# Advanced Entry DT228/DT211

Logic Gates & Microprocessors

## Negative numbers in binary

- Uses a system called two's-complement
- Most-significant bit (leftmost as written) becomes the sign bit
  - 0 means positive
  - 1 mean negative
- You must first know the size (width) of the binary representation

## Converting to a negative number

- Consider decimal -56. What is this as an 8-bit binary representation?

  Step 1: Convert to binary in 8 bits

  00111000

  Step 2: Invert all the bits (one's complement)

  11000111

  Step 3: Add 1

  11001000

## Dual representation of numbers

- So how do you know, when you see a binary number, if it's positive or negative?
- The sign bit only tells you if you know it's a two's complement number
- That means that you need to know which it it in advance
- So what are the two possible interpretations of -56 in binary?

## Logical operations in binary

- In addition to normal arithmetic operations, binary also allows of logical operations
- These are at the heart of how computers work
- The basic logical operations are
  - AND
  - OR
  - NOT

## The AND operation

- X **AND** Y is true only if X is true <u>and</u> Y is true
- Truth table

| **X** | **Y** | **X AND Y** |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | F |

## The OR operation

- X **OR** Y is true only if X is true <u>or</u> Y is true
- Truth table

| **X** | **Y** | **X OR Y** |
|---|---|---|
| T | T | T |
| T | F | T |
| F | T | T |
| F | F | F |

## The NOT operation

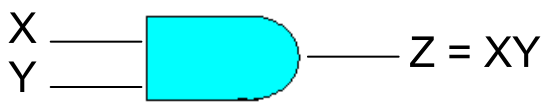
- NOT X is true only if X is false and NOT X is false only if X is true
- Truth table

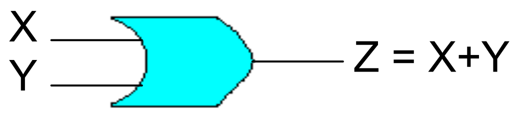
| **X** | **NOT X** |
|---|---|
| T | F |
| F | T |

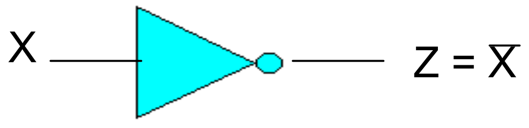
# Logic gates

Logic Gates:

- Logic gates are electronic circuits that operate on one or more input signals to produce an output signal.
- The following represent the graphic symbols used to designate the three types of gates: AND, OR and NOT.







## Boolean Algebra

- Named after the English mathematician George Boole who released a book in 1854 introducing the mathematical theory of logic.

- We can use Boolean algebra to simplify Boolean logic and consequently logic design.

## Building truth tables

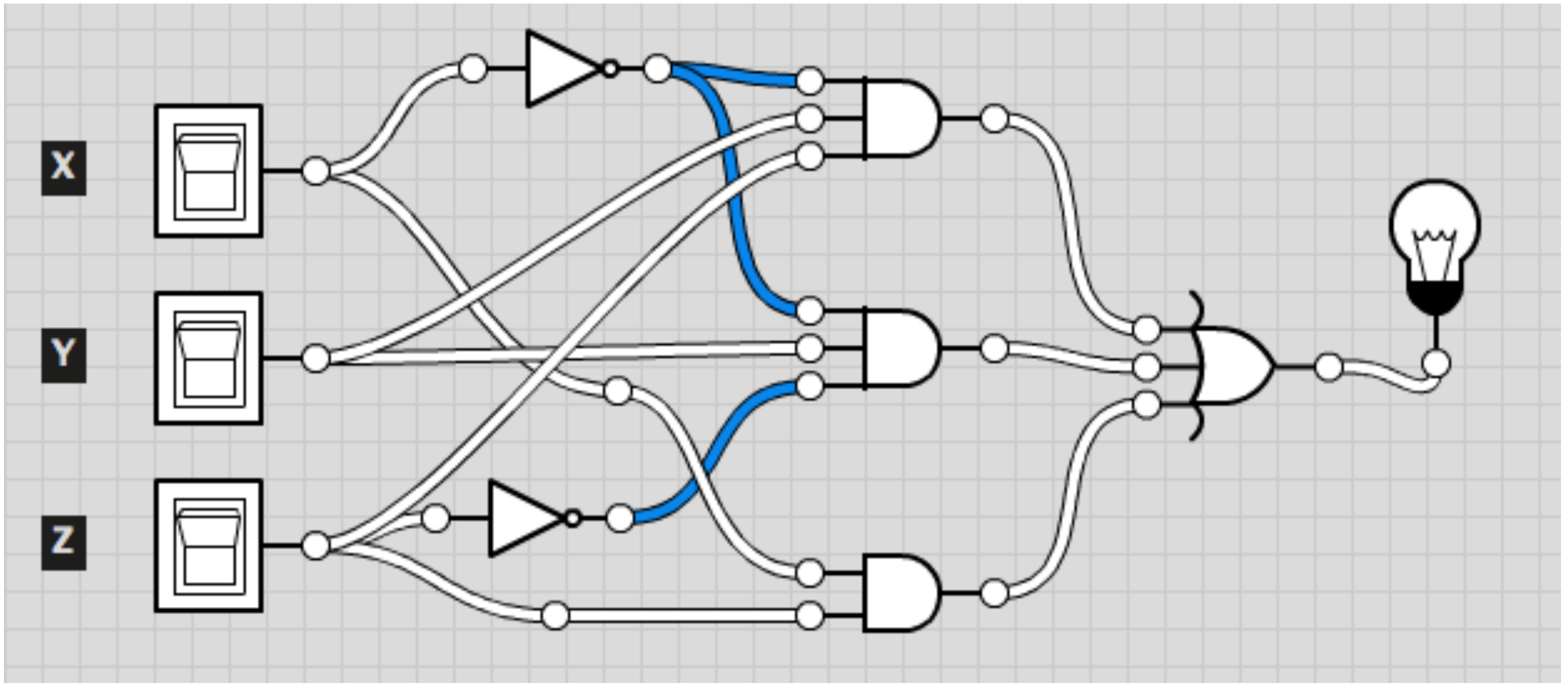
- The number of rows and columns is a function of the number of variables in the Boolean expression
- Example:
  - F = P AND R OR NOT Q
- Number of columns (ncols) needed is equal to the number of independent variables
- Number of rows needed is:
  - $2^{ncols}$
  - E.g. 3 variables require $2^3 = 8$ rows

## Simplifying circuits using algebra

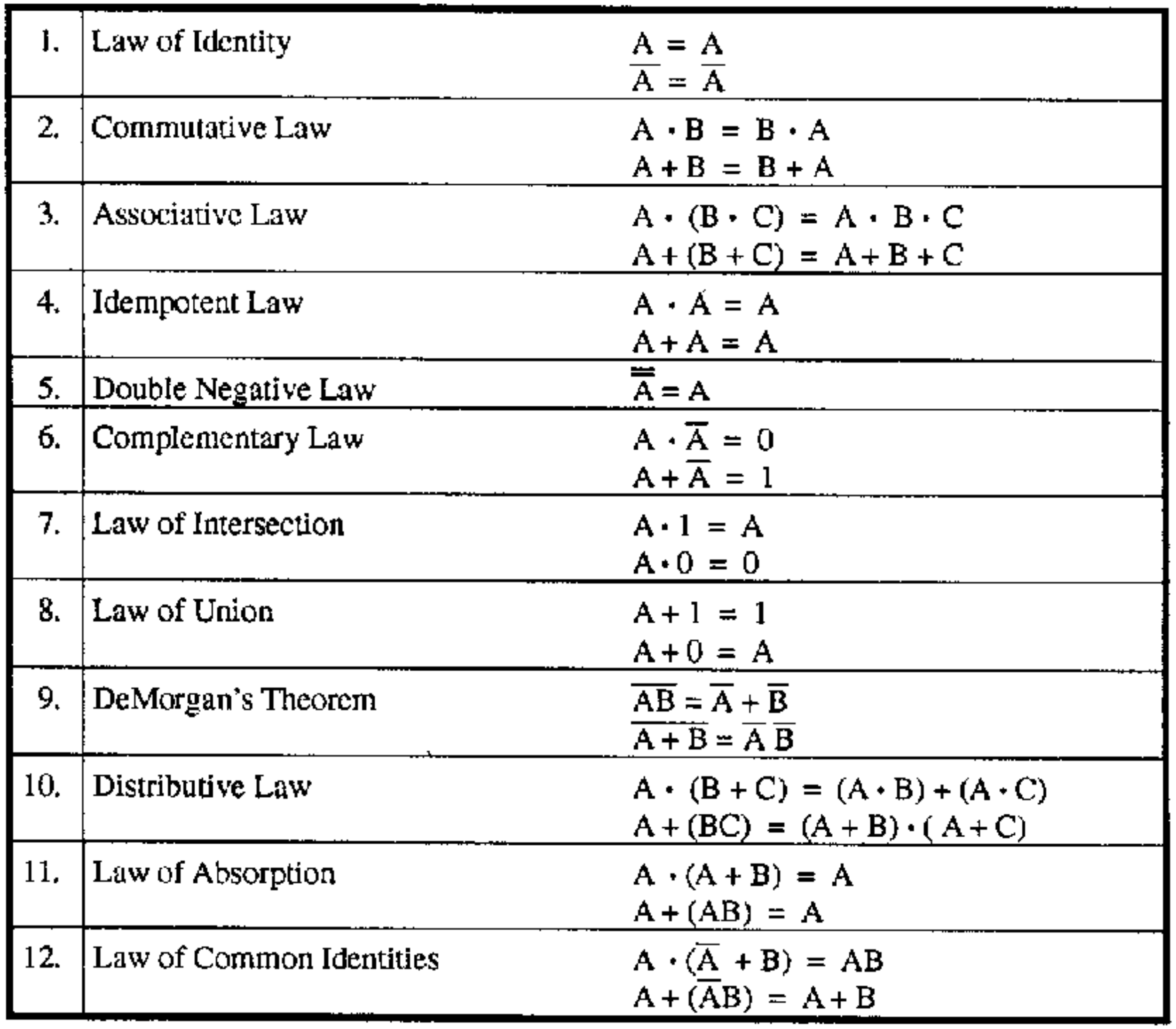
Consider the following circuit given as:

`F = ~X.Y.Z + ~X.Y.~Z + X.Z`

Can this be simplified?



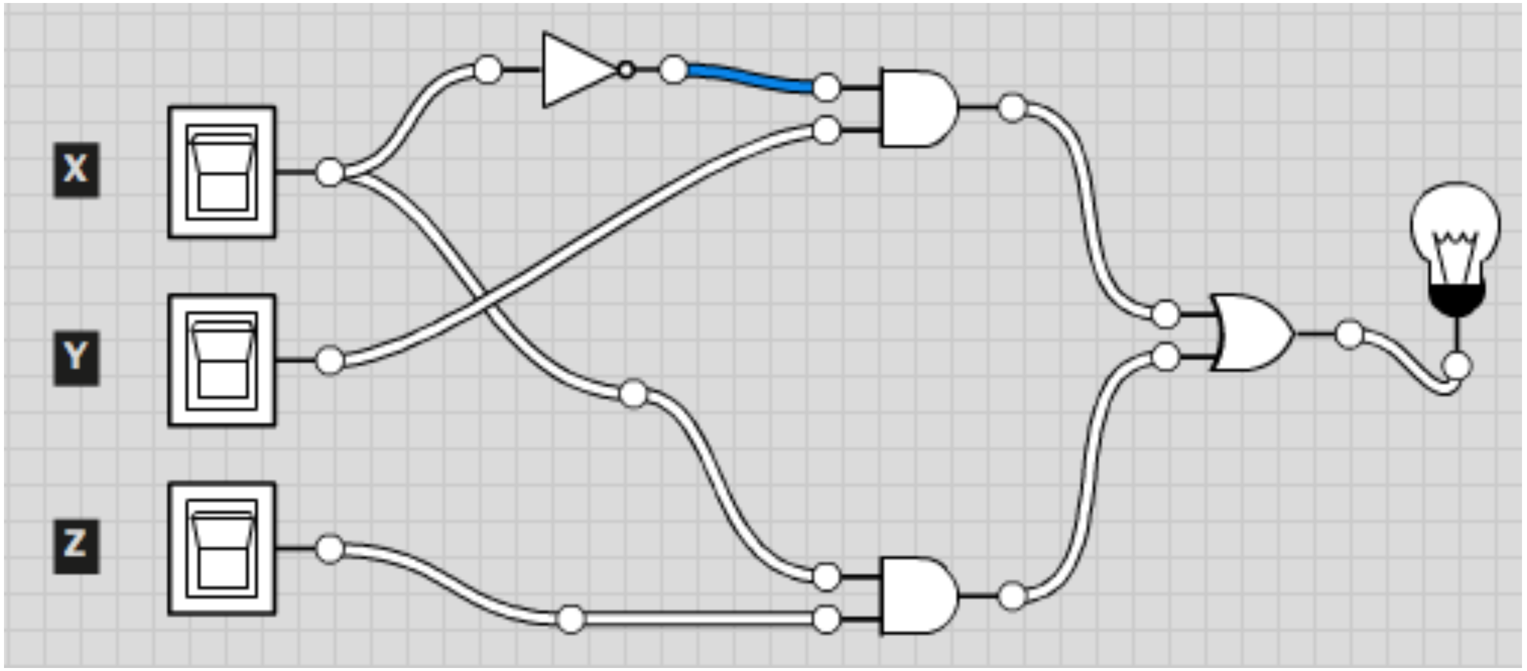
# Recall our Boolean Identities



| | | |
|---|---|---|
| 1. | Law of Identity | $A = A$<br>$\overline{A} = \overline{A}$ |
| 2. | Commutative Law | $A \cdot B = B \cdot A$<br>$A + B = B + A$ |
| 3. | Associative Law | $A \cdot (B \cdot C) = A \cdot B \cdot C$<br>$A + (B + C) = A + B + C$ |
| 4. | Idempotent Law | $A \cdot A = A$<br>$A + A = A$ |
| 5. | Double Negative Law | $\overline{\overline{A}} = A$ |
| 6. | Complementary Law | $A \cdot \overline{A} = 0$<br>$A + \overline{A} = 1$ |
| 7. | Law of Intersection | $A \cdot 1 = A$<br>$A \cdot 0 = 0$ |
| 8. | Law of Union | $A + 1 = 1$<br>$A + 0 = A$ |
| 9. | DeMorgan's Theorem | $\overline{AB} = \overline{A} + \overline{B}$<br>$\overline{A + B} = \overline{A}\,\overline{B}$ |
| 10. | Distributive Law | $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$<br>$A + (BC) = (A + B) \cdot (A + C)$ |
| 11. | Law of Absorption | $A \cdot (A + B) = A$<br>$A + (AB) = A$ |
| 12. | Law of Common Identities | $A \cdot (\overline{A} + B) = AB$<br>$A + (\overline{A}B) = A + B$ |

## Reduction process

```
F = ~X.Y.Z + ~X.Y.~Z + X.Z
F = ~X.Y(Z + ~Z) + X.Z                (identity 10)
F = ~X.Y.1 + X.Z                      (identity 6)
F = ~X.Y + X.Z                        (identity 7)
```



## Class exercise 1

- PART A
  - Let F = `~X.Y.Z + ~X.Y.~Z + X.Z`
  - Let G = `~X.Y + X.Z`
  - Prove that functions F and Q are equivalent

- PART B
  - Draw the circuit diagram for the expression below
  - Reduce the expression to a simpler form

  `( A + C ).( A.D + A.~D) + A.C + C`

## The XOR operation

- P **XOR** Q is *true* only if P is *true* and Q is *false* or P is *false* and Q is *true*
- What would P XOR P always yield?

| **P** | **Q** | **P XOR Q** |
|---|---|---|
| T | T | F |
| T | F | T |
| F | T | T |
| F | F | F |

## Motivating example of XOR

- The XOR operation is an extremely useful operation in computer science as you will see on this module
- In cryptography, the XOR operation can be used to implement a simple but potentially powerful encryption scheme called a *perfect cipher*.
- A perfect cipher is one which is theoretically unbreakable
- But perfect ciphers are not easy to implement in practice

## Encrypting a message

Suppose that Alice has some 32 byte plaintext message *M* that she wants to send securely to Bob. By some prior arrangement, Alice and Bob already know some shared secret key *K*, also 32 bytes in length. Alice can encrypt M by generating a ciphertext C as follows:

```
    C = M XOR K
```

This produces a ciphertext of 32 bytes in length where each bit $C_i$ of C is the XOR of $M_i$ and $K_i$ for all *i* in the range $0 <= i <= 255$.

Alice now sends the message C to Bob.

The question is: Having received C, what does Bob need to do to reveal the original message M, given that he knows K. Why does this work?

## Decrypting a message

- XOR is directly reversible such that for any Boolean P and Q, then:
  - **`P = (P XOR Q) XOR Q`**
  - **`Q = (P XOR Q) XOR P`**

- Therefore, because Bob already knows the value of K, he can recover M (the plaintext) as follows:
  - **`M = C XOR K`**

## Perfection not actually practical

- The XOR cipher is an example of a stream cipher in that it operates on one bit at a time
- Also called a one-time pad
- It's cryptographic strength relies on the key K
    - Being unpredictable (i.e. truly random)
    - Not being reused again
- In practice it is difficult to have two parties agree on a truly synchronised, random source of bits (to generate K)
- Instead, we use symmetric and asymmetric block ciphers with strong algorithmic encryption tecniques

## Bit-shifting

- In addition to the Boolean operations such as AND, OR, XOR, etc, there are two other notable operations that are very useful to know
- Left-shift causes all of the bits of a binary value to move to the left by one place, with the low-order bit being set to zero, e.g.

```
left-shift(00101111) = 01011110
```

## Bit-shifting (cont'd)

- Unsigned right-shift causes all of the bits of a binary value to move to the right by one place, with the high-order bit being set to zero, e.g.

  ```
  unsigned-right-shift(00101111) = 00010111
  ```

- Signed right-shift causes all of the bits of a binary value to move to the right by one place, with the high-order bit being set to original sign bit value, e.g.

  ```
  signed-right-shift(00101111) = 00010111
  signed-right-shift(10101111) = 11010111
  ```

## Visualising Boolean logic in Python

- The Python language has built-in Boolean operators
- In high level languages, like Python, these generally break down into logical and bitwise operators and functions
- Logical operators operate on the language-specific definition of *truthiness* and *falsiness* (not portable between languages)

```
and, or, not
```

- Bitwise operators operate on binary quantities (very portable)

```
&, |, ^, ~, <<, >>
```

## Class exercise 2

Answer the following, confirming your answers through Python:

- What are Python's truthy and falsy values?
- What is 0x0f bitwise-AND 0x06?
- What is 0x0f bitwise-OR 0x70?
- How do you create the number 8 from the number 1 and a shift operator
- How do you create the number 16 from the number 64 and a shift operator?

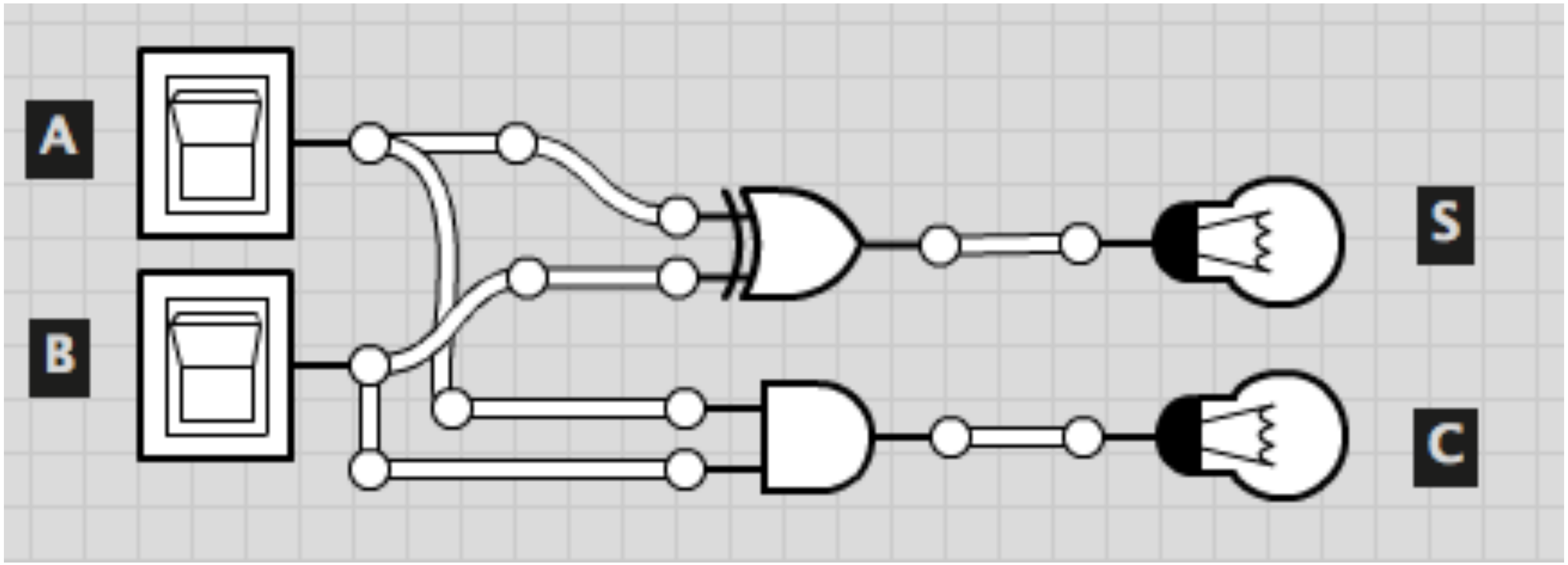
## Implementing basic arithmetic

- Logic gates can be coupled together to create simple arithmetic functions such as addition and multiplication
- Complex, multi-bit operations are built upon large networks of simple operations
- Let's consider the implementation two-bit addition in two steps
    - Half adder (2 logic gates)
    - Full adder (5 logic gates)

## Half adder

Given two input bits A and B, the sum and carry are given as:

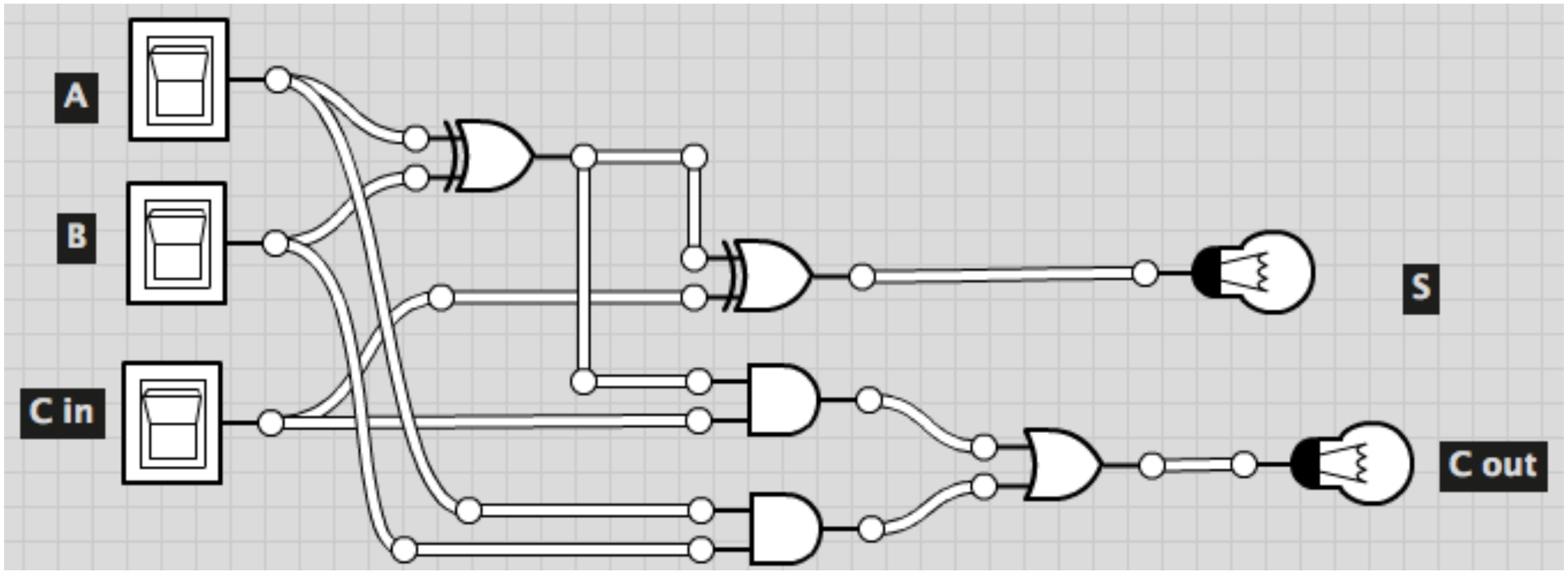
```
S = A XOR B
C = A AND B
```



## Full adder

Takes account of possible carry from a previous step:

```
S = (A XOR B) XOR Cin
Cout = (A AND B) OR (Cin AND (A XOR B))
```



## Programs and data

- Recall the Von Neumann innovation which merged the representations of program instructions and program data into a single memory device
- Hence, program instructions are also just data, but data with a special meaning to the processor
- The same relationship, albeit at a higher, level exists with a language like Python
- The program text is just data but it has special meaning to the Python interpreter

## Microprocessor Instructions

- You've seen how logic gates can be combined to construct higher level primitives such as single bit arithmetic
- In practice, modern microprocessors operate on multiple bits simultaneously
- Depending on the CPU architecture, this can be 8, 16, 32 or 64-bit quantities
- All CPUs provide hardware implementations of basic instructions like arithmetic and logical operations
- Instructions specify the operation to perform and zero or more associated operands

## Addressing modes

- CPU instructions access operands using a variety of *addressing modes* depending on the type of instruction and CPU architecture.
- Typical architectures support the following (know by different names on different architectures):
  - None - The instruction has no operands (e.g. NOP)
  - Implied - The operand is implied by the instruction (e.g. RET, POP)
  - Register - The operands are in one more registers (e.g. ADD, MOV)
  - Immediate - The literal operand value is encode
  - Indirect - the address of an operand is in a register or memory location
  - Double indirect - the address of the address of an operand is in a register or memory location
- Memory addresses can be absolute or relative

## Absolute vs relative addresses

- Absolute addressing means that the address of a location is its fully specified, unique location in memory
- Relative addressing means that the address is specified as some offset (number of bytes) from the calling instructure
    - Can be forwards or backwards
    - Usually there is some limit on the size of the offset, e.g. $2^{16}$
- Relative addressing is also usually requires fewer bytes to encode

## Fetch-Decode-Execute Cycle

- Describes how instructions are executed
- Although implemented differently between different CPU architectures the fetch-decode-execute model has remained largely conceptually unchanged for decades
- It refers to the process by which a processor executes instructions in memory
- To understand how this works in detail, let's consider the principal registers and circuits in a typical CPU

## Microprocessor registers

- Conceptually, a microprocessor operates using the following registers and circuits
  - Program counter (PC)
  - Memory address register (MAR)
  - Memory data register (MDR)
  - Instruction register (IR)
  - Control unit (CU)
  - Arithmetic logic unit (ALU)

## Program counter

- Holds the address (in memory) of the next instruction to be executed
- Initialised to a default value when the processor starts up
    - Typically to an start address of an OS boot loader in read-only memory
- Called the *instruction pointer* (IP) in Intel processor family
- Advances by the size (in bytes) of each instruction as they are executed

## Memory address register

- Holds the address of the memory location to be accessed (read from or written to)
- The address of any value being read from or written to in memory must first be loaded into the MAR before a processing operation can be effected

## Memory data register

- Register which holds data just fetched from memory and waiting to be processed by the CPU or data which has been processed by the CPU and is waiting to be stored back to memory

## Instruction register

- Temporary register location which holds the code for the next instruction to be executed by the CPU
- Depending on the CPU architecture, the binary instruction code (also known as an opcode) may be a fixed length instruction (RISC) or variable length instruction (CISC)

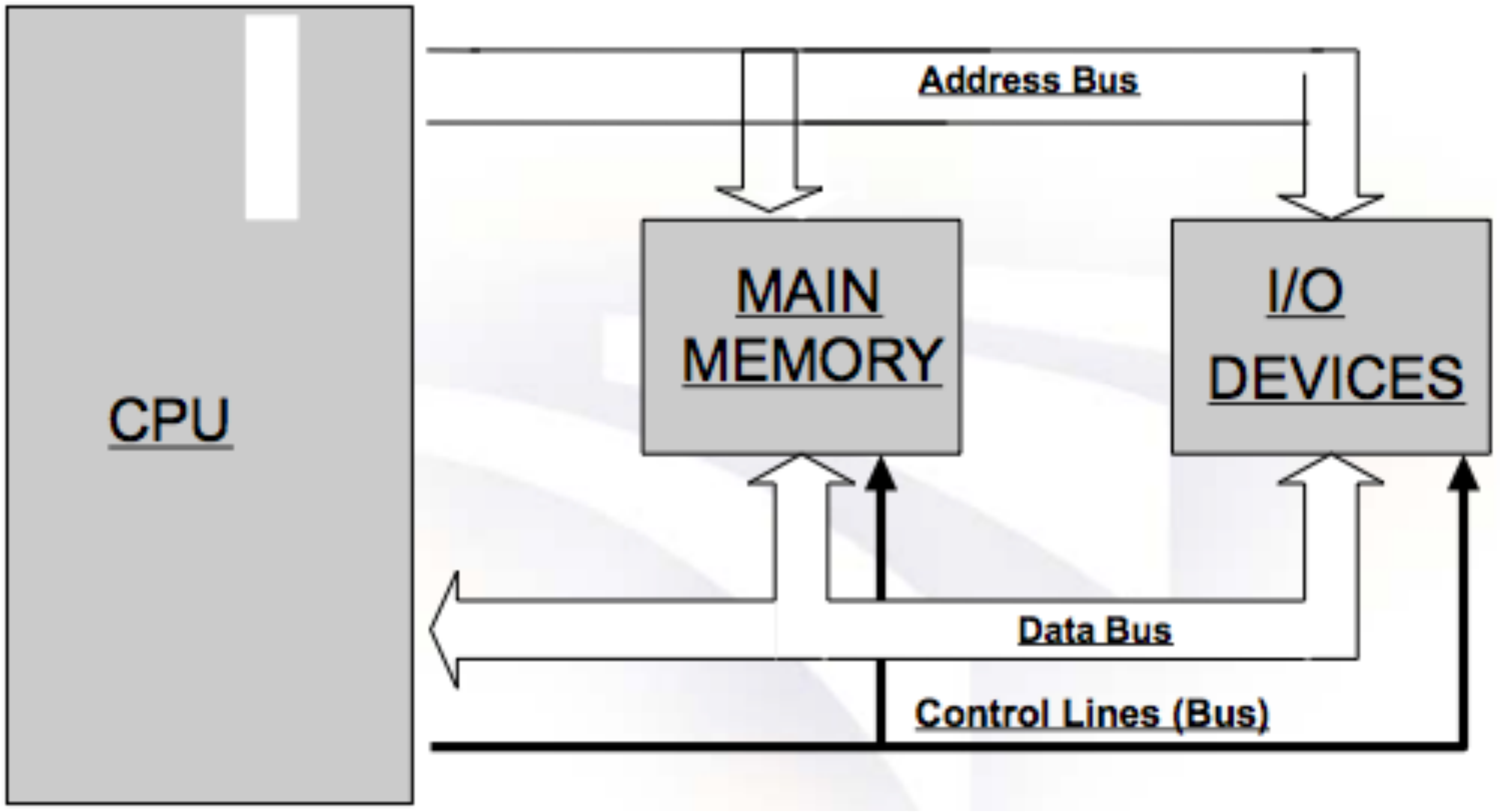
## Control unit

- Responsible for decoding the instruction in the IR
- Selects machine resources such as a data source register and a particular arithmetic operation
- Coordinates execution process

## Arithmetic logic unit

- Performs the mathematical operations on integers and floating point numbers

## Simplified microprocessor



## Fetch

- The fetch stage loads the contents of the PC (IP) into the MAR
- The value at the location pointed to by the MAR is loaded into the MDR
- The contents of the MDR is loaded into the IR
- The PC is incremented past this instruction to the address of the next instruction to execute
- If an invalid address is present in the PC an error occurs and is handled (more later)

## Decode

- The decode stage interprets the instruction in the IR
- Typically an instruction encodes two things
- Firstly, what it is to do
  - E.g. Integer addition, memory move, etc
- Secondly (and optionally), what it is to operate on (i.e. the operands)
  - Operands can be immediate literals or the address (es) of memory locations containing data and addresses of data
- The decode stage generally fetches operand values

## Execute

- The execute stage does as you'd expect
- The CU and ALU (if applicable) process the instruction with its operands and stores the result (if applicable) to the specified location which could be another register or a memory location

## Encoding instructions

- As discussed, the decode step takes a binary value in the IR and decides what to do
- Instructions are encoded as binary strings of one more more bytes (variable length or fixed)
- These bytes are divided into bit fields, each having meaning in the context of a particular instructions
- Only some bit string combinations are valid instructions for any given CPU architecture

## Writing CPU code

- Nowadays programming is done in high level languages like C and Python which converts there instruction representations down into machine code to be executed directly on the microprocessor
- In theory you could just write valid binary values into memory and execute them as programs
- This is tedious and error prone
- Programming machine code is normally done through assembly language

## Assembly language

- Although machine code is the lowest level of CPU programs, the term assembly language is used in practice by programmers to express this lowest level of abstraction
- Assembly language is a set of mnemonics and syntax which translate directly into machine code and data
- The translation is done by an assembler
- Language compilers can produce assembly language which is then *assembled* into machine code

## The Intel (x86) instruction set

- Let's look at the Intel x86 architecture instructions as a case study

| **Assember Mnemonic** | **Description** |
|---|---|
| ADD, SUB, MUL, DIV | Arithmetic operations |
| AND, OR, NOT, XOR | Bitwise operations |
| CALL | Save the IP and jump to an address |
| RET | Restore the IP |
| JMP | Jump to an address (unconditionally) |
| CMP | Compare operands |

## x86 instruction set (cont'd)

| **Assember Mnemonic** | **Description** |
|---|---|
| Jcc | Conditional jump (following compare) |
| INC, DEC | Increment and decrement (by 1) |
| MOV | Move data between locations (registers and memory) |
| LEA | Load effective address (computed address) |
| NEG | Two complement negation |
| PUSH | Put data onto the top of stack, advancing stack pointer |
| POP | Remove data from top of stack, restoring stack pointer |
| ROR, ROL | Rotate right, left |
| SHR, SHL | Unsigned shift right, left |
| SAR, SAL | Signed shift right, left |

## Intel x86 Registers

- CPU local storage
- High speed read and write
- One copy of register set per core so have to be shared between all running programs
- Usage of particular registers by convention and by necessity (i.e. required by instructions)
- See also:
  - http://www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html

## The program stack

- The stack is a memory data structure which can be used, among other things, to temporarily store register values
- Stacks are abstract data structures having the following principal operations
    - PUSH - add an item to the top of stack
    - POP - remove the item at the current top of the stack
- Microprocessors make such use of the stack that there are hardware instructions for push and pop on most, if not all, architectures