kinds of variants of interpretation and compilation, but the main message to take home here is that **CPUs can only execute machine code**.

Machine Code

- A very simple computer language.
- Different for each CPU architecture.
- Programs = **sequences** of **instructions**.
  - stored in memory
  - each instruction is encoded into one or more **words**

Example memory contents:

```
0001000000000100
0011000000000101
0010000000000110
0111000000000000
0000000010001110
0000110110000000
0000000000000000
```

- each line is one binary 16-bit word
- could be a program, could be data!

We'll see later that the first four words are in fact instructions, while the last three are data (numbers in this case). This is a really important point: the memory contains both program instructions and data.

The instructions that a particular type of CPU understands are called the

Instruction Set Architecture (ISA).

What does a CPU need to be able to do?

- do some maths (add, subtract, multiply, compare)
- move data between memory, CPU and I/O
- execute *conditionals* and *loops*

Recall the Von Neumann architecture from week 2. We already saw that the CPU contains *registers*, an *arithmetic-logic unit (ALU)*, and a *control unit*. The instruction set needs to reflect the capabilities of these units.

We'll now introduce a very simple CPU architecture and instruction set that will help us understand how CPUs work.

MARIE: A simple CPU

Very basic machine architecture:

- 16 bit words
- only 16 instructions
- one general purpose register
- instructions are composed of an *opcode* (4 bits) and an *address* (12 bits)

opcode	address
0001	000110001110

The MARIE architecture works with 16 bit words, i.e., the size of one unit of information (e.g. a number stored in a single memory location or a register) is 16 bits. Instructions are also 16 bits (so that one instruction fits exactly into one memory location). The table above shows how the CPU interprets one of these 16-bit pieces of data as an instruction: the left-most 4 bits tell it which instruction it is (the opcode), and the remaining 12 bits tell it the address of the memory location that the instruction should operate on.

See Moodle for MARIE simulator link.

MARIE: Registers

A *register* is a storage cell for temporary data.

- **AC** (Accumulator): a "general purpose" register.
- **MAR** (Memory Address Register): stores memory addresses.
- **MBR** (Memory Buffer Register): holds data read from or written to memory.
- **IR** (Instruction Register): current instruction.
- **PC** (Program Counter): address of next instruction.

The MARIE CPU only has a single general purpose register, i.e., one that can be used by the instructions to store e.g. results of arithmetic operations etc. Since there is only one such register, the instructions don't need to contain information about which register to use. That is why each instruction only consists of an opcode and an address - if it uses a register, it's always the AC.

Modern CPUs have many more general purpose registers. The reason is that a memory transfer (i.e. loading a word of data from memory into a register) is incredibly slow compared to anything that happens inside the CPU. So the more intermediate results a program can keep in registers, the faster it will run.

Assembly language

Machine code is hard to write and read.

What does 0010000000000110 mean?

So we use *assembly language*:

- one *mnemonic* per machine instruction

- can be translated 1:1 into machine code

MARIE assembly

This is an overview of half of the instruction set for MARIE. The X in the instructions stands for the address part.

Opcode	Instruction
0001	`Load X`
0010	`Store X`
0011	`Add X`
0100	`Subt X`
0111	`Halt`
1000	`Skipcond`
1001	`Jump X`

(There's a few missing, we'll cover them later.)

MARIE programming

Let's write a small program adding two numbers.

Pseudocode:

```
1. Load first number from memory into AC
2. Add second number from memory to AC
3. Store result from AC into memory
4. Stop execution.
```

A pseudocode is a program written in a "programming language" that doesn't exist. It is useful when you plan a program, since you can just write down the general structure without having to use the exact syntax of a particular programming language.

In this case, the program consists of four steps that are supposed to be executed in this order. Let us now translate it into CPU machine code.

```
Address  Instruction       Memory contents

0x000    Load   0x004      0001000000000100
0x001    Add    0x005      0011000000000101
0x002    Store  0x006      0010000000000110
0x003    Halt              0111000000000000
0x004    142               0000000010001110
```

```
0x005    3456              0000110110000000
0x006    0                 0000000000000000
```

Note: program and data both reside in memory!

The CPU follows the **fetch-decode-execute cycle** in order to run the program:

- The PC (program counter) register contains the memory address where the next instruction to be executed is stored. In the fetch cycle, the CPU transfers the instruction from memory into the IR (instruction register).
- It then increments the PC by one. This concludes the fetch cycle.
- In the decode cycle, the CPU looks at the leftmost 4 bits in IR (which represent the *op-code*), and then gets any data ready. For example, if the instruction requires data from memory, it places the bottom 12 bits of IR into MAR.
- In the execute cycle, if the instruction needs to read from memory, the data is transferred from the memory cell pointed to by MAR into the memory buffer register MBR. Then the actual operation that is encoded in the instruction is executed.

The HTML version of the slides contains an animation that shows you how data flows from memory into registers, ALU and back. It illustrates the *data paths* of the MARIE architecture. You can also access the MARIE data path view from the online simulator, which can step through a program and highlight which parts of the machine are active at any step.

MARIE instructions

Let's go through the most important instructions.

We will use **Register Transfer Language** (RTL) to specify what each instruction means.

RTL is a simple notation that breaks down an instruction into a small number of steps that the CPU must follow to execute the instruction. All steps consist of moving data between registers, hence the name.

The RTL for the MARIE instructions mentioned in the table before look as follows:

Load X: Load data from address *X* into `AC`.

```
MAR ← X
MBR ← M[MAR]
AC  ← MBR
```

Store X: Store data in `AC` into address *X*.

```
MAR     ← X
MBR     ← AC
M[MAR] ← MBR
```

Add X: Add data in address *X* to `AC`. Store result in `AC`.

```
MAR ← X
MBR ← M[MAR]
AC  ← AC + MBR
```

Subt X: Subtract data in address *X* from `AC`, store result in `AC`.

```
MAR ← X
MBR ← M[MAR]
AC  ← AC - MBR
```

Halt: Stop execution.

Skipcond X: Skip next instruction based on AC.

- If X is 0 and `AC` < 0
- If X is hex 400 (*010000000000*) and `AC` = 0
- If X is hex 800 (*100000000000*) and `AC` > 0

then

```
PC ← PC + 1
```

Jump X: Continue execution at address *X*.

```
PC ← X
```

We will see a few more instructions later.

Constructing a MARIE CPU

Circuits required to build a MARIE CPU:

- Perform simple maths (addition, subtraction etc)
- Store and load data in registers and memory
- Fetch, decode and execute instructions

Let's start with the basics.

Combinational Circuits

(output is a function of the inputs)

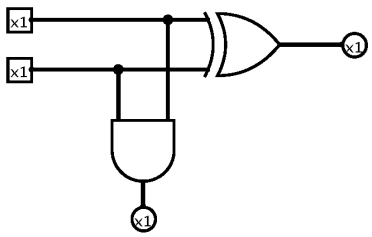
Adders

The simplest possible circuit for doing arithmetic is adding two bits together. The result is a single output bit and a carry.

Here is the **truth table** for adding two bits:

Bit 1	Bit 2	Result	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Let's try this in Logisim.



This is called a **half-adder**.

But what if we have an input carry?

Full Adders

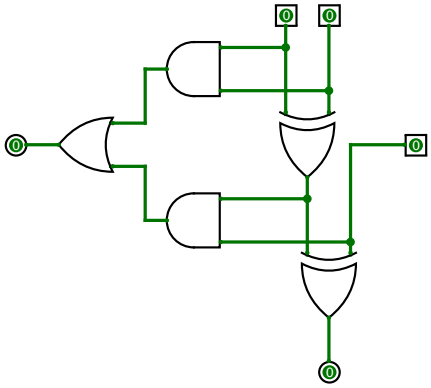
Adding **three** bits.

We now have two regular inputs (the bits we're adding up) and one "carry in". The result can still always be represented with two bits (it can at most be 3, which is 11 in binary). Here is the corresponding truth table and a circuit that implements it.

Bit 1	Bit 2	Carry In	Result	Carry Out
0	0	0	0	0

... continued on next page

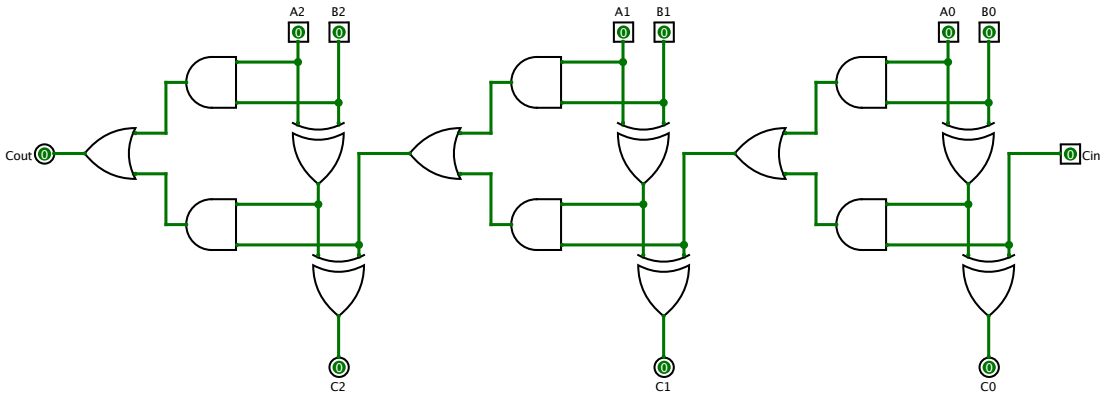
Bit 1	Bit 2	Carry In	Result	Carry Out
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1



Ripple-Carry Adder

The full adder circuit may seem very primitive (when do we ever want to add up individual bits???). But because it can deal with a carry in and produces a carry out, we can put several full adders together to create a circuit that can add longer binary numbers.

Join carry-out of several 1-bit-adders!



This circuit computes A+B=C for 3-bit numbers A, B and C. Note how the carry out of each full adder is fed into the carry in of the next adder (starting from the rightmost full adder).

Delay

Gates require time to **propagate** the signals.

When the input changes, it takes a (short) amount of time before the output reflects that change. In a complex circuit, the *longest path* through the circuit determines the *propagation delay*, i.e., the time one has to wait before the output is guaranteed to be correct.

For n bit ripple carry: $2n+1$

The time it takes e.g. an adder to perform one n bit addition is crucial for the performance of a processor. An addition is usually supposed to take no more than one clock cycle. Let's use a clock frequency of 1 GHz (1 billion cycles per second) as an example. One clock cycle then takes 1 ns (nanosecond). If we implement 32 bit addition using a ripple-carry adder, the propagation delay is roughly 68, meaning that within the 1 nanosecond, 68 gates have to fully propagate their value. The delay of an individual gate must therefore not be longer than 1/68 of a nanosecond, or 14 picoseconds.

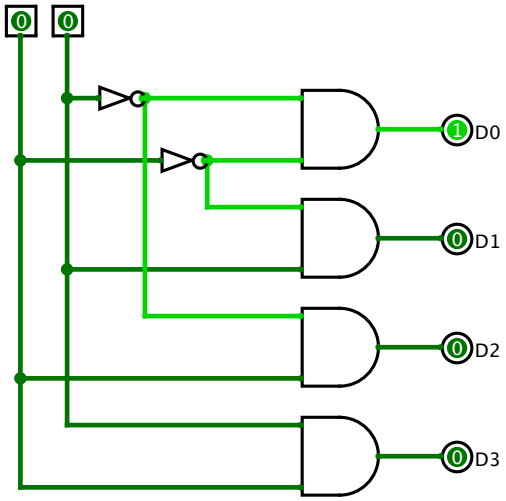
(There are more efficient adders.)

The trick with more efficient adders is to avoid having the carry ripple through all bits. Instead, circuits are added that pre-compute whether entire groups of bits will generate a carry.

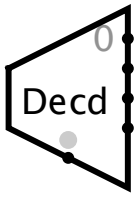
Decoders

Activate one output based on a binary number.

This is a very simple circuit that turns a binary number into a single signal output. We will use this type of circuit a lot since it allows us to decode instructions.



This is the symbol commonly used for decoders in diagrams:

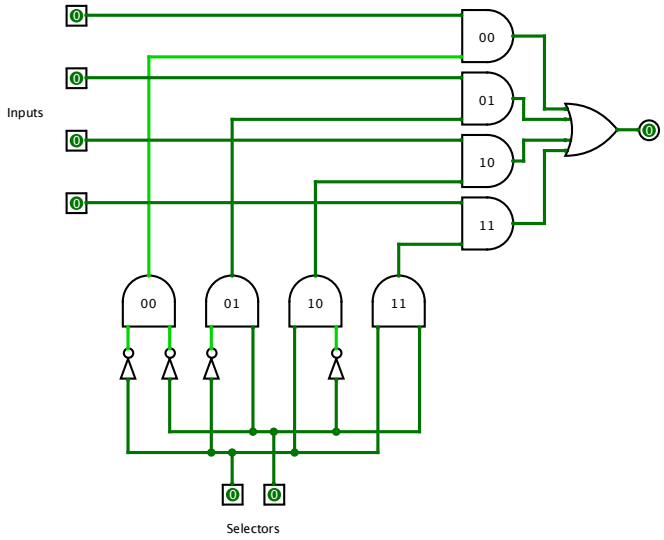


Multiplexers

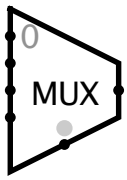
Select one of several inputs.

This is a simple circuit that consists of a *selector* input, e.g. two bits in this case representing the numbers from 0 to 3, and a *data input*, which has 4 bits in our example. The circuit has a single output that is equal to the selected input bit. So for instance if the selector is set to 10, then the output will be equal to the third input from the top.

Have a look at how the selector bits are decoded.



This is the symbol commonly used for multiplexers (MUXes) in diagrams:



Arithmetic Logic Unit

ALU

Implements basic computations:

- integer addition, subtraction, multiplication, negation
- bitwise Boolean operations (and, or, not)
- shifting

Inputs:

- two n-bit **operands**
- op-code (which operation?)

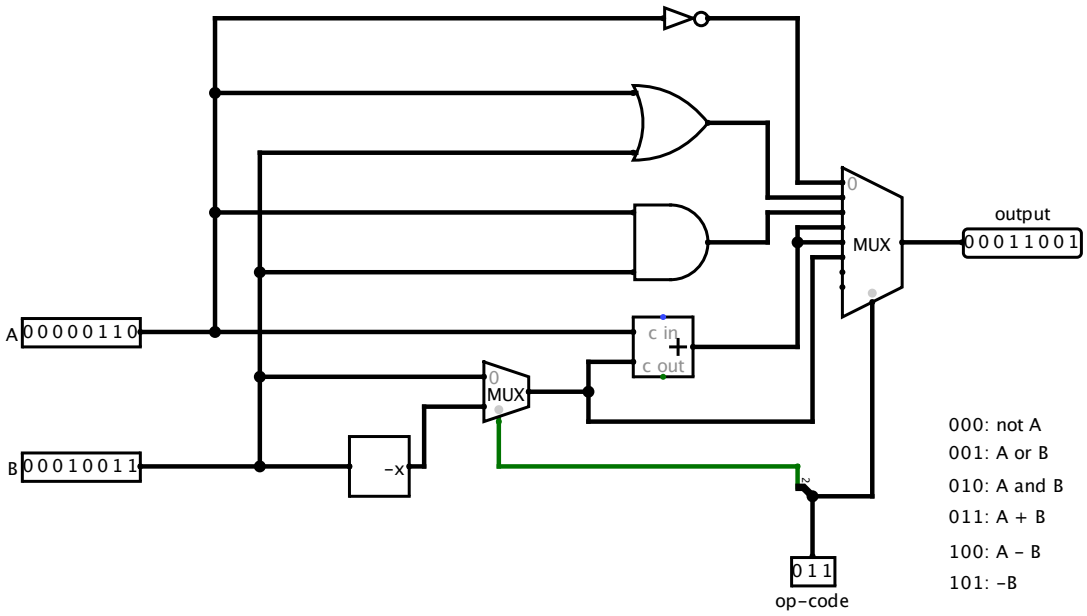
Outputs:

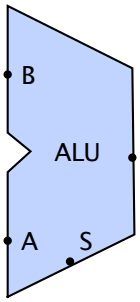
- n-bit result and status flags (overflow? error?)

How does the ALU decide which operation to use?

- Simply do all in parallel
- And then only use the one prescribed by the op-code

Sounds like a job for a MUX!





In this circuit, the lines between some of the components are more than one bit wide. That is a very convenient feature of circuit design tools, because otherwise we would have to make 8 distinct connections for each wire.

The gates (and components like MUXes) are also available in multi-bit versions. You can also see a splitter just above the op-code input in the bottom right corner: in this particular case, it takes a 3-bit input and splits it up so that bit 2 can be accessed individually. This bit is then connected to the small MUX. Have a look at the op-codes: bit 2 distinguishes between addition and subtraction. If it is set to 1, the small MUX will use the output of the -x circuit (to implement A-B), otherwise it will use B directly (to get A+B).