Computer Organization and Levels of Abstraction

Announcements

- Today:
  - PS 7 (due now)
  - Lab 8: Sound Lab tonight – bring machines and headphones!
  - PA 7
- Tomorrow: Lab 9
- Friday: PS8

Today

- (Short) Floating point review
- Boolean logic
- Combinational Circuits
- Levels of Abstraction

Floating point

$1.0011101 \times 2^{01001011}$

+/-	Exponent	Mantissa
1 bit	8 bits	23 bits

- Sign is a 0 or 1
- Exponent is an binary integer
- Mantissa is a binary fraction

Floating point Sign

$1.0011101 \times 2^{\mathbf{01001011}}$

0_ ______ ______

+/-	Exponent	Mantissa
1 bit	8 bits	23 bits

- Sign is a 0 or 1

Exponent

$1.0011101 \times 2^{\mathbf{01001011}}$

- Exponent 01001011
- Is an **unsigned** integer
- But exponent can be negative – how to distinguish?
- IEEE-754 specifies a bias: 127
- This gives us a range of -126 to +127
- Makes comparison easier (for large and small values)

Floating point Mantissa

$1.0011101 \times 2^{\mathbf{01001011}}$

0_	11001010	0011101________________
+/-	Exponent	Mantissa
1 bit	8 bits	23 bits

- Pad the mantissa

Floating point Mantissa

$1.0011101 \times 2^{\mathbf{01001011}}$

0_	11001010	00111010000000000000000
+/-	Exponent	Mantissa
1 bit	8 bits	23 bits

- Pad the mantissa

Floating point Mantissa

$1.0011101 \times 2^{\mathbf{01001011}}$

011001010 00111010000000000000000

Boolean Logic

Conceptualizing bits and circuits

- **ON** or **1**: **true**
- **OFF** or **0**: **false**
- circuit behavior: expressed in *Boolean logic* or *Boolean algebra*

Boolean Logic (Algebra)

- Computer circuitry works based on Boolean Logic (Boolean Algebra) : operations on True (1) and False (0) values.

A	B	A ∧ B (A AND B) (conjunction)	A ∨ B (A OR B) (disjunction)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

A	¬A (NOT A) (negation)
0	1
1	0

- A and B in the table are Boolean variables, AND and OR are operations (also called functions).

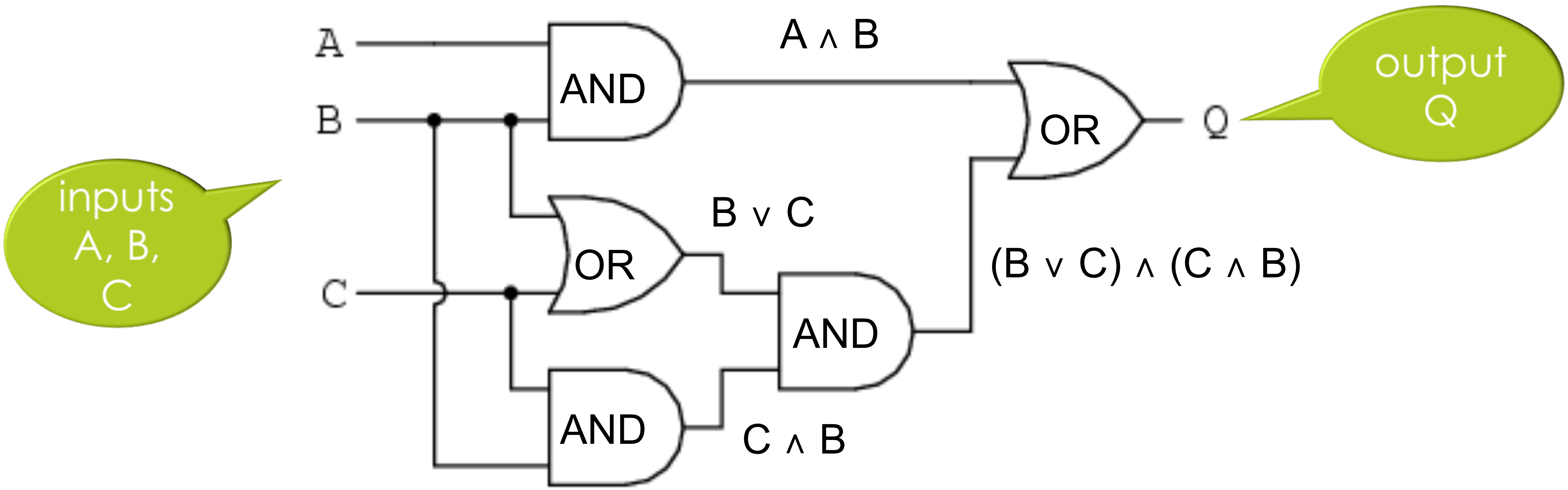
Logic Gates

- A gate is a physical device that implements a Boolean operator by performing basic operations on electrical signals.

logical picture of gates

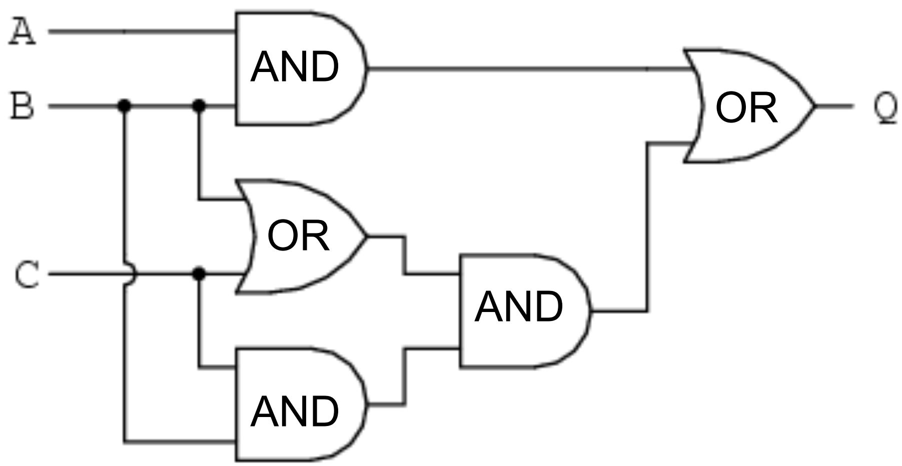
Combinational Circuits

The logic states of inputs at any given time determine the state of the outputs.



What is Q? $(A \wedge B) \vee ((B \vee C) \wedge (C \wedge B))$

Truth Table of a Circuit



Q = (A ∧ B) ∨ ((B ∨ C) ∧ (C ∧ B))

A	B	C	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Describes the relationship between inputs and outputs of a device

http://www.allaboutcircuits.com/vol_4/chpt_7/6.html

Describing Behavior of Circuits

- Boolean expressions
- Circuit diagrams
- Truth tables

Equivalent notations

Continued…

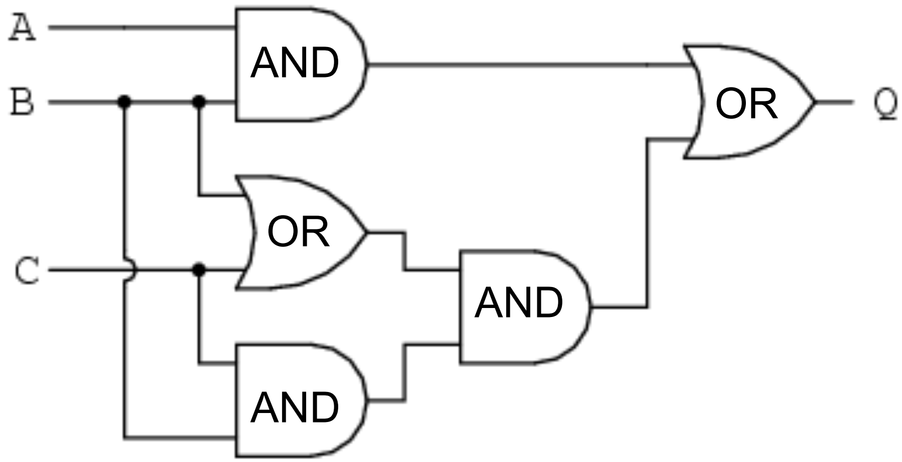
Manipulating circuits

Boolean algebra and logical equivalence

Why manipulate circuits?

- The design process
  - simplify a complex design for easier manufacturing, faster or cooler operation, …
- Boolean algebra helps us find another design guaranteed to have same behavior

Logical Equivalence

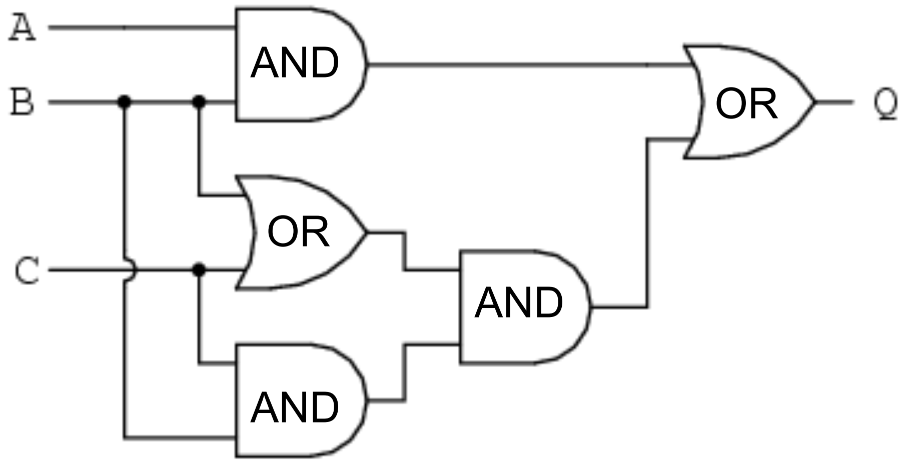


$Q = (A \wedge B) \vee ((B \vee C) \wedge (C \wedge B))$

A	B	C	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

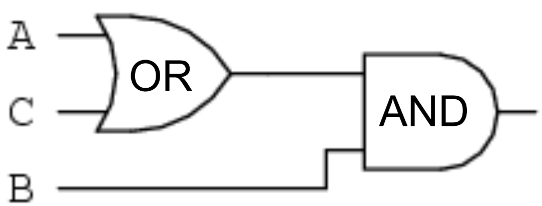
Can we come up with a simpler circuit implementing the same truth table?
Simpler circuits are typically cheaper to produce, consume less energy etc.

Logical Equivalence



$Q = (A \wedge B) \vee ((B \vee C) \wedge (C \wedge B))$

A	B	C	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



$Q = B \wedge (A \vee C)$

This smaller circuit is logically equivalent to the one above: they have the same truth table. By using **laws of Boolean Algebra** we convert a circuit to another equivalent circuit.

Laws for the Logical Operators $\wedge$ and $\vee$ (Similar to $\times$ and $+$)

- Commutative: $A \wedge B = B \wedge A$ $\quad$ $A \vee B = B \vee A$
- Associative: $A \wedge B \wedge C = (A \wedge B) \wedge C = A \wedge (B \wedge C)$
  $A \vee B \vee C = (A \vee B) \vee C = A \vee (B \vee C)$
- Distributive: $A \wedge (B \vee B) = (A \wedge B) \vee (A \wedge C)$
  $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$
- Identity: $A \wedge 1 = A$ $\quad$ $A \vee 0 = A$
- Dominance: $A \wedge 0 = 0$ $\quad$ $A \vee 1 = 1$
- Idempotence: $A \wedge A = A$ $\quad$ $A \vee A = A$
- Complementation: $A \wedge \neg A = 0$ $\quad$ $A \vee \neg A = 1$
- Double Negation: $\neg \neg A = A$

Laws for the Logical Operators $\wedge$ and $\vee$ (Similar to × and +)

- Commutative: $A \wedge B = B \wedge A$ $\quad$ $A \vee B = B \vee A$
- Associative: $A \wedge B \wedge C = (A \wedge B) \wedge C = A \wedge (B \wedge C)$
  $A \vee B \vee C = (A \vee B) \vee C = A \vee (B \vee C)$
- Distributive: $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$
  $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$ ← Not true for + and ×
- Identity: $A \wedge 1 = A$ $\quad$ $A \vee 0 = A$

The A's and B's here are schematic variables! You can instantiate them with any expression that has a Boolean value:

$$\underbrace{(x \vee y)}_{A} \wedge \underbrace{z}_{B} = \underbrace{z}_{B} \wedge \underbrace{(x \vee y)}_{A} \quad \text{(by commutativity)}$$

Applying Properties for ∧ and ∨

Showing →	**$(x \wedge y) \vee ((y \vee z) \wedge (z \wedge y)) = y \wedge (x \vee z)$**
Commutativity $A \wedge B = B \wedge A$	$(x \wedge y) \vee ((z \wedge y) \wedge (y \vee z))$
Distributivity $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$	$(x \wedge y) \vee (z \wedge y \wedge y) \vee (z \wedge y \wedge z)$
Associativity, Commutativity, Idempotence	$(x \wedge y) \vee ((z \wedge y) \vee (y \wedge z))$
Commutativity, idempotence $A \wedge A = A$	$(y \wedge x) \vee (y \wedge z)$
Distributivity (backwards) $(A \wedge B) \vee (A \wedge C) = A \wedge (B \vee C)$	$y \wedge (x \vee z)$

Conclusion:

$$(x \wedge y) \vee ((y \vee z) \wedge (z \wedge y)) = y \wedge (x \vee z)$$

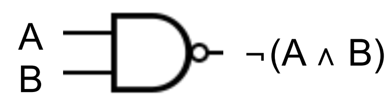
Extending the system

more gates and DeMorgan’s laws

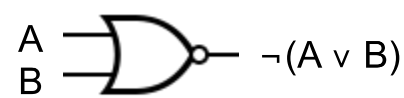
More gates (NAND, NOR, XOR)

A	B	A nand B	A nor B	A xor B
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	0

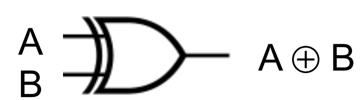
- **nand (“not and”)**: A nand B = not (A and B)



- **nor (“not or”)**: A nor B = not (A or B)



- **xor (“exclusive or”)**:
  A xor B = (A and not B) or (B and not A)



DeMorgan's Law

Nand: $\neg(A \land B) = \neg A \lor \neg B$

Nor: $\neg(A \lor B) = \neg A \land \neg B$

DeMorgan's Law

Nand: $\neg(A \wedge B) = \neg A \vee \neg B$

```
if not (x > 15 and x < 110):  ...
```

is logically equivalent to

```
if (not x > 15) or (not x < 110): ...
```

Nor: $\neg(A \vee B) = \neg A \wedge \neg B$

```
if not (x < 15 or x > 110): ...
```

is logically equivalent to

```
if (not x < 15) and (not x > 110): ...
```

A circuit for parity checking

Boolean expressions and circuits

A Boolean expression that checks parity

- 3-bit odd parity checker F: an expression that should be true when the count of 1 bits is odd: when 1 or 3 of the bits are 1s.

P =

A	B	C	P
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

A Boolean expression that checks parity

- 3-bit odd parity checker F: an expression that should be true when the count of 1 bits is odd: when 1 or 3 of the bits are 1s.

$$P = (\neg A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge \neg C) \vee (A \wedge \neg B \wedge \neg C) \vee (A \wedge B \wedge C)$$

A	B	C	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

There are specific methods for obtaining canonical Boolean expressions from a truth table, such as writing it as a disjunction of conjunctions or as a conjunction of disjunctions.

Note we have four subexpressions above each of them corresponding to exactly one row of the truth table where P is 1.

The circuit

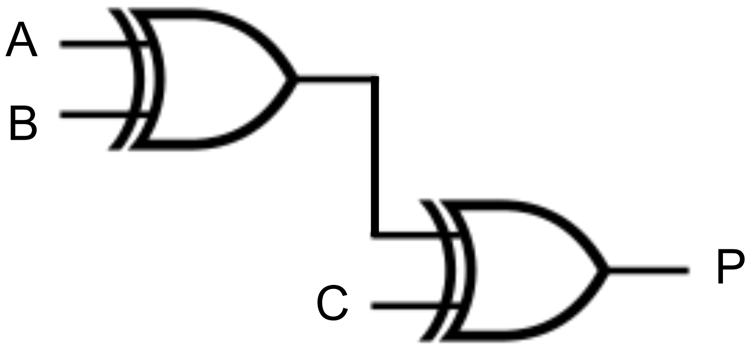
3-bit odd parity checker

P = (¬A ∧ ¬B∧ C) ∨ (¬A ∧ B ∧ ¬C) ∨ (A ∧ ¬B∧ ¬C) ∨ (A ∧ B ∧ C)

logically equivalent

A	B	C	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

P = (A ⊕ B) ⊕ C



Summary

You should be able to:

- Identify basic gates
- Describe the behavior of a gate or circuit using Boolean expressions, truth tables, and logic diagrams
- Transform one Boolean expression into another given the laws of Boolean algebra

Circuits for arithmetic

Adding Binary Numbers: 1 bit

+	0	1
0	0	1
1	1	10

A:	0	0	1	1
B:	0	1	0	1
	---	---	---	---
	0	1	1	1 0

Adding Binary Numbers: 1 bit

+	0	1
0	0	1
1	1	10

A:	0	0	1	1
B:	0	1	0	1
	---	---	---	---
	0	1	1	1 0

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Adding Binary Numbers: 1 bit

+	0	1
0	0	1
1	1	10

A:	0	0	1	1
B:	0	1	0	1
	---	---	---	---
	0	1	1	1 0

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Adding two 1-bit numbers without taking the carry into account

Adding Binary Numbers: 1 bit

+	0	1
0	0	1
1	1	10

A:	0	0	1	1
B:	0	1	0	1
	---	---	---	---
	0	1	1	1 0

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Adding two 1-bit numbers without taking the carry into account

What is a logical gate or boolean operation that does this?

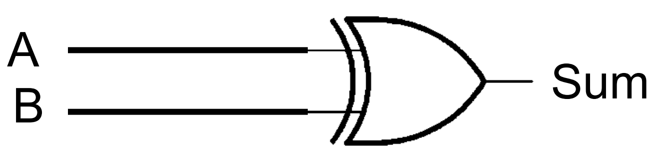
Adding Binary Numbers: 1 bit

+	0	1
0	0	1
1	1	10

A:	0	0	1	1
B:	0	1	0	1
	---	---	---	---
	0	1	1	1 0

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Adding two 1-bit numbers without taking the carry into account



Sum = A $\oplus$ B

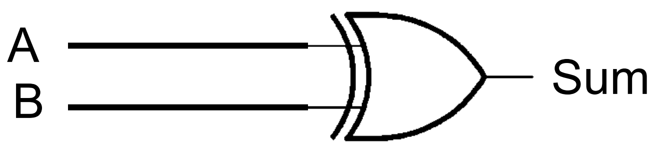
Adding Binary Numbers: 1 bit

+	0	1
0	0	1
1	1	10

A:	0	0	1	1
B:	0	1	0	1
	---	---	---	---
	0	1	1	1 0

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Adding two 1-bit numbers without taking the carry into account



$$\text{Sum} = A \oplus B$$

How can we handle the carry (out)?

Adding Binary Numbers: 1 bit

+	0	1
0	0	1
1	1	10

A:	0	0	1	1
B:	0	1	0	1
	---	---	---	---
	0	1	1	1 0

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Adding Binary Numbers: 1 bit

+	0	1
0	0	1
1	1	10

A:	0	0	1	1
B:	0	1	0	1
	---	---	---	---
	0	1	1	1 0

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

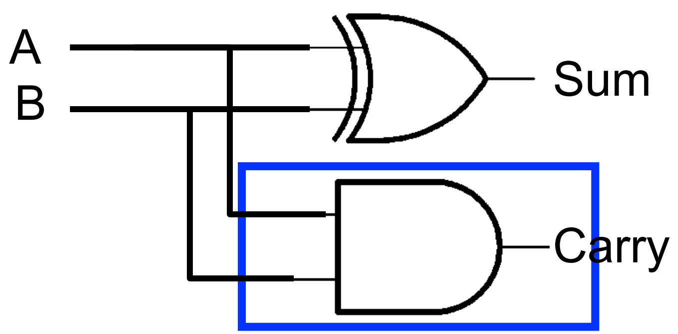
What is a logical gate or boolean operation that does this?

Adding Binary Numbers: 1 bit

+	0	1
0	0	1
1	1	10

A:	0	0	1	1
B:	0	1	0	1
	---	---	---	---
	0	1	1	1 0

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

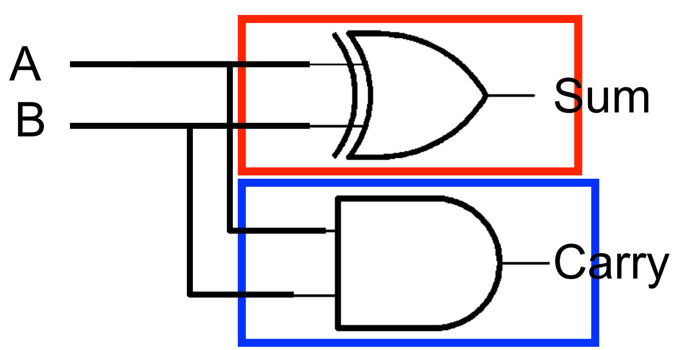


Adding Binary Numbers: 1 bit

+	0	1
0	0	1
1	1	10

A:	0	0	1	1
B:	0	1	0	1
	---	---	---	---
	0	1	1	1 0

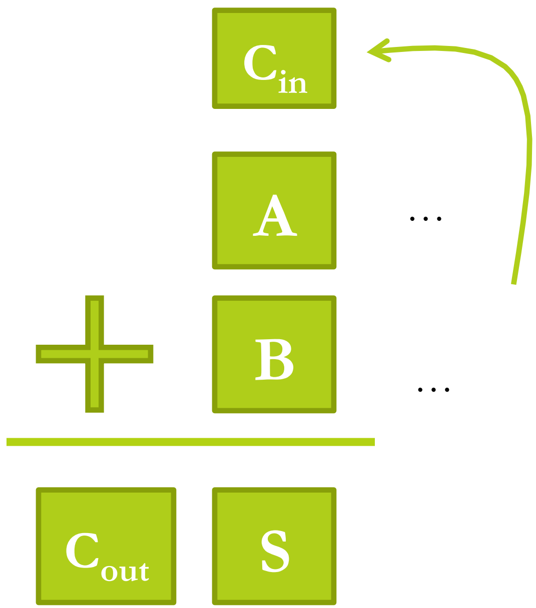
A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Half Adder:

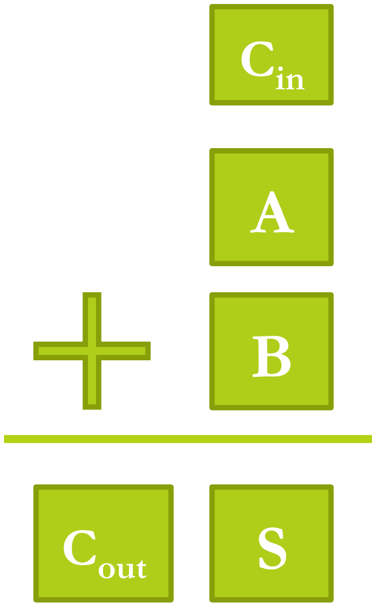
→ adds two single binary digits (1 bit each)

A Full Adder



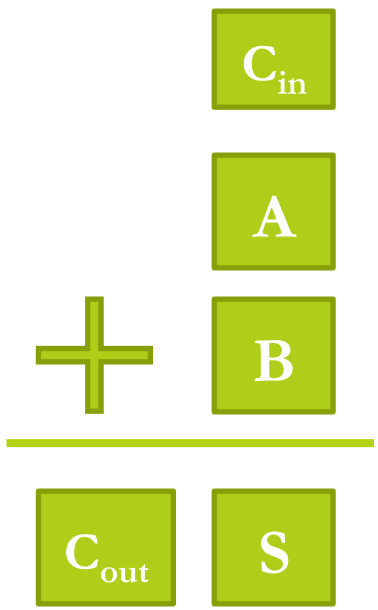
A	B	C_{in}	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

A Full Adder



A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

A Full Adder

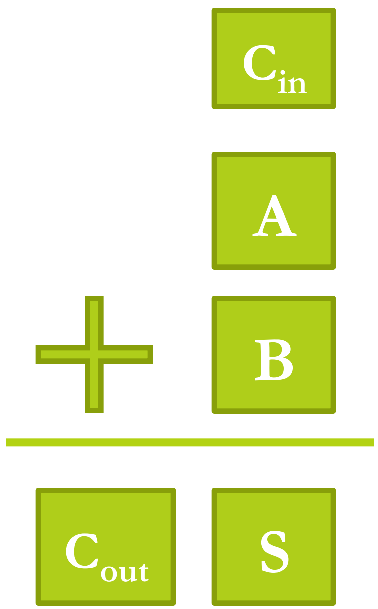


A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = ((A \oplus B) \wedge C_{in}) \vee (A \wedge B)$$

A Full Adder



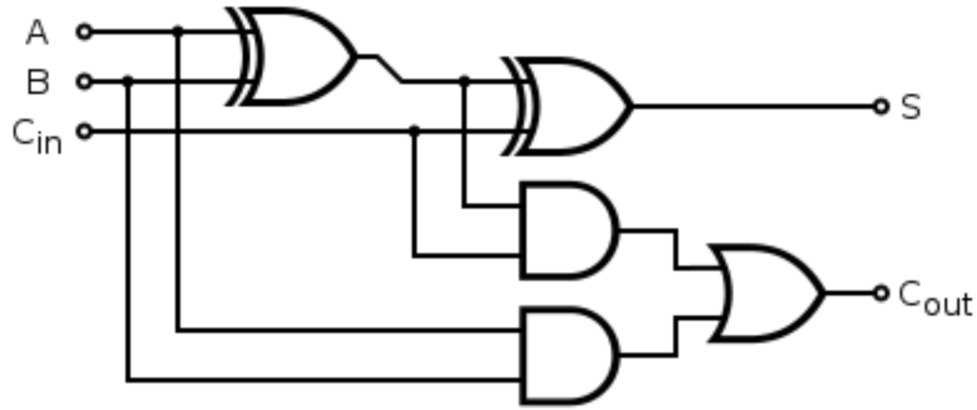
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S: 1 when there is an odd number of bits that are 1

C_{out} : 1 if both A and B are 1 or, one of the bits and the carry in are 1.

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = ((A \oplus B) \wedge C_{in}) \vee (A \wedge B)$$

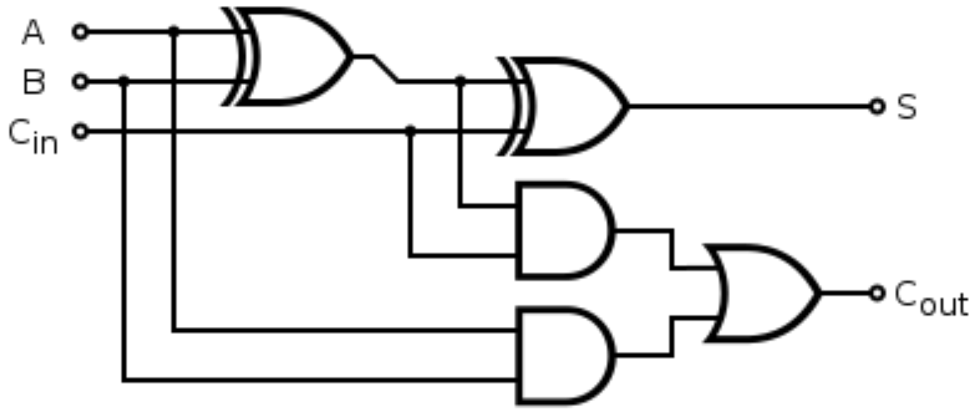
Full Adder (FA)



$S = A \oplus B \oplus C_{in}$

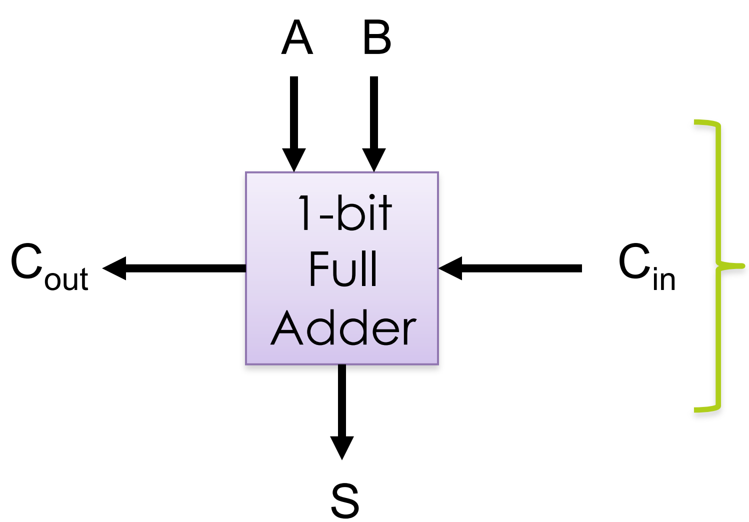
$C_{out} = ((A \oplus B) \wedge C_{in}) \vee (A \wedge B)$

Full Adder (FA)



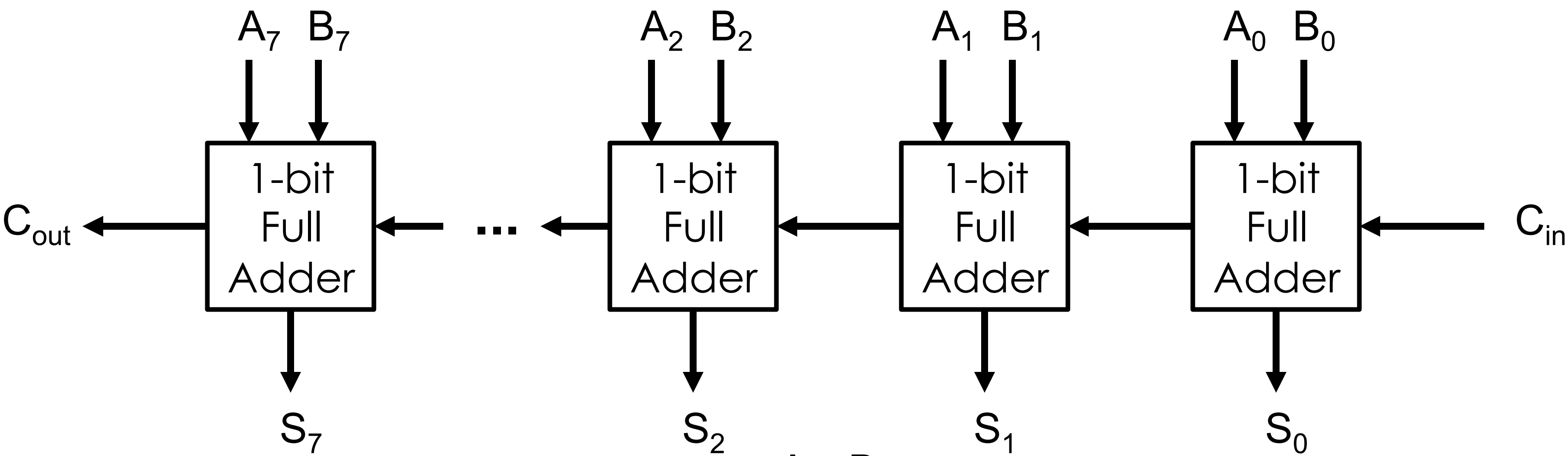
$S = A \oplus B \oplus C_{in}$

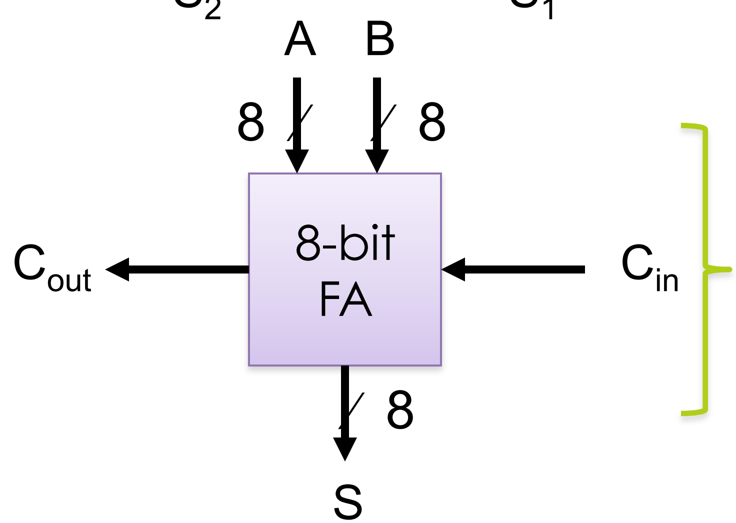
$C_{out} = ((A \oplus B) \wedge C_{in}) \vee (A \wedge B)$



More abstract representation of the above circuit. Hides details of the circuit above.

8-bit Full Adder





More abstract representation of the above circuit. Hides details of the circuit above.

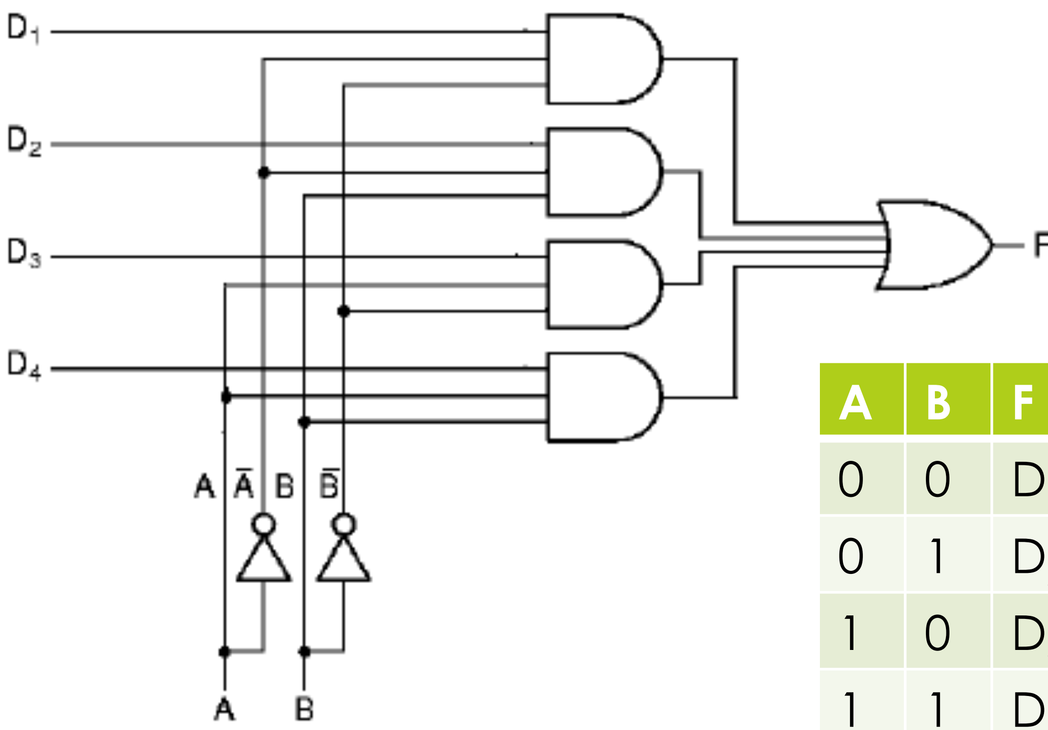
Control Circuits

- In addition to circuits for basic logical and arithmetic operations, there are also circuits that determine **the order in which operations are carried out and to select the correct data values to be processed**.

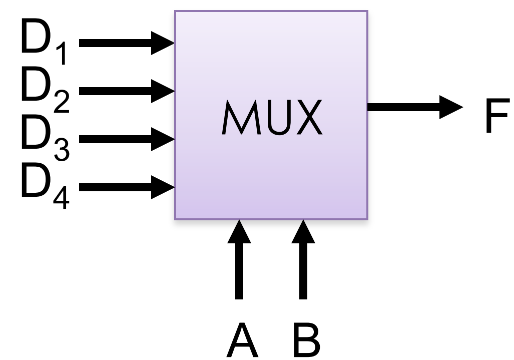
Multiplexer (MUX)

- A multiplexer chooses one of its inputs.

  2^n input lines, n selector lines, and 1 output line



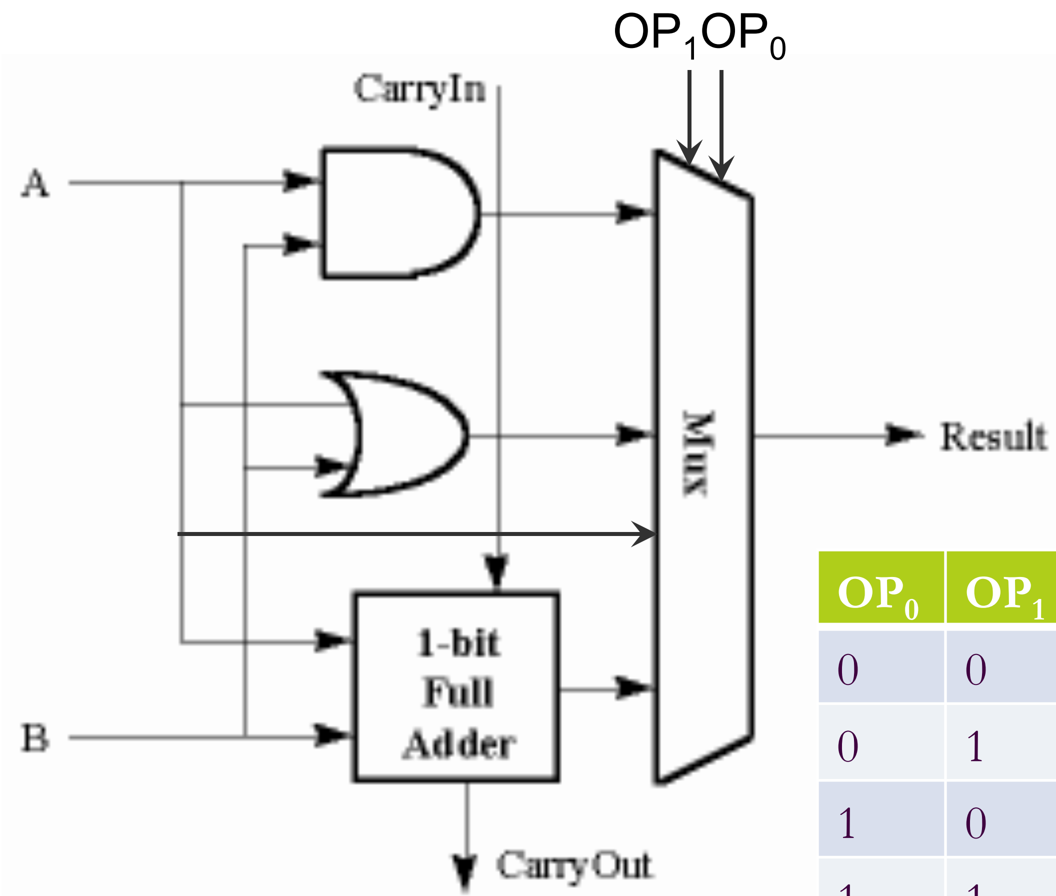
A	B	F
0	0	D_1
0	1	D_2
1	0	D_3
1	1	D_4

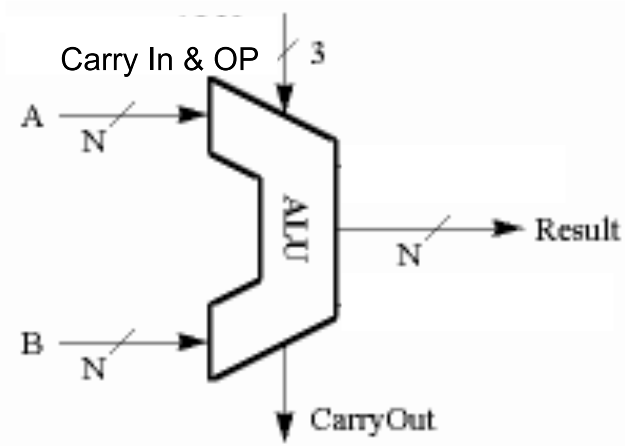


hides details of the circuit on the left

http://www.cise.ufl.edu/~mssz/CompOrg/CDAintro.html

Arithmetic Logic Unit (ALU)





OP_0	OP_1	F
0	0	$A \wedge B$
0	1	$A \vee B$
1	0	A
1	1	A + B

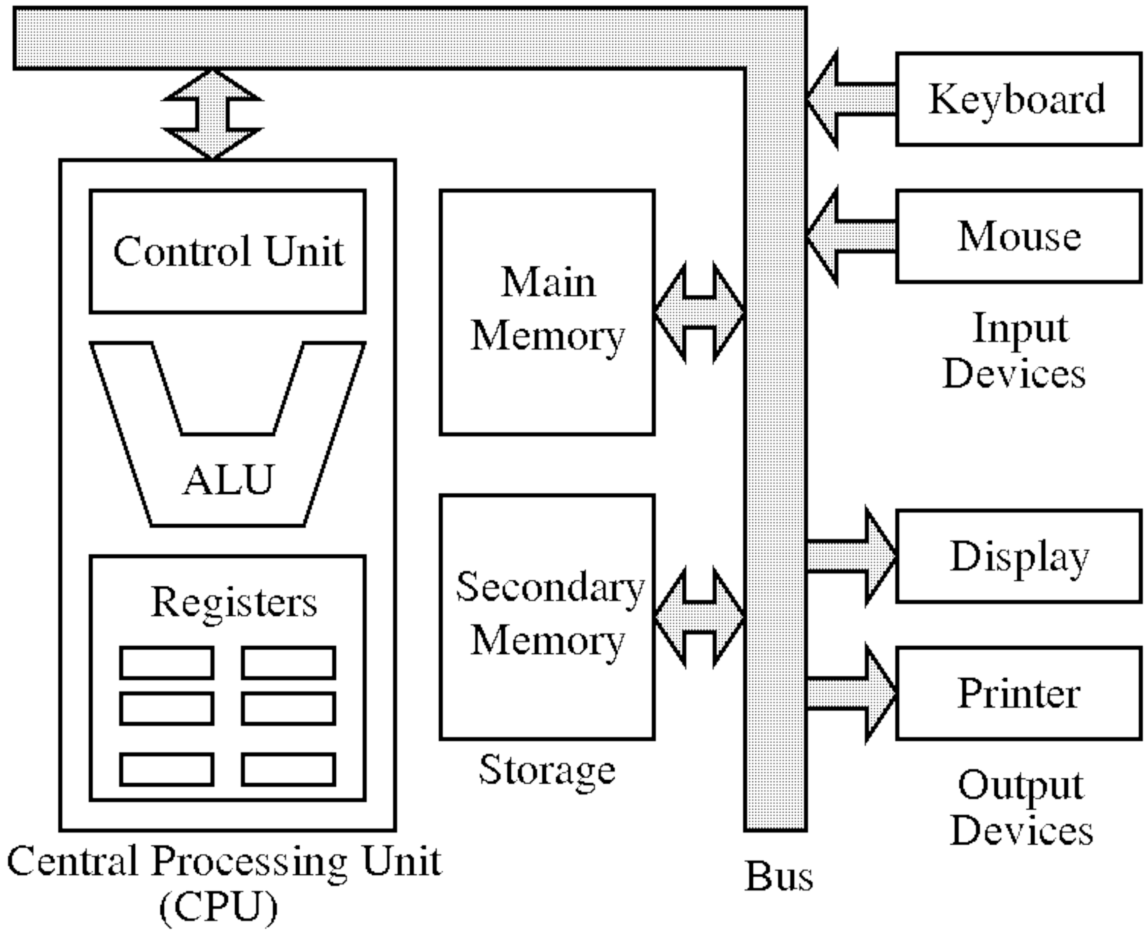
Depending on the OP code Mux chooses
the result of one of the functions (and, or, identity, addition)

Building A Complete Computer from Parts

Computing Machines

- An **instruction** is a single arithmetic or logical operation.
- A **program** is a sequence of instructions that causes the desired function to be calculated.
- A **computing system** is a combination of programs and machine (computer).
- How can we build a computing system that calculates the desired function specified by a program?

Stored Program Computer



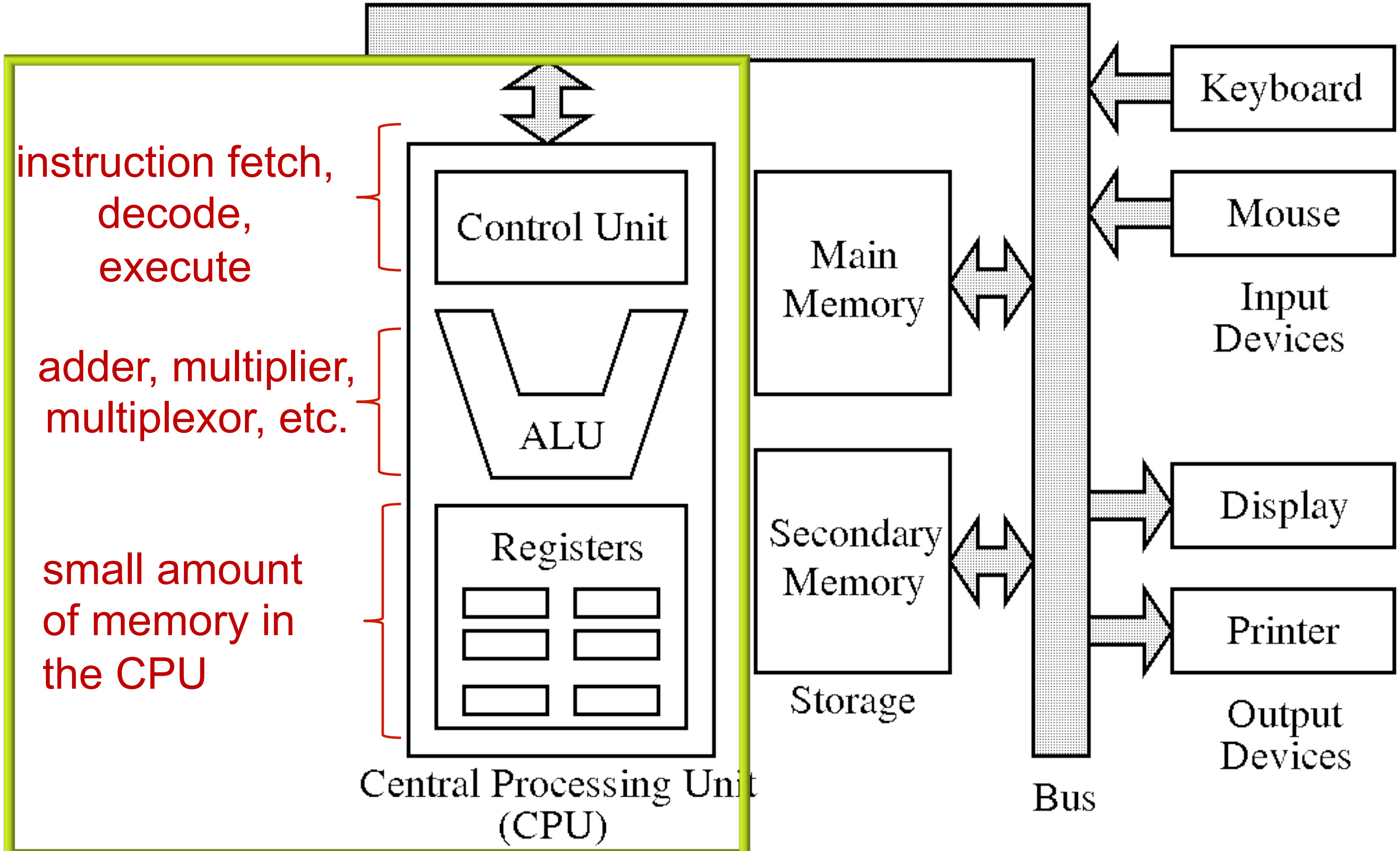
http://cse.iitkgp.ac.in/pds/notes/intro.html

A stored program computer is electronic hardware that implements an instruction set.

Von Neumann Architecture

- Big idea: **Data** and **instructions** to manipulate the data are both bit sequences
- Modern computers built according to the Von Neumann Architecture include separate units
  - To process information (CPU): reads and executes instructions of a program in the order prescribed by the program
  - To store information (memory)

Stored Program Computer



http://cse.iitkgp.ac.in/pds/notes/intro.html

Central Processing Unit (CPU)

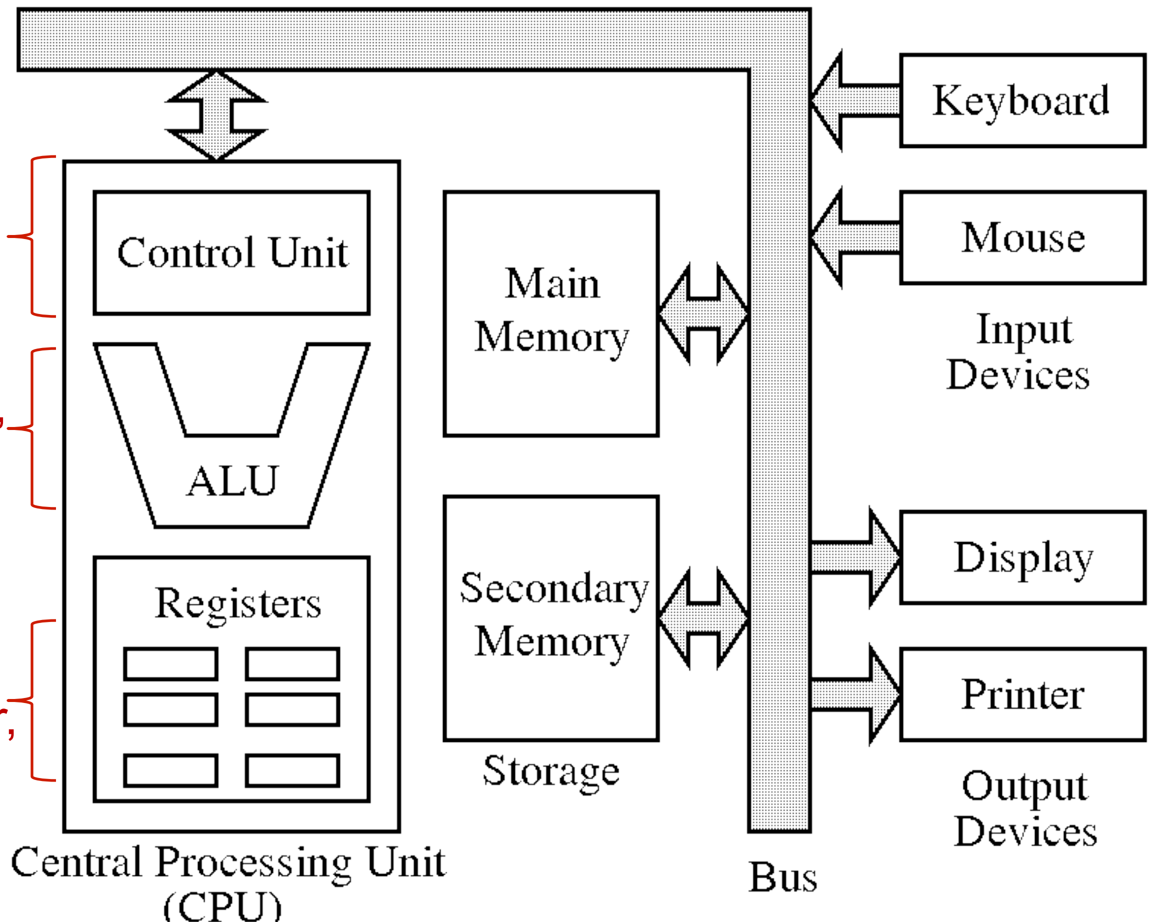
- A CPU contains:
  - Arithmetic Logic Unit to perform computation
    - The brain of the computer; performs all computations
  - Registers to hold information
    - Instruction register (current instruction being executed)
    - Program counter (PC) (to hold location of next instruction in memory)
    - Accumulator (to hold computation result from ALU)
    - Data register(s) (to hold other important data for future use)
  - Control unit to regulate flow of information and operations that are performed at each instruction step

Stored Program Computer

instruction fetch, decode, execute

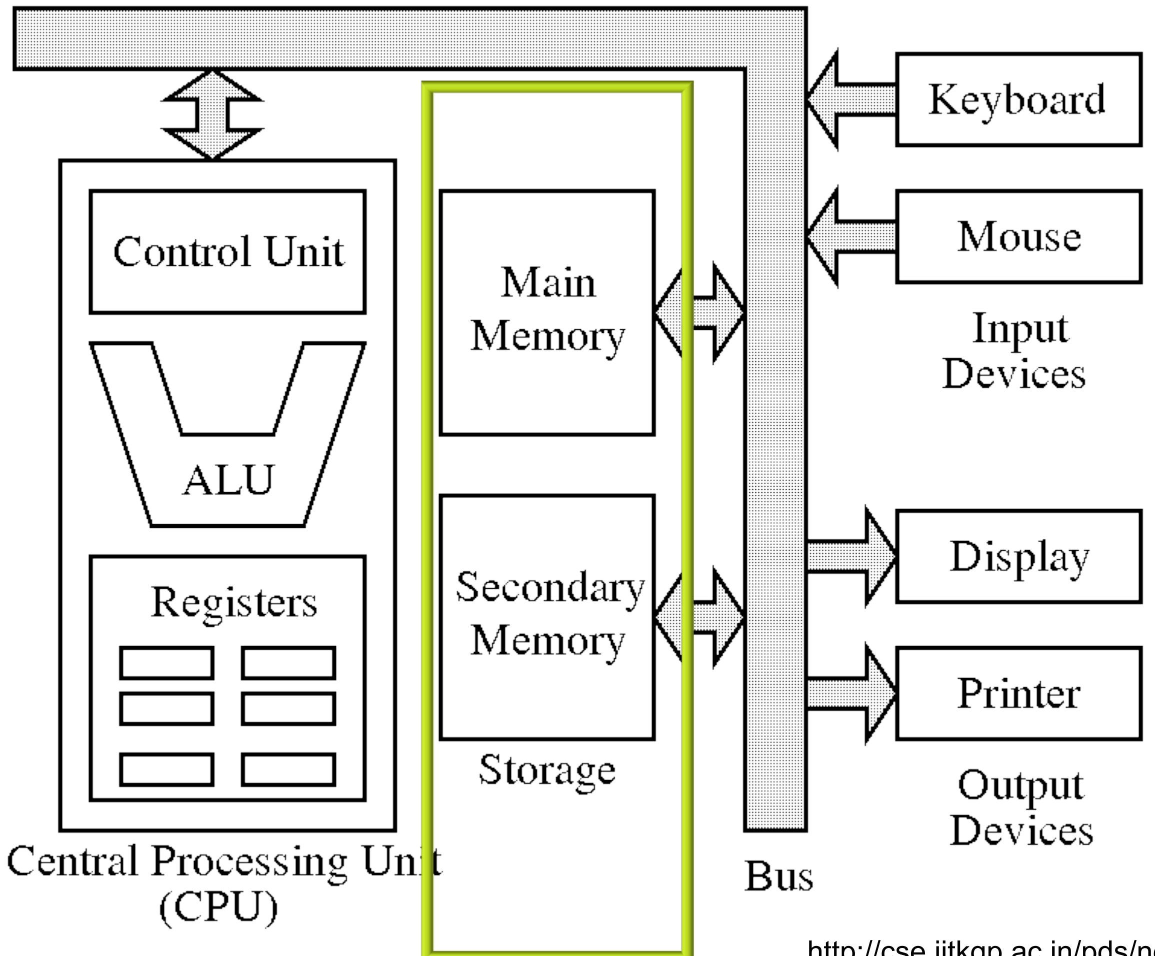
adder, multiplier, multiplexor, Etc.

program counter, instruction register, Etc.



Two specialized registers: the instruction register holds the current instruction to be executed and the program counter contains the address of the next instruction to be executed.

Stored Program Computer



http://cse.iitkgp.ac.in/pds/notes/intro.html

Memory

- The simplest unit of storage is a bit (1 or 0). Bits are grouped into bytes (8 bits).
- Memory is a collection of cells each with a unique physical address.
  - We use the generic term **cell** rather than byte or word because the number of bits in each *addressable location* varies from machine one machine to another.
  - A machine that can generate, for example, 32-bit addresses, can utilize a memory that contains up to 2^{32} memory cells.

Memory Layout

Address	Content
100:	50
104:	42
108:	85
112:	71
116:	99

Address	Content
01100100:	... 01100100
01101000:	... 01010100
01101100:	... 01010101
01110000:	... 01000111
01110100:	... 01100011

We saw this picture in Unit 6. It hid the bit representation for readability. Assumes that memory is byte addressable and each integer occupies 4 bytes.

In this picture and in reality, addresses and memory contents are sequences of bits.

Memory

- Main (or primary) memory:
  - high-speed memory close to the CPU
  - programs are first loaded in the main memory and then executed
  - volatile, i.e., its contents are lost after power-down
- Secondary memory:
  - relatively inexpensive, bigger and low-speed memory
  - for off-line storage, i.e., storage of programs and data for future processing
  - permanent, i.e., its contents last even after shut-down
  - examples of secondary storage include floppy disks, hard disks and CDROM disks

http://cse.iitkgp.ac.in/pds/notes/intro.html

Processing Instructions

- Both data and instructions are stored in memory as bit patterns
  - Instructions stored in contiguous memory locations
  - Data stored in a different part of memory

- **The address of the first instruction is loaded into the program counter and and the processing cycle starts.**

Fetch-Decode-Execute Cycle

- Modern computers include **control logic** that implements the **fetch-decode-execute** cycle introduced by John von Neumann:
  - Fetch next instruction from memory into the instruction register.
  - Decode instruction to a control signal and get any data it needs (possibly from memory).
  - Execute instruction with data in ALU and store results (possibly into memory).
  - Repeat.

Note that all of these steps are implemented with circuits of the kind we have seen in this unit.

Power of abstraction

Using Abstraction in Computer Design

- We can use layers of abstraction to hide details of the computer design.

- We can work in any layer, not needing to know how the lower layers work or how the current layer fits into the larger system.

Low

High

transistors
gates
circuits (adders, multiplexors …)
central processing units (ALU, registers …)
computer

- A component at a higher abstraction layer uses components from a lower abstraction layer without having to know the details of how it is built.
  - It only needs to know what it does.

Abstraction in Programming

- The set of all operations that can be executed by a processor is called its instruction set.
- Instructions are built into hardware: electronics of the CPU recognize binary representations of the specific instructions. That means each CPU has its own machine language that it understands.
- But we can write programs without thinking about on what machine our program will run. This is because we can write programs in high-level languages that are abstractions of machine level instructions.

A High-Level Program

```
# This programs displays “Hello, World!”

print(“Hello world!”)
```

A Low-Level Program

```
title   Hello World Program
; This program displays "Hello, World!"

dosseg
.model small
.stack 100h

.data
hello_message db 'Hello, World!',0dh,0ah,'$'

.code
main  proc
      mov    ax,@data
      mov    ds,ax

      mov    ah,9
      mov    dx,offset hello_message
      int    21h

      mov    ax,4C00h
      int    21h
main  endp
end   main
```

Obtaining Machine Language Instructions

- Programs are typically written in higher-level languages and then translated into machine language (executable code).
- A **compiler** is a program that translates code written in one language into another language.
- An **interpreter** translates the instructions one line at a time into something that can be executed by the computer's hardware.

Summary

- A **computing system** is a combination of program and machine (computer).
- In this lecture, we focused on how a machine can be designed using levels of abstraction:

  gates → circuits for elementary operations → basic processing units → computer