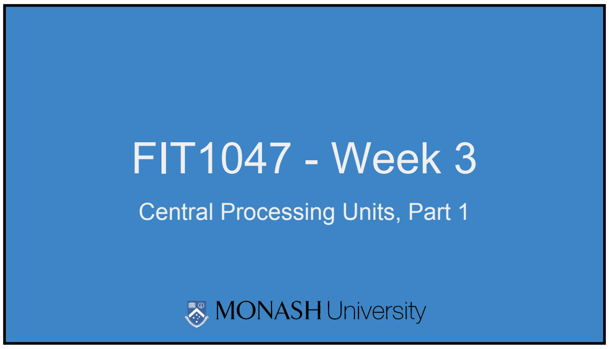
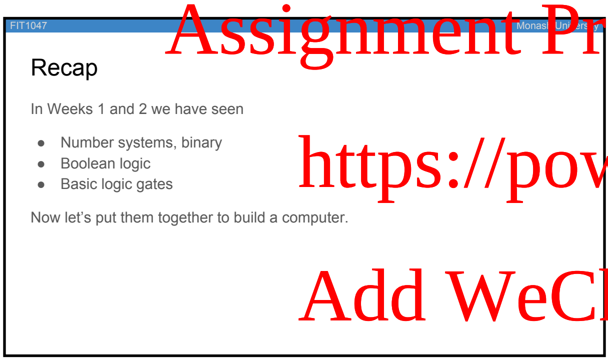


FIT1047 - Week 3

Central Processing Units, Part 1

MONASH University



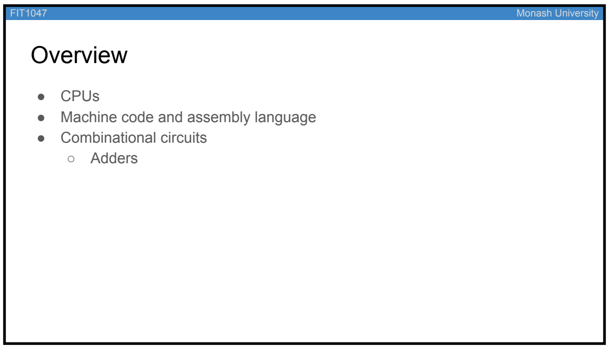
FIT1047 Monash University

Recap

In Weeks 1 and 2 we have seen

- Number systems, binary
- Boolean logic
- Basic logic gates

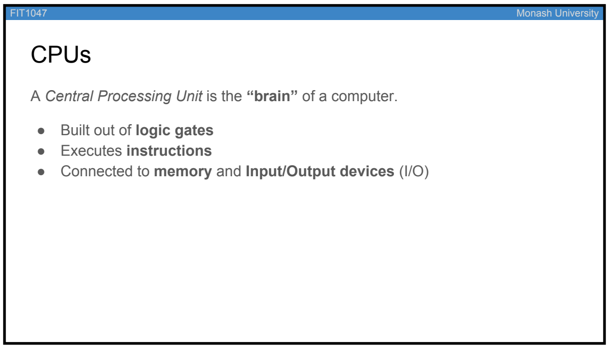
Now let's put them together to build a computer.



FIT1047 Monash University

Overview

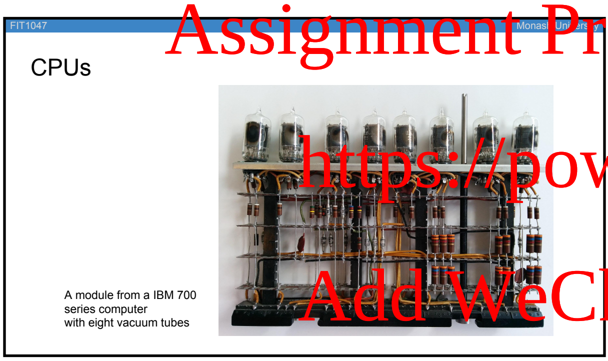
- CPUs
- Machine code and assembly language
- Combinational circuits
    - Adders



CPUs

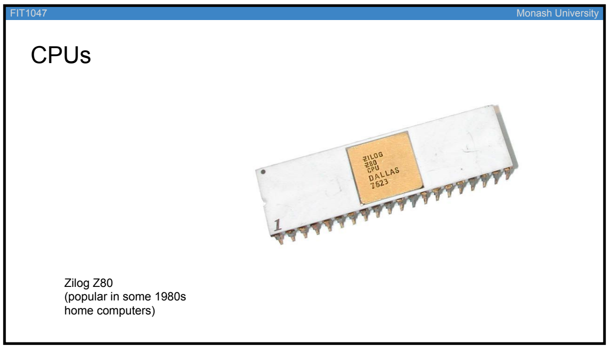
A *Central Processing Unit* is the **“brain”** of a computer.

- Built out of **logic gates**
- Executes **instructions**
- Connected to **memory** and **Input/Output devices** (I/O)



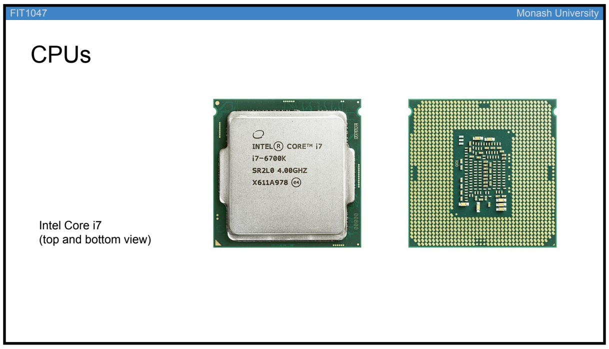
CPUs

A module from a IBM 700
series computer
with eight vacuum tubes



CPUs

Zilog Z80
(popular in some 1980s
home computers)

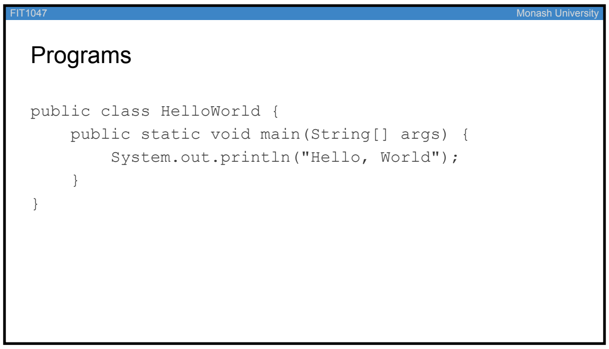


CPUs

Intel Core i7
(top and bottom view)

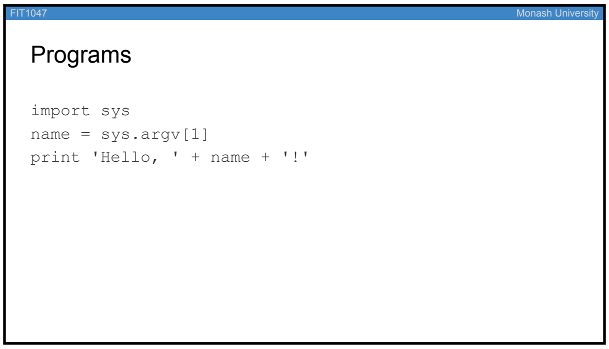
CPUs

ESP8266
Microcontroller with WiFi
($2.50)
Can be used to build
“smart things” (IoT)



Programs

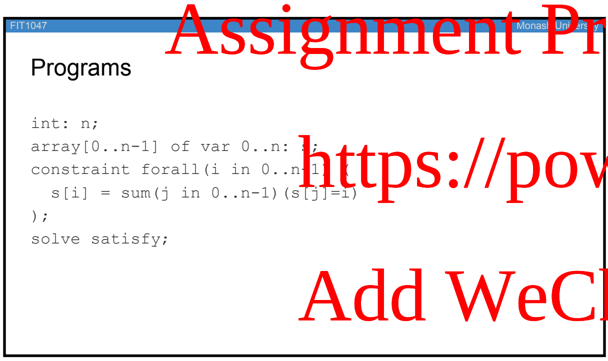
```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```



FIT1047 Monash University

Programs

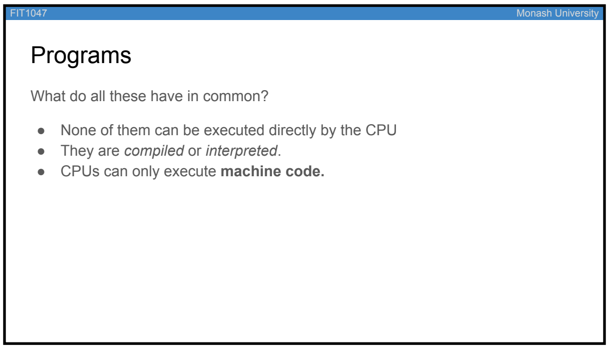
```
import sys
name = sys.argv[1]
print 'Hello, ' + name + '!'
```



FIT1047 Monash University

Programs

```
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;
```



FIT1047 Monash University

Programs

What do all these have in common?

- None of them can be executed directly by the CPU
- They are *compiled* or *interpreted*.
- CPUs can only execute **machine code.**

Machine Code

- A very simple computer language.
- Different for each CPU architecture
  - E.g. different machine code in your smartphone, your laptop and your washing machine
- Machine code programs are
  - Sequences of **instructions**
  - Stored in memory
  - Each instruction is encoded into one or more **words**

Machine Code

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

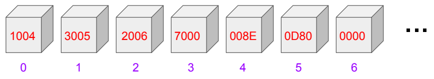
Instruction Set Architecture (ISA)

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*

Memory

Think of it as a sequence of “boxes”:



1004	3005	2006	7000	008E	0D80	0000	…
0	1	2	3	4	5	6	

Each box contains a **value** (here: a 16-bit number).

This could be a **machine code instruction**, or **data**.

We give each box an **address**: the number of the box, starting from 0.

Registers

Very fast, very small memory inside the CPU

- Each register can store a single word (like one “box”)
- General purpose (GP) register:
  - Used by CPU to store temporary values for calculations
  - Can be used like a variable in programs
- CPU contains fixed small number (e.g. 16 GP registers for Intel CPUs)
- Special purpose register:
  - Used internally to enable CPU operations
  - Cannot be used directly in programs

Assembly Language

Machine code is hard to write and read.

Example: what does `0010000000000110` mean?

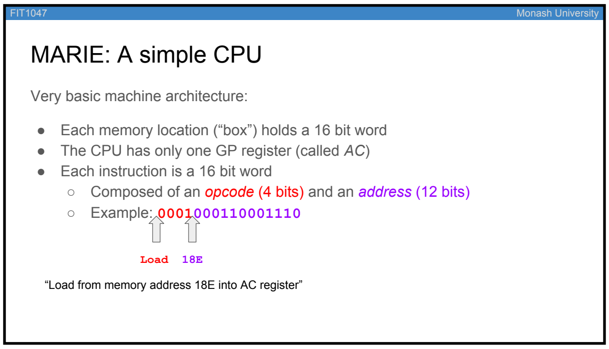
We use *assembly language:*

- Each instruction has a *mnemonic*, a word that is easy to remember
- Assembly language can be translated easily into machine code
  - Each line in the program is one instruction in machine code

Assembly Instructions

These are not real instructions, just examples.

- `Load 0xA003, R0`
  Load the number stored in memory at address A003 into GP register R0
- `Add R0, R1, R2`
  Add the number stored in R0 to the number stored in R1, store the result in R2
- `Store R0, 0xA004`
  Store the number in R0 into memory address A004
- `Jump 0x1000`
  Continue program execution at address 1000



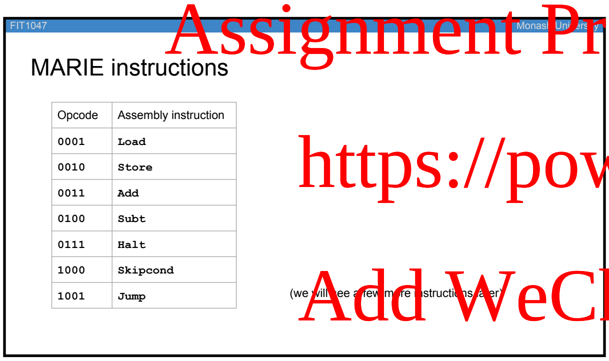
MARIE: A simple CPU

Very basic machine architecture:

- Each memory location (“box”) holds a 16 bit word
- The CPU has only one GP register (called *AC*)
- Each instruction is a 16 bit word
    - Composed of an *opcode* (4 bits) and an *address* (12 bits)
    - Example: 0001000110001110

Load 18E

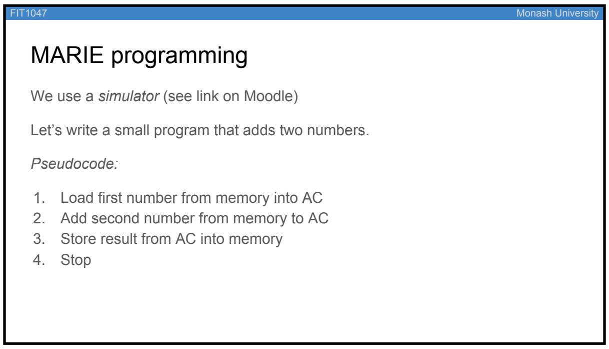
“Load from memory address 18E into AC register”



MARIE instructions

Opcode	Assembly instruction
0001	Load
0010	Store
0011	Add
0100	Subt
0111	Halt
1000	Skipcond
1001	Jump

(we will see a few more instructions later)



MARIE programming

We use a *simulator* (see link on Moodle)

Let’s write a small program that adds two numbers.

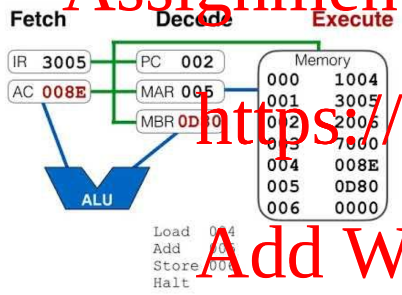
Pseudocode:

```
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
```

MARIE programming

Address	Memory contents	Instruction	Data (decimal)
000	0001000000000100	Load 004	
001	0011000000000101	Add 005	
002	0010000000000110	Store 006	
003	0111000000000000	Halt	
004	0000000010001110		142
005	0000110110000000		3456
006	0000000000000000		0

Note: program and data share the same memory!



Memory	
000	1004
001	3005
002	2006
003	7000
004	008E
005	0D80
006	0000

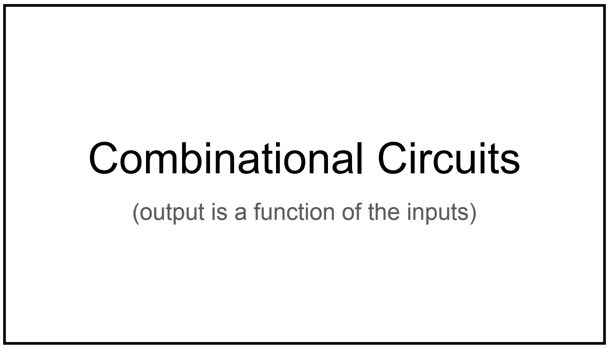
```
Load  004
Add   005
Store 006
Halt
```

Constructing a MARIE CPU

Circuits required to build a MARIE CPU:

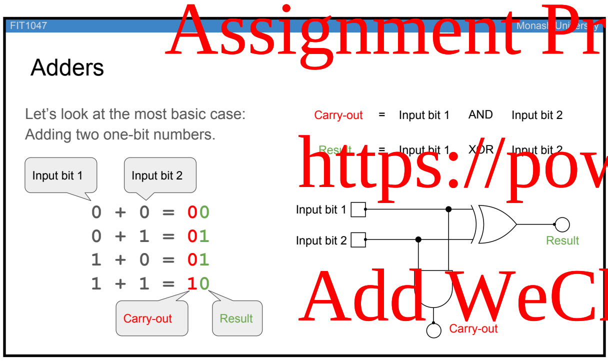
- Perform simple maths (addition, subtraction, comparison)
- Store and load data in registers and memory
- Fetch, decode and execute instructions

Let's start with the basics.



Combinational Circuits

(output is a function of the inputs)



Adders

Let's look at the most basic case: Adding two one-bit numbers.

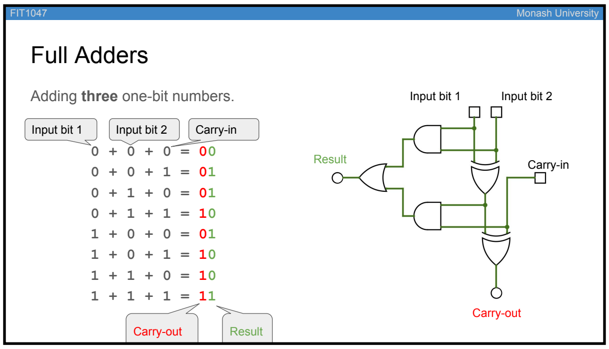
Input bit 1 Input bit 2

0 + 0 = 00
0 + 1 = 01
1 + 0 = 01
1 + 1 = 10

Carry-out Result

Carry-out = Input bit 1 AND Input bit 2

Result = Input bit 1 XOR Input bit 2



Full Adders

Adding **three** one-bit numbers.

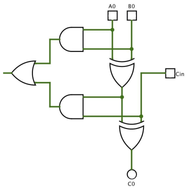
Input bit 1 Input bit 2 Carry-in

0 + 0 + 0 = 00
0 + 0 + 1 = 01
0 + 1 + 0 = 01
0 + 1 + 1 = 10
1 + 0 + 0 = 01
1 + 0 + 1 = 10
1 + 1 + 0 = 10
1 + 1 + 1 = 11

Carry-out Result

Ripple-Carry Adder

Add two 3-bit numbers (A+B=C):



Outlook

Tutorials this week:

- MARIE programming
- Circuits for adding and subtracting

Next lecture:

- More circuits