FIT1047 - Week 4

Part 1:
Control, Memory, Indirect Addressing

MONASH University

Recap

Last week we saw

- Basic MARIE programming
- Combinational circuits (decoders, muxes, adders, ALUs)
- Sequential circuits (flip flops, registers)

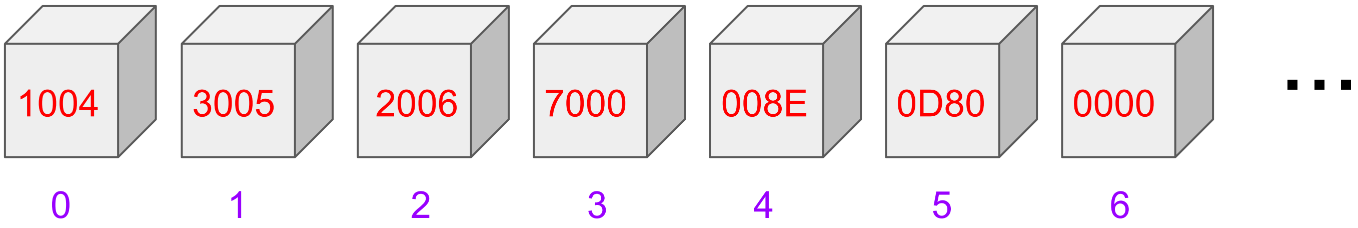
Overview

- Memory organisation
  - Addresses
- Accessing memory
  - Indirect addressing
  - Subroutines

Memory

Memory

Think of it as a sequence of “boxes”:



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.

Programs can **read** and **change** the value stored at a location.

What is stored in memory?

Address
000
001
002
003
004
005
006

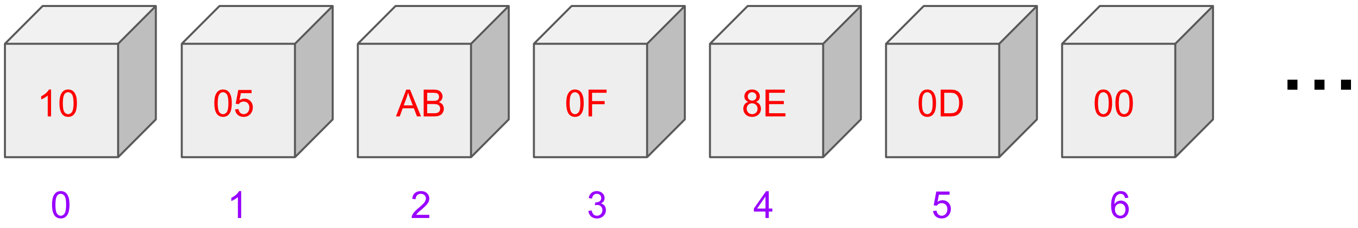
What is stored in memory?

Address	Hex Value	Integer	Bit pattern	Instruction
000	1			Load 004
001	3			Add 005
002	2			Store 006
003	7			Halt
004	0			JnS 08E
005	0D			JnS D80
006	0000	0	0000000000000000	JnS 000

The memory doesn't know!
The CPU doesn't know!
It's up to the program to
interpret the memory
in a certain way.

Addressing

Most architectures store **one byte per memory location:**

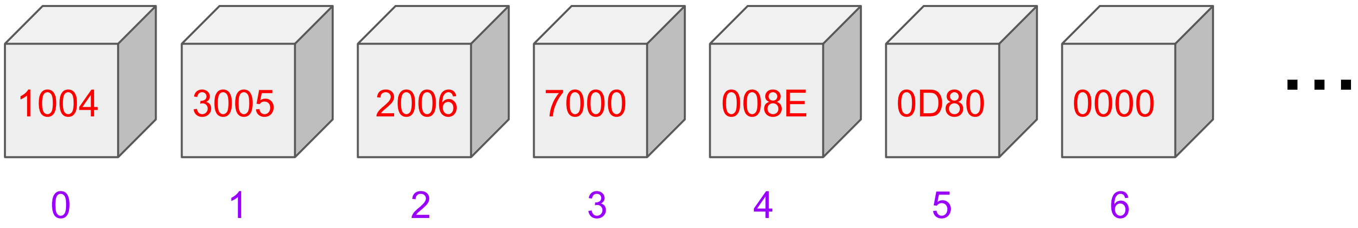


So **each byte has its own address.**

This is called **byte-addressable.**

Addressing in MARIE

Some architectures (including MARIE) store **one word per location:**

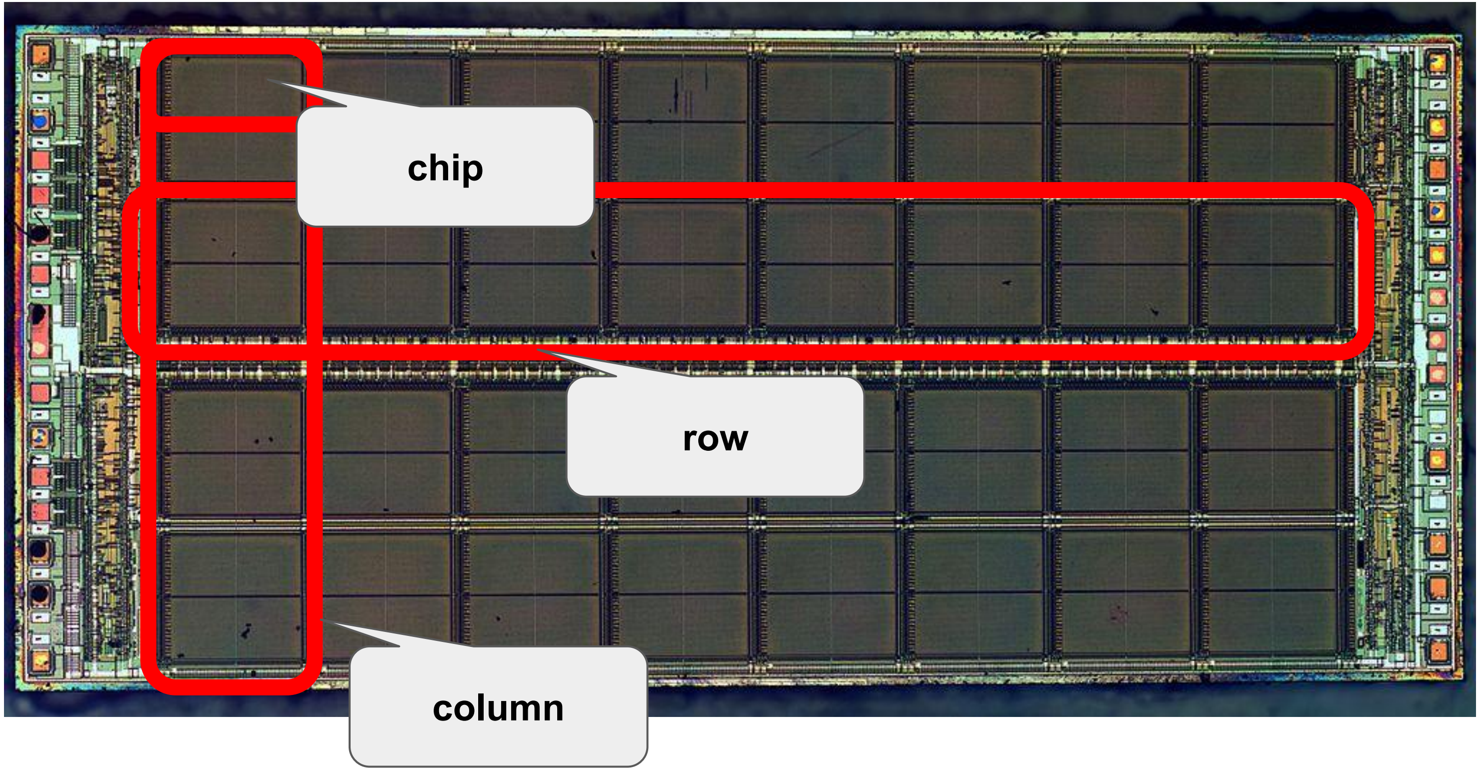


So each **word** has its own address.

This is called **word-addressable.**

Remember: In MARIE, one word is 16 bits.

RAM



A RAM module with 1 megabit (2^{20} bit) capacity. Source: Wikipedia.

RAM

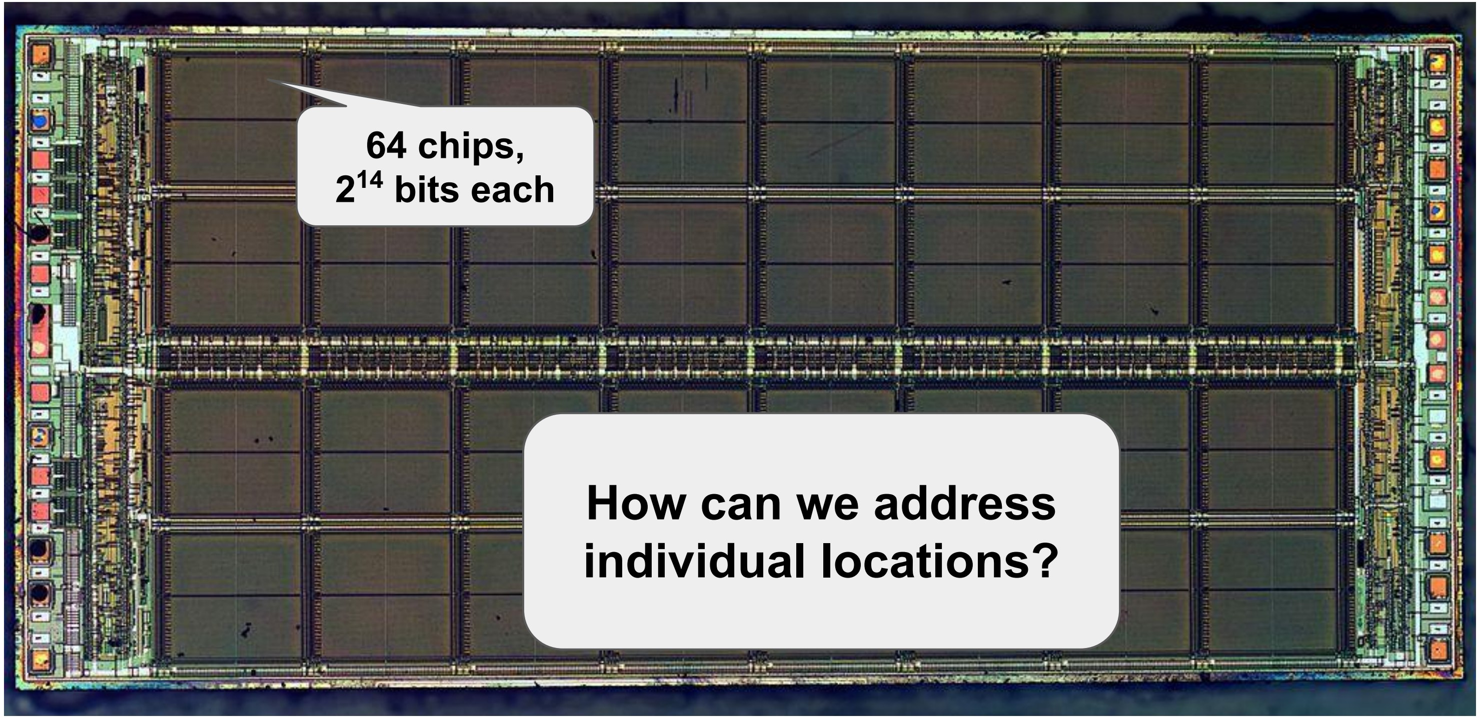
Each **module** is made up of multiple **chips**

Each chip has a fixed size $L \times W$

- L: number of locations
- W: number of bits per location

E.g. 2K×8 means 2×2^{10} locations of 8 bits each = $2 \times 2^{10} \times 2^3 = 2^{14}$ bits per chip

RAM



A RAM module with 1 megabit (2^{20} bit) capacity. Source: Wikipedia.

RAM

8 rows

$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$
$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$
$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$
$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$
$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$
$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$
$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$
$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$	$2^{11}×8$

8 columns = $8×2^{11}$ bytes per row

$8 \times 8 \times 2^{11} =$
2^{17} locations

Each address
is 17 bits long!

RAM addressing

Example address (17 bits):

01011000010010011

Row 2

Column 6

Byte 147

We could implement this using MUXes!

- One MUX per chip selects the correct byte (here: 147)
- One MUX per row selects the chip in a column (here: 6)
- One MUX per module selects the row (here: 2)

Indirect Addressing

Accessing memory in MARIE

So far:

- `Store X`
- `Load X`
- `Add X`
- `Jump X`

Use value stored at **`X`**

This is not very flexible!

Indirect Addressing

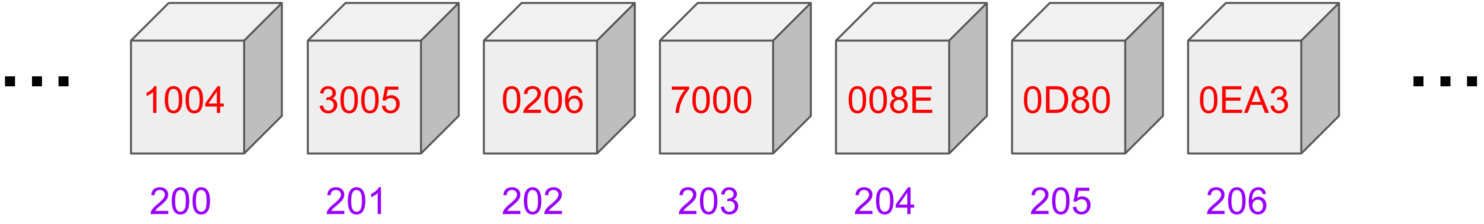
Use *address* stored at **`X`**

Comparison:

- **`Load X`**:
  Load value stored at address **`X`** into **`AC`**.
- **`LoadI X`**:
  Look up value stored at address **`X`**, **use it as an address**, load value from that address into **`AC`**.
  (“indirect load”)

Indirect Addressing

Example:



`Load 202`: Load value from address 202 into **`AC`**. Result: **`AC=0206`**.

`LoadI 202`: Look up value stored at address 202. Value is 0206. Then load value from that address into **`AC`**. Result: **`AC=0EA3`**.

Indirect Addressing

Advantages:

- Addresses don’t need to be hard-coded into our program code
- We can **compute** the address!
- This enables important programming patterns, e.g. looping through a list of values

Indirect Addressing: Example

```
000  Loop,   LoadI Addr
001          SkipCond 800
002          Jump End
003          Add Sum
004          Store Sum
005          Load Addr
006          Add One
007          Store Addr
008          Jump Loop
009  End,    Load Sum
00A          Output
00B          Halt
```

```
00C  One,    DEC 1
00D  Sum,    DEC 0
00E  Addr,   HEX 00F
00F          DEC 70
010          DEC 73
011          DEC 84
012          DEC 0
```

List of numbers

End of list indicated by **DEC 0**

Program computes sum of a list of numbers.
Note: **length of list is not hard-coded!**

Indirect Addressing

Other instructions that work with indirect addressing:

- **`AddI X`**:
  Use *address* stored at **`X`**, load value from that address and add to value currently stored in AC.
- **`JumpI X`**:
  Jump to address stored at address **`X`**.

RTL for `LoadI X`

1. MAR ← PC
2. MBR ← M[MAR]
3. IR ← MBR
4. PC ← PC+1

fetch

5. MAR ← X
6. MBR ← M[MAR]

decode

7. MAR ← MBR
8. MBR ← M[MAR]
9. AC ← MBR

execute

Subroutines

Subroutines

AKA procedures, functions, methods

A piece of code that

- Has a well-defined function
- Needs to be executed often
- We can **call**, passing **arguments** to it
- **Returns** to where it was called from

Subroutines in Machine Code

ISAs provide support for subroutines.

In MARIE:

- **`JnS X`**:
  Stores **`PC`** into **`X`**, then jumps to **`X+1`**.
  `X` hold the **return address.**
  ("Jump and Store")
- **`JumpI X`**:
  Jump to address stored at **`X`**.
  Returns to the calling code.

Subroutine Example

```
            Load  FortyTwo
            Store Print_Arg
            JnS   Print
            Halt

FortyTwo,   DEC 42

            / Subroutine that prints one number
Print_Arg,  DEC 0                / put argument here
Print,      HEX 0                / return address
            Load Print_Arg
            Output
            JumpI Print          / return to caller
```

Summary + Outlook

Memory:

- Stores bits, up to the program to interpret them
- Need one address per byte (in byte-addressable memory) or word (in word-addressable memory, e.g. MARIE)

Indirect addressing + subroutines:

- Important if we want to write more complex programs

Labs this week:

- More MARIE programming