CMPT 295

Unit - Data Representation

Lecture 2 – Representing data in memory

Textbook
Chapter 2

Data → information
→ data
→ instructions
(code, programs)

CAL Volunteer Note-Taker Position

- If you are taking lecture notes in CMPT 295 and your hand writing is 👍 you may be interested in applying for the following volunteer note-taker position:
  - The Centre for Accessible Learning (CAL) is looking for a CMPT 295 note-taker
  - CAL volunteer lecture note-takers are provided with a $100 credit applied to their student account in acknowledgment of their assistance
- Interested?
  - Please see the email CAL has sent us

- Please feel free to call 778-782-3112 or email calexams@sfu.ca the Centre if you have any questions

Last Lecture

- ✓ COVID Protocol
- ✓ What is CMPT 295?
  - ✓ What shall we learn in CMPT 295?
  - ✓ What should we already know?
  - ✓ Which resources do we have to help us learn all this?
- ✓ Activity
- ✓ Questions

Feedback on Lecture 1 Activity

- Thank you for participating in the Lecture 1 Activity!

- Feedback now posted on our course web site
- Check it out!

chapter 2 in our textbook

Unit Objectives

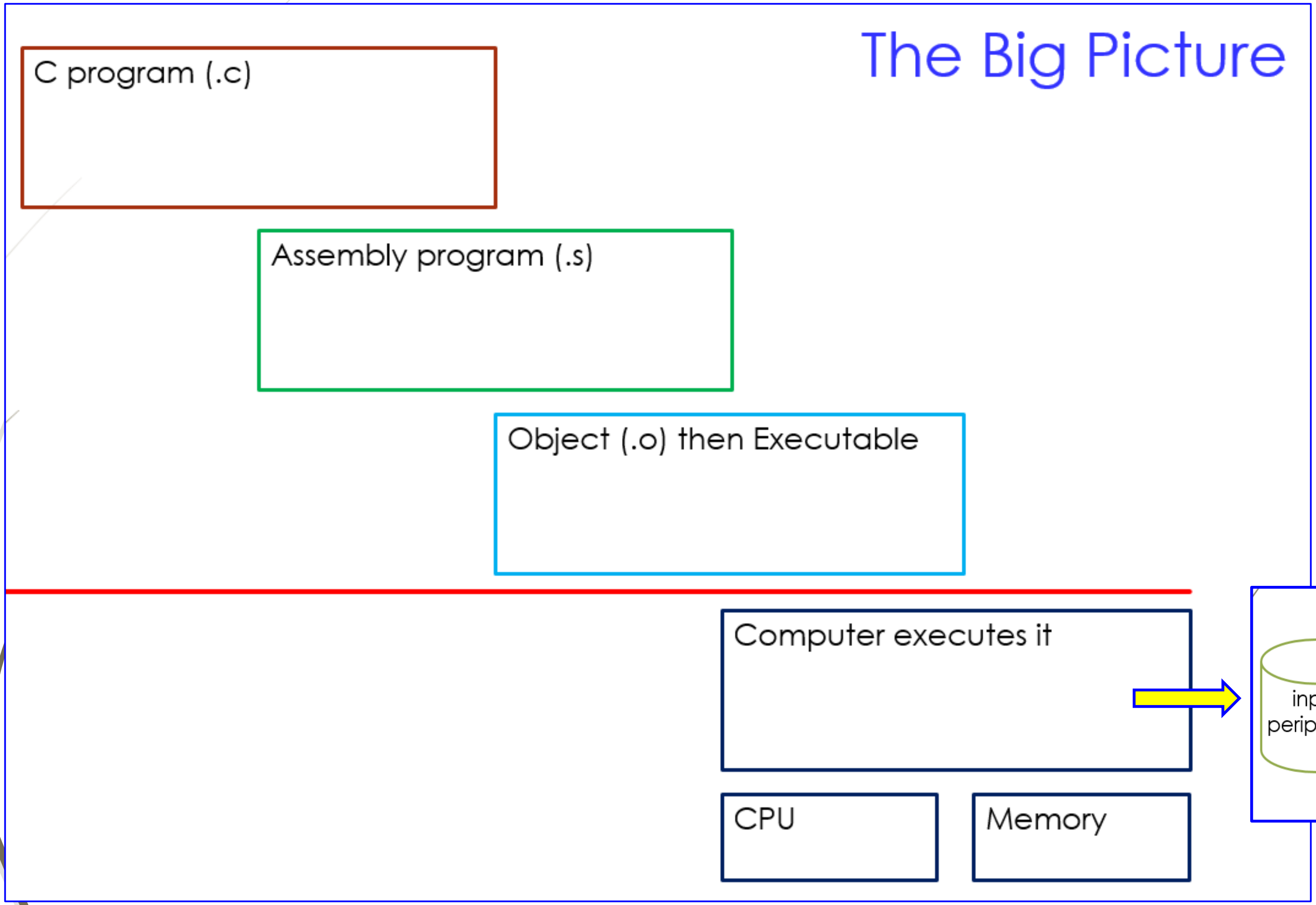
- Understand how a computer represents (encodes) data in (fixed-size) memory
- Become aware of the impact this fixed size has on …
  - Range of values represented in memory
  - Results of arithmetic operations
- Become aware of …
  - How one data type is converted to another
  - And the impact this conversion has on the values

- Bottom Line: allow software developers to write more reliable code

Today's Menu

- Representing data in memory – Most of this is review
  - "Under the Hood" - Von Neumann architecture
  - Bits and bytes in memory
    - How to diagram memory -> Used in this course and other references
    - How to represent series of bits -> In binary, in hexadecimal (conversion)
    - What kind of information (data) do series of bits represent -> Encoding scheme
    - Order of bytes in memory -> Endian
  - Bit manipulation – bitwise operations
    - Boolean algebra + Shifting
- Representing integral numbers in memory
  - Unsigned and signed
  - Converting, expanding and truncating
  - Arithmetic operations
- Representing real numbers in memory
  - IEEE floating point representation
  - Floating point in C – casting, rounding, addition, …

"Under the hood" - Von Neumann architecture

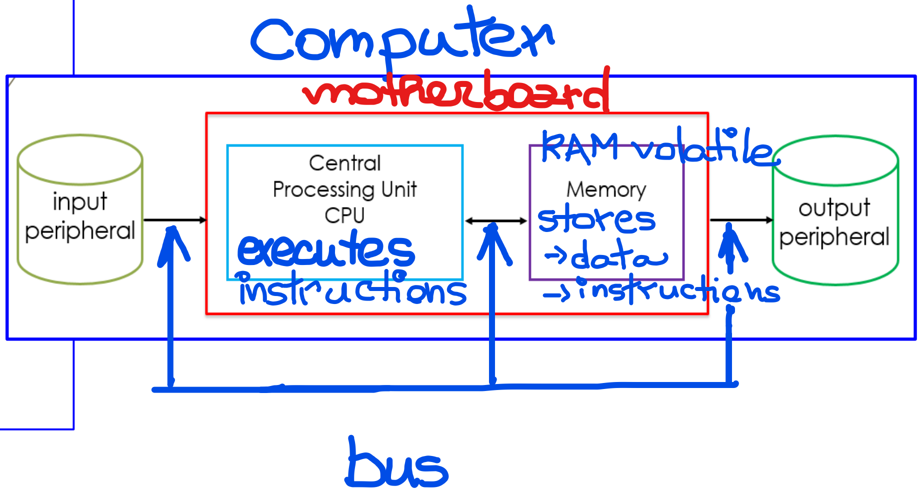


The Big Picture

Architecture of most computers

Its features:

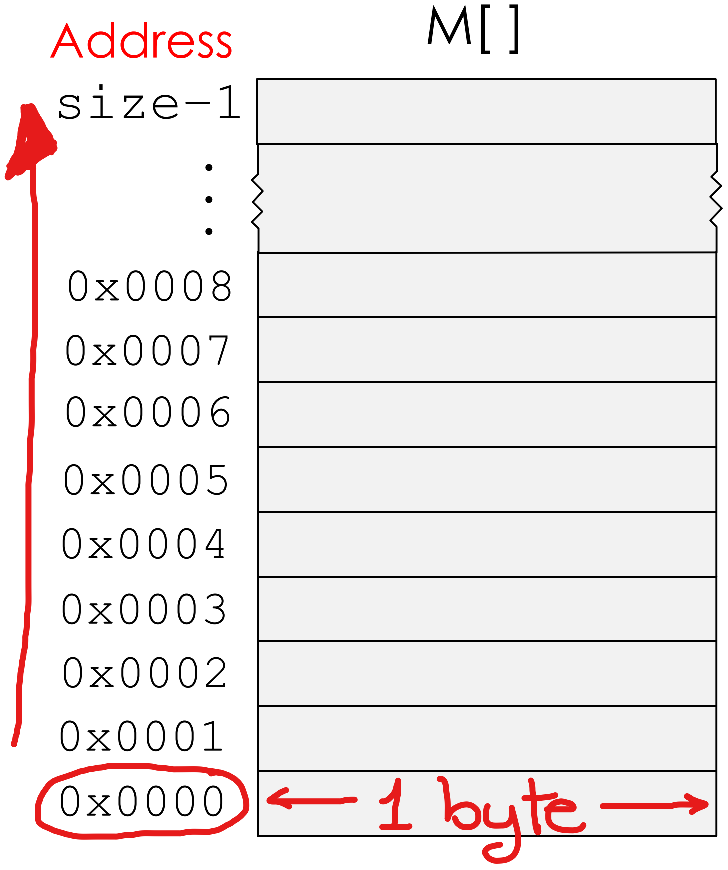
- ✓ CPU, memory, input and ouput, bus
- ✓ Data and instructions (code/programs) both stored in memory



How to diagram memory

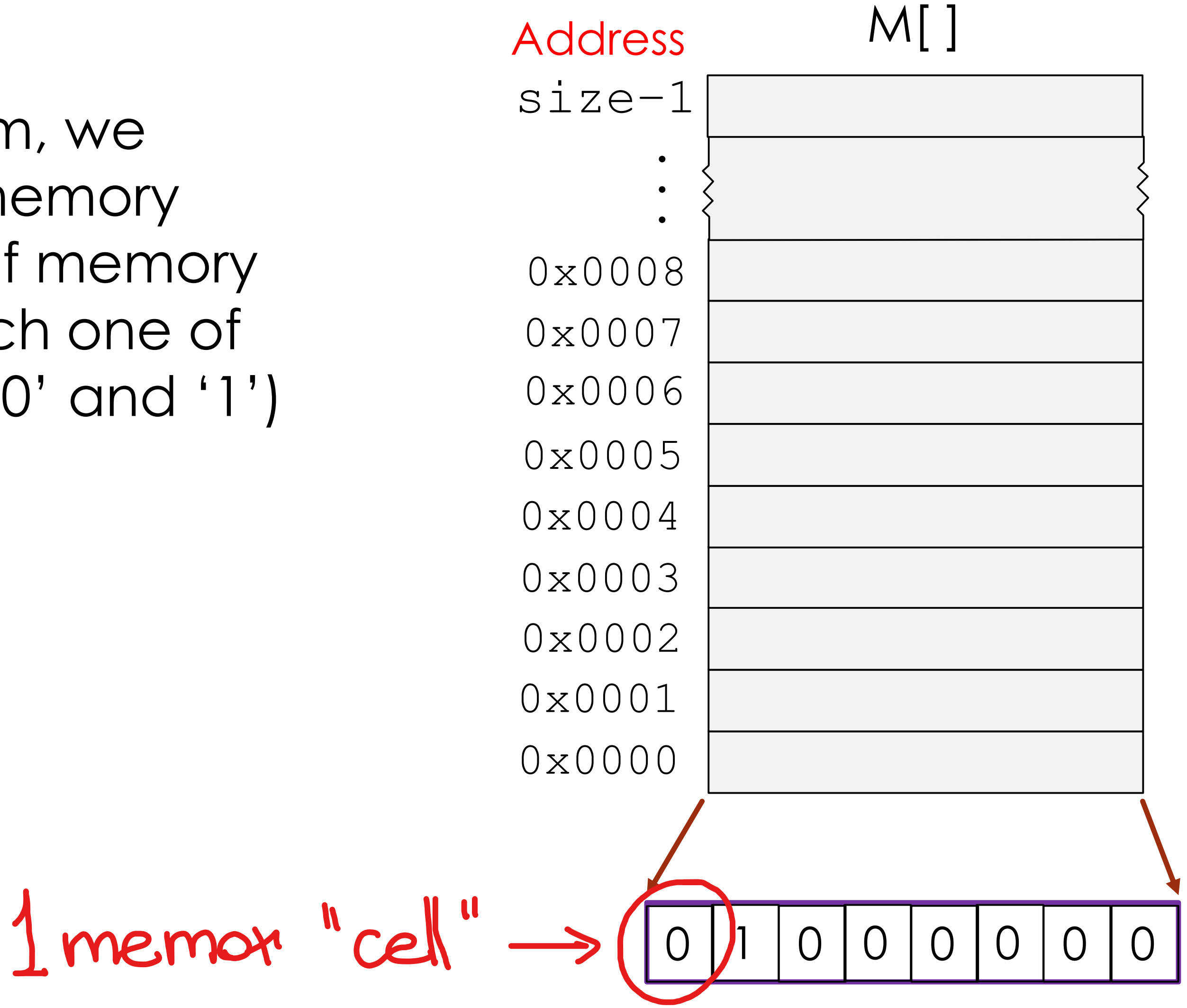
- Seen as a linear (contiguous) array of bytes
- 1 byte (8 bits) smallest addressable unit of memory
  - Each byte has a unique address
  - *Byte-addressable* memory .
- Computer reads a **word** worth of bits at a time (=> word size) w
- Questions:
  1. If word size is 8, how many bytes are read at a time from memory?

     Answer: 1 byte
  2. If a computer can read 4 bytes at a time, its word size is 32 bits .

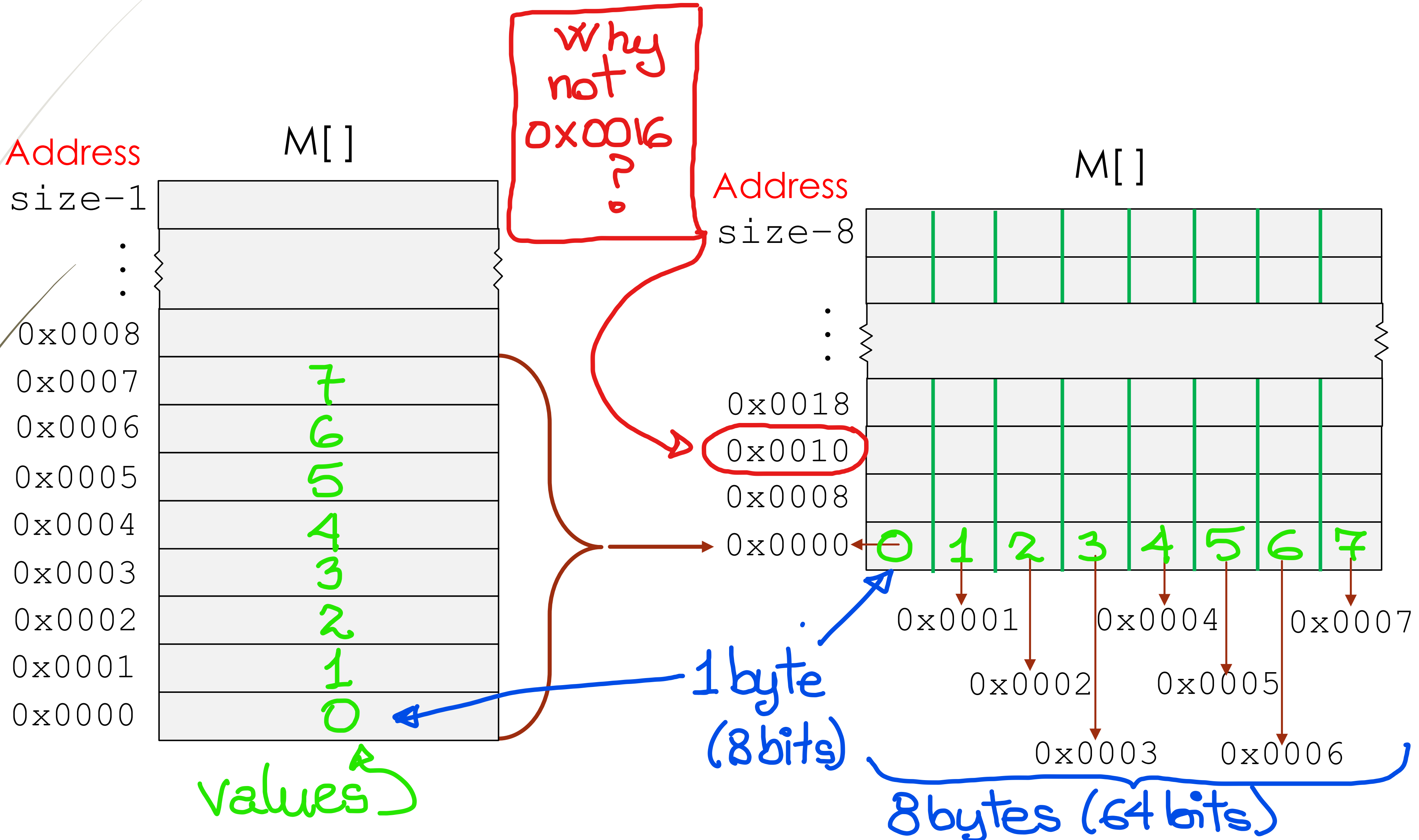


Closer look at memory

- Typically, in a diagram, we represent memory (memory content) as a series of memory "cells" (or bits) in which one of two possible values ('0' and '1') is stored

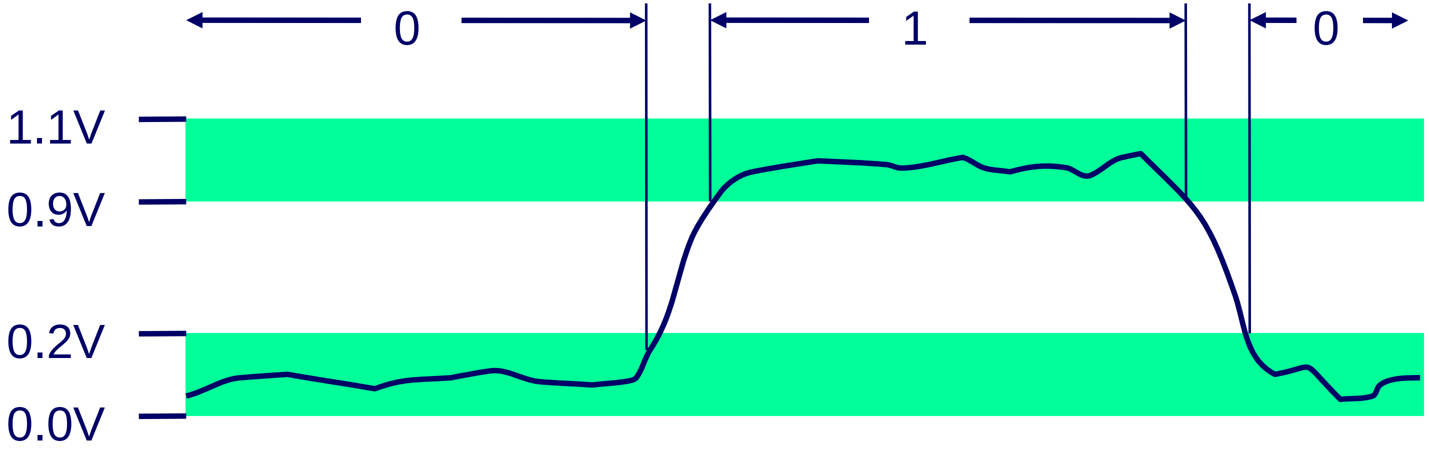


Compressed view of memory



Why can only two possible values be stored in a memory "cell"?

- As electronic machines, computers use two voltage levels
  - Transmitted on noisy wires -> value of two voltage levels vary over a range
  - These ranges are abstracted using "0" and "1"

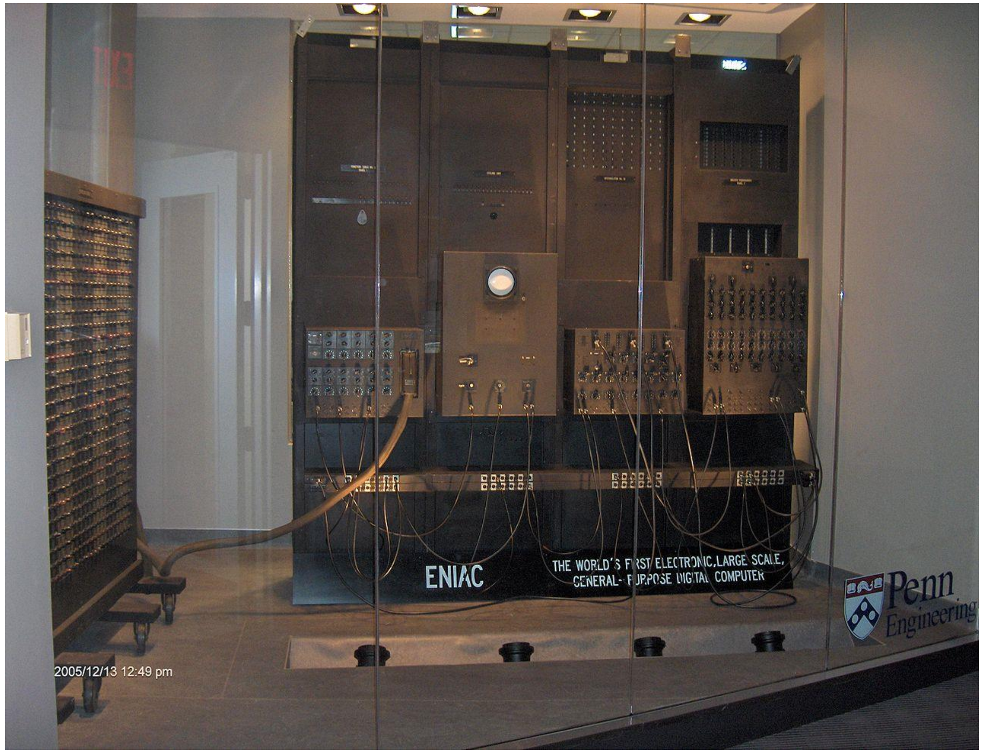


- Back to the question *Why can only two possible values be stored in a memory "cell"?*
  - Because computers manipulate two-valued information

A bit of history

ENIAC: Electronic Numerical Integrator And Calculator

- U. Penn by Eckert + Mauchly (1946)
- Data: 20 × **10-digit** regs
  + ~18,000 vacuum tubes
- To code: manually set switches
  and plugged cables
  - Debugging was manual
  - No method to save program for later use
  - Separated code from the data



Source: https://en.wikipedia.org/wiki/ENIAC#/media/File:ENIAC_Penn1.jpg

Review

Back to our bits
How to represent series of bits

- From binary numeral system
- Base: 2
- Bit values: 0 and 1
- Possible bit patterns in a byte: 00000000_2 to 11111111_2
- *Drawback of manipulating binary numbers?*
  - What number is this?
    - $1001100\ 11001001\ 01000101\ 01001000_2$
  - Lengthy to write -> not very compact
  - Difficult to read

∴ 256 possible bit patterns

Error prone!

Review

A solution: hexadecimal numbers

- Base: 16
- Values: 0, 1, 2, …, 9, A, B, C, D, E, F
- Possible patterns in a byte: 00_{16} to FF_{16}
- Conversion binary -> hex (w=32)

padding →

e.g.: 0100|1100|1100|1001|0100|0101|0100|1000$_2$

0x 4 C C 9 4 5 4 8

conversion algorithm?

HW 0

- Conversion hex -> binary (w=16)

e.g.: $3D5F_{16}$ (in C: 0x3D5F)

⇒ 0011 1101 0101 1111$_2$

conversion algorithm?

Decimal	Binary	Hexadecimal
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

What could these 32 bits represent?
What kind of information could they encode?

$01100010\ 01101001\ 01110100\ 01110011_2$

Answer:

- integer
- string of characters
- colour
- ...

What kind of information (data) do series of bits represent?

Bit pattern →

$01100010\ 01101001$
$01110100\ 01110011_2$

Encoding Scheme

- ASCII character
- Unsigned integer
- Two’s complement (signed) integer
- Floating point
- Memory Address
- Assembly language
- RGB
- MP3
- …

→

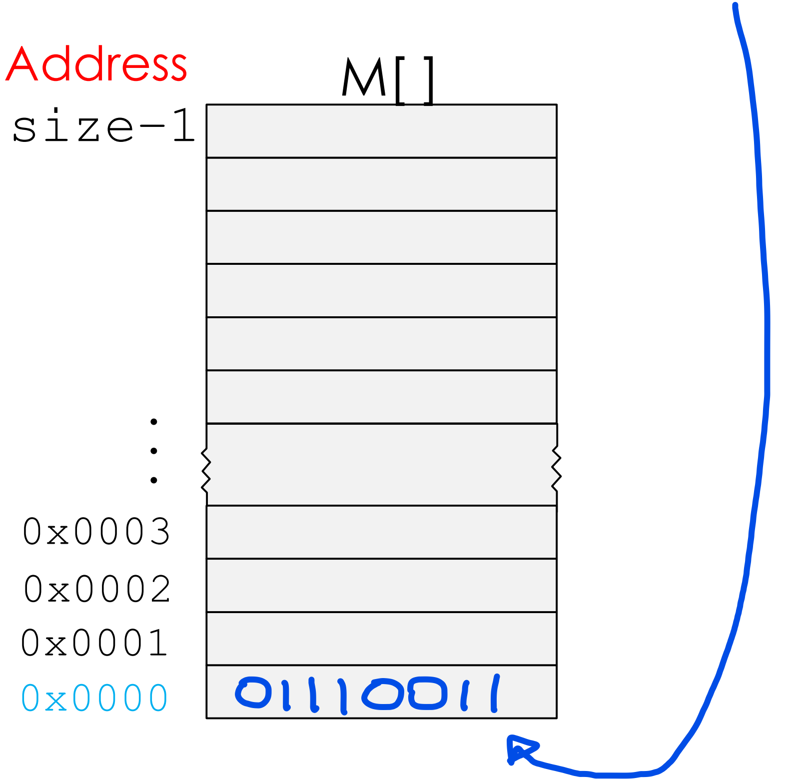
- Letters and symbols
- Positive numbers
- Negative numbers
- Real numbers
- C pointers
- Machine-level instructions
- Colour
- Audio/Sound
- …

Definition: An encoding scheme is an interpretation (representation) of a series of bits

Bottom line: Which encoding scheme is used to interpret a series of bits depends on the application currently executing (the “context”) not the computer

Endian – Order of bytes in memory

- It is straight forward to store a byte in memory
  - All we need is the byte (series of bits) and a memory address
  - For example, let's store byte $0\,1\,1\,1\,0\,0\,1\,1_2$ at address `0x0000`



Endian – Order of bytes in memory

Question: But how do we store several bytes in memory?

w=32 (bits)

- For example, let's store these 4 bytes starting at address 0x0000

MS 01000010 01101001 01110100 01110011_2 LSB

Answer:

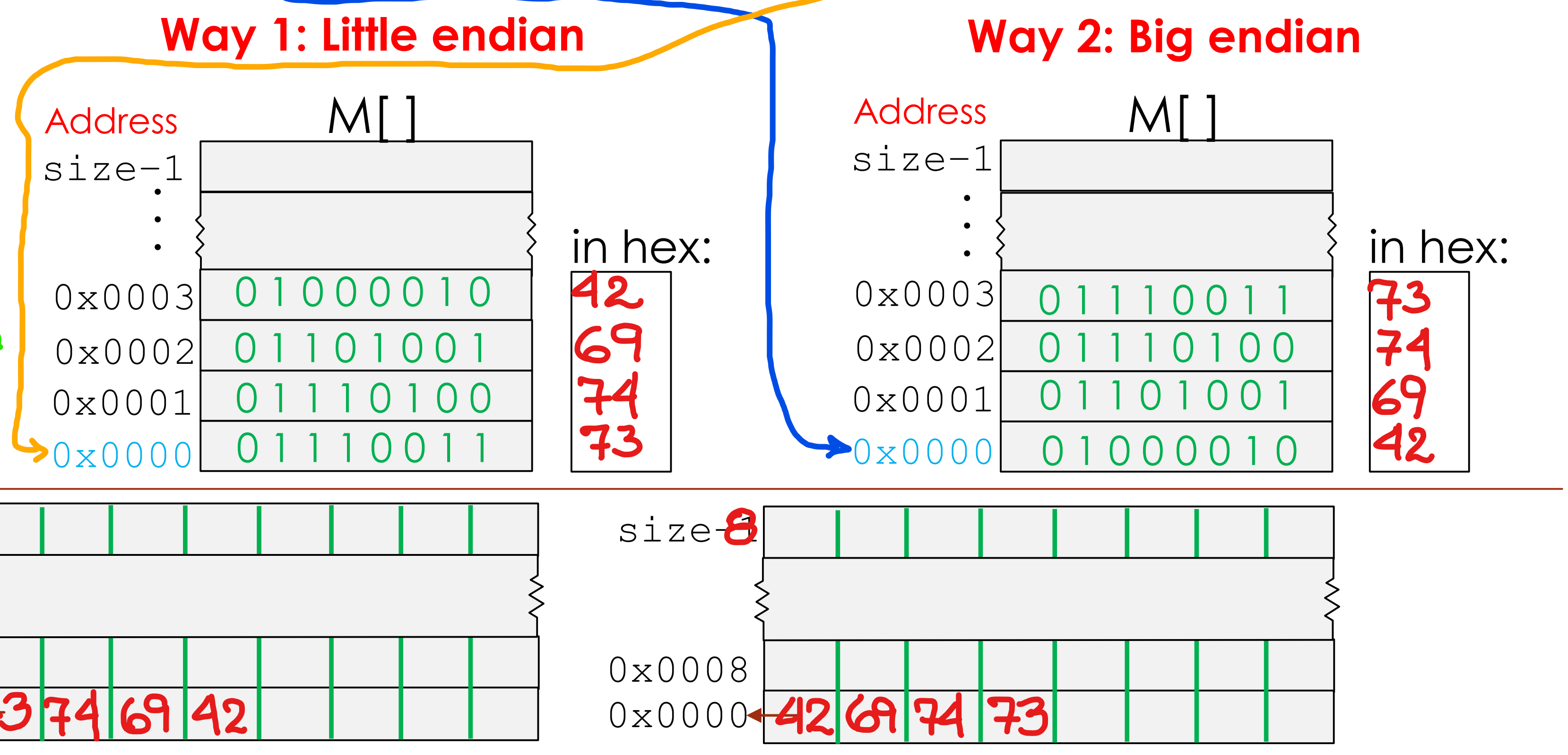
Trick: L L L

Little endian

LSB

Lowest address number

HW1



Way 1: Little endian

Address	M[ ]	in hex:
size−1		
⋮		
0x0003	01000010	42
0x0002	01101001	69
0x0001	01110100	74
0x0000	01110011	73

Way 2: Big endian

Address	M[ ]	in hex:
size−1		
⋮		
0x0003	01110011	73
0x0002	01110100	74
0x0001	01101001	69
0x0000	01000010	42

Compressed view of memory

Address								
size−8								
⋮								
0x0008								
0x0000	73	74	69	42				

size−8								
⋮								
0x0008								
0x0000	42	69	74	73				

Fixed!

No matter what a series of bits represent, they can be manipulated using bit-level operations:
- Boolean algebra
- Shifting

Review

Bit Manipulation - Boolean algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0

- **AND** -> A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

- **OR** -> A | B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

- **NOT** -> ~A = 1 when A=0

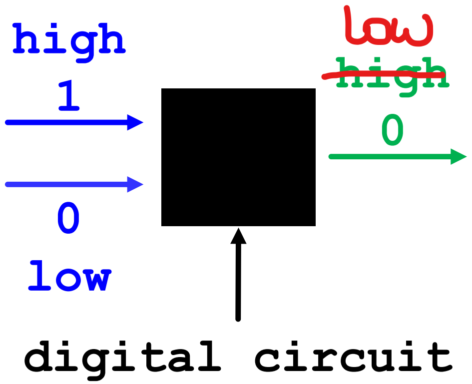
~	
0	1
1	0

- **XOR** (Exclusive-Or) -> A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

Interesting fact about Boolean algebra and digital logic

- Claude Shannon – 1937 master's thesis
- Made connection between Boolean algebra and digital logic
  - Boolean algebra could be applied to design and analysis of digital systems (digital circuits)
- Example:



=> we can describe it as an *AND gate*

Fixed!

Review HW2

Let's try some Boolean algebra!

- Operations applied bitwise -> to each bit
- Spot the error(s):

$$\begin{array}{r} 01101001_2 \\ \& \;\; 01010101_2 \\ \hline 01000001_2 \end{array} \qquad \begin{array}{r} 01101001_2 \\ | \;\; 01010101_2 \\ \hline 01111101_2 \end{array} \qquad \begin{array}{r} 01101001_2 \\ \wedge \;\; 01010101_2 \\ \hline 00111110_2 \end{array} \qquad \begin{array}{r} \\ \sim \;\; 01010101_2 \\ \hline 10101010_2 \end{array}$$

error!

Useful bit manipulations

- Using a binary mask (or bit mask) as an operand
  1. AND: Extracts particular bit(s) so we can test whether they are set

     i.e., set to **1**

     - Example:

       $$\begin{array}{rl} & 10110011_2 \\ \& & 00000001_2 \\ \hline & 00000001_2 \end{array}$$

       <- some value **x**

       <- binary mask

       The result tells us that the least significant bit (LSb) of **x** is set
  2. XOR: Toggle specific bits
     - Example:

       $$\begin{array}{rl} & 10110011_2 \\ \wedge & 00011100_2 \\ \hline & 10101111_2 \end{array}$$

       <- some value **x**

       <- binary mask

       We get a toggled version of the 3 original bits (of **x**) that correspond to the 3 set bits of the binary mask
- Using two operands
  1. OR: Merge all set bits of operands
     - Example:

       $$\begin{array}{rl} & 10110011_2 \\ | & 00011100_2 \\ \hline & 10111111_2 \end{array}$$

       <- some value **x**

       <- some value **y**

       The result contains all the set bits of **x** and **y**

Bit Manipulation - Shift operations

- Left Shift: `x << y`
  - Shift bit vector **x** left **y** positions
    - Effect:
      - Throw away **y** most significant bits (MSb) of **x** on left
      - Fill **x** with **y** `0`'s on right

a series of bits

- Right Shift: `x >> y`
  - Shift bit vector **x** right **y** positions
    - Effect:
      - Throw away **y** least significant bits (LSb) of **x** on right
    - Logical shift: Fill **x** with **y** `0`'s on left
    - Arithmetic shift: Fill **x** with **y** copies of **x**'s sign bit on left
      - Sign bit: most significant bit (MSb) of **x** (before shifting occurred)

- LSb: least significant bit is the rightmost bit of a series of bits (or bit vector)
- MSb: most significant bit is the leftmost bit of a series of bits (or bit vector)

HW 3

Bit Manipulation - Shift operations – Let's try!

- Left Shift: $10111001_2 << 4$

answer: 10010000_2

- Left Shift: $10111001_2 << 2$

answer: 11100100_2

- Right Shift: $00111001_2 >> 4$

logical

answer: $0000\ 0011_2$

sign bit (MSb)

- Right Shift: $\underline{1}0111001_2 >> 4$

arithmetic

answer: 11111011_2

- Right Shift: $\underline{1}0111001_2 >> 2$

logical and arithmetic

answers: 00101110_2

11101110_2

Summary

- Von Neumann architecture
  - Architecture of most computers
  - Its components: CPU, memory, input and ouput, bus
  - One of its characteristics: Data and code (programs) both stored in memory
- A look at memory: defined *byte-addressable* memory, diagram of (compressed) memory
  - Word size (w): size of a series of bits (or bit vector) we manipulate, also size of machine words (see Section 2.1.2)
- A look at bits in memory
  - Why binary numeral system (0 and 1 -> two values) is used to represent information in memory
  - Algorithm for converting binary to hexadecimal (hex)
    1. Partition bit vector into groups of 4 bits, starting from right, i.e., least significant byte (LSB)
       - If most significant "byte" (MSB) does not have 8 bits, pad it: add 0's to its left
    2. Translate each group of 4 bits into its hex value
  - What do bits represent? Encoding scheme gives meaning to bits
  - Order of bytes in memory: little endian versus big endian
- Bit manipulation – regardless of what bit vectors represent
  - Boolean algebra: bitwise operations => **AND** (`&`), **OR** (`|`), **XOR** (`^`), **NOT** (`~`)
  - Shift operations: left shift, right logical shift and right arithmetic shift
    - Logical shift: Fill x with y 0's on left
    - Arithmetic shift: Fill **x** with **y** copies of **x**'s sign bit on left
    - Sign bit: Most significant bit (MSb) before shifting occurred

NOTE:
C logical operators and C bitwise (bit-level) operators behave differently!
Watch out for `&&` versus `&`, `||` versus `|`, …

Next Lecture

- Representing data in memory – Most of this is review
  - "Under the Hood" - Von Neumann architecture
  - Bits and bytes in memory
    - How to diagram memory -> Used in this course and other references
    - How to represent series of bits -> In binary, in hexadecimal (conversion)
    - What kind of information (data) do series of bits represent -> Encoding scheme
    - Order of bytes in memory -> Endian
  - Bit manipulation – bitwise operations
    - Boolean algebra + Shifting
- Representing integral numbers in memory
  - Unsigned and signed
  - Converting, expanding and truncating
  - Arithmetic operations
- Representing real numbers in memory
  - IEEE floating point representation
  - Floating point in C – casting, rounding, addition, …