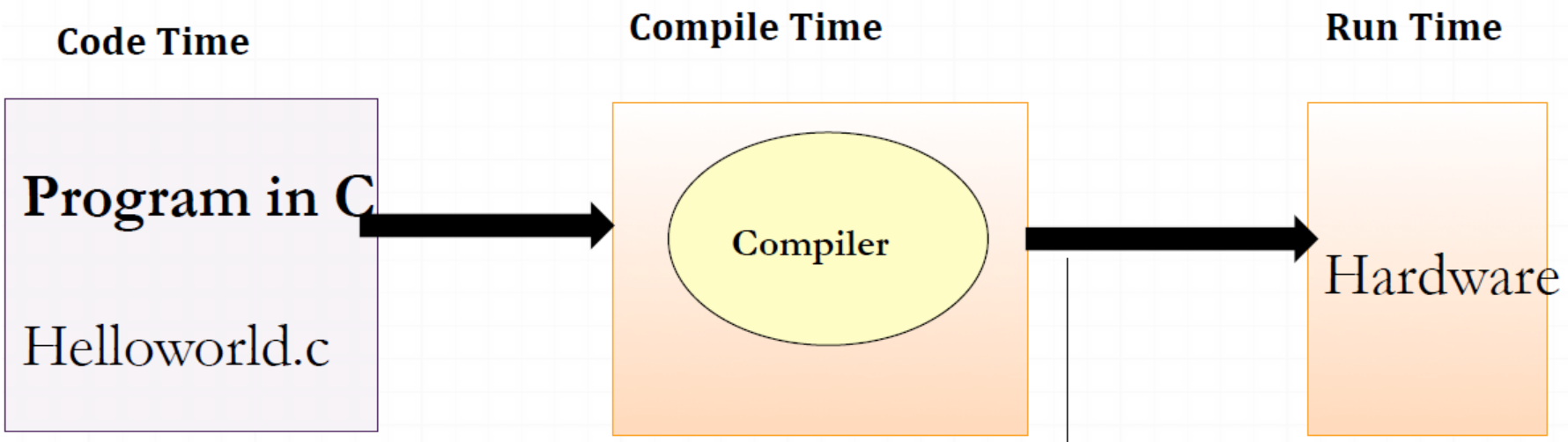
CSIS 212 : Machine Organization & Assembly Language
Lecture 3

Duy Nguyen, Ph.D.

dnguyen@palomar.edu

858.204.5232 (cell)

Steps in Program Translation

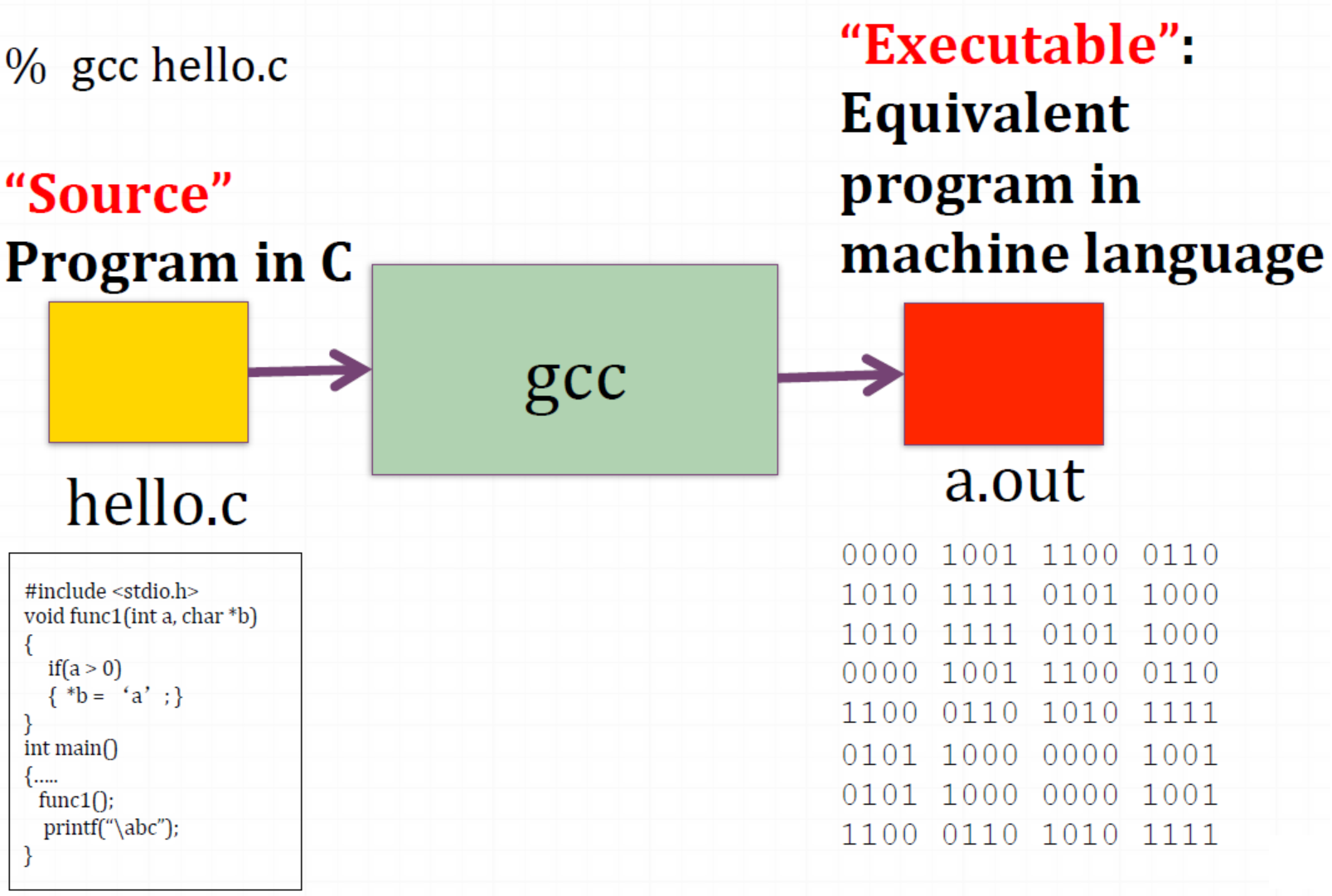


Program:
Text file stored on computers hard disk or some secondary storage

Executable:
Program in machine code
+Data in binary

```
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
```

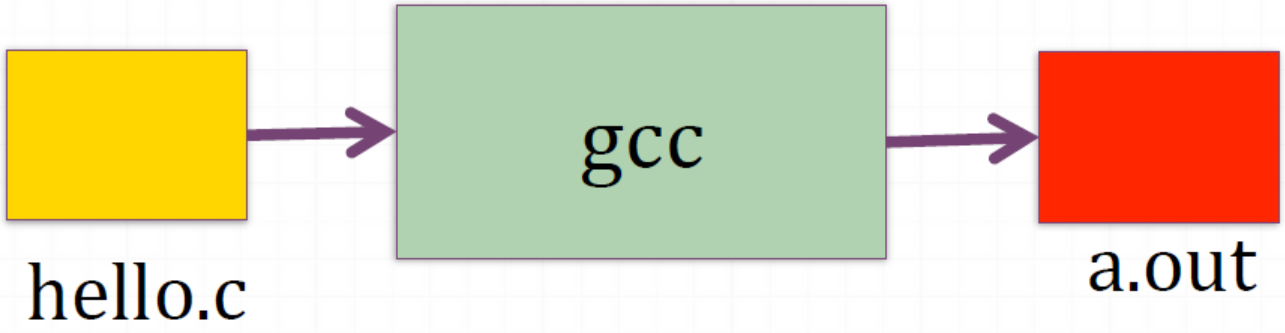
What Does “gcc” Do?



What Does “gcc” Do?

```
%  gcc hello.c
% ./a.out    (executable loaded in memory and processed)
```

Also referred to as “running” the C program



“Source”: Program in C

“Executable”: Equivalent program in machine language

How is “other” code Included?

- Include Header files (.h) that contain function declarations - the function interface
- The corresponding .c files contain the actual code

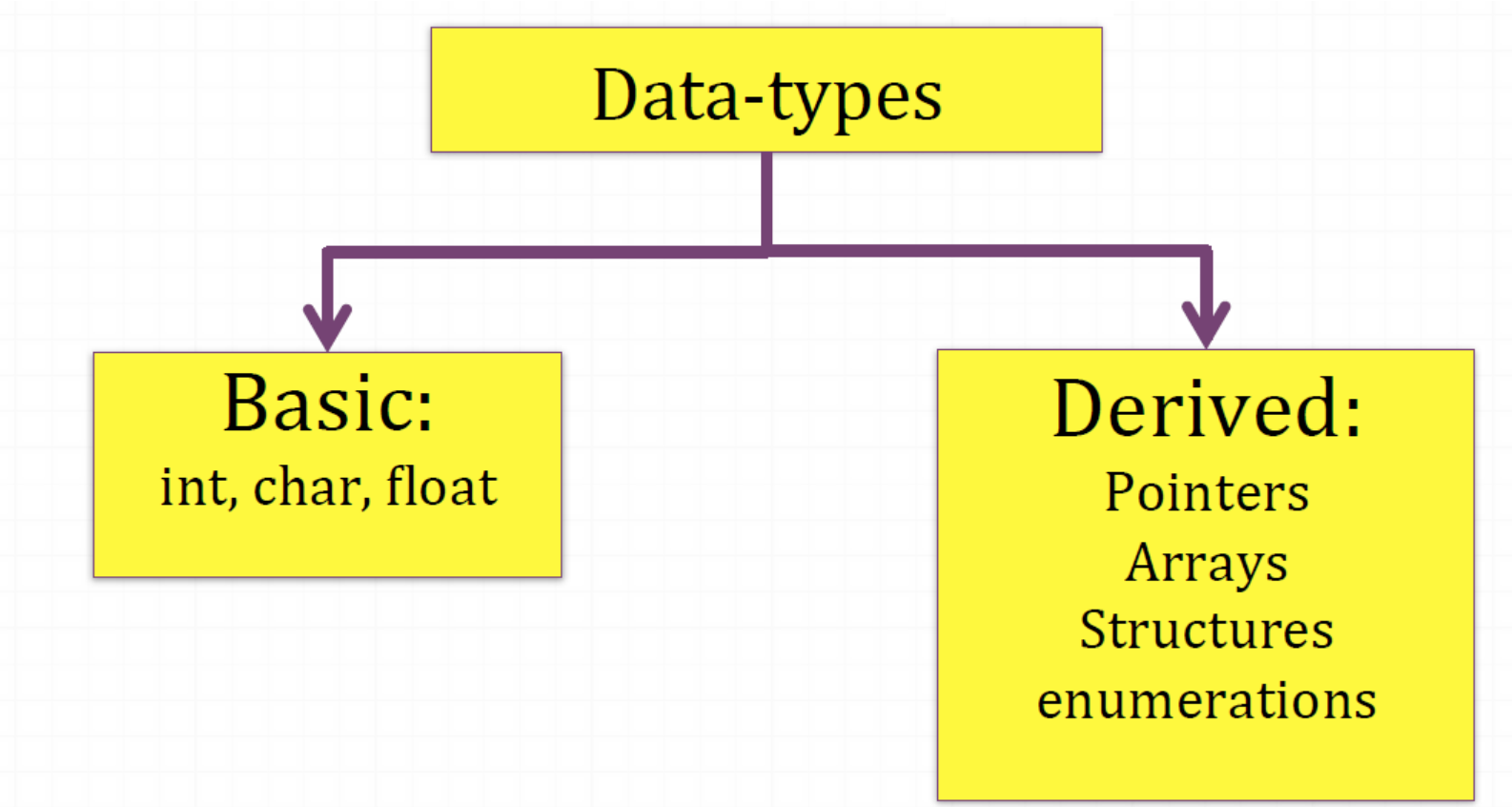
hello.h

```
void func1(int, char *);
int func2(char *, char *);
```

hello.c

```
#include <stdio.h>
void func1(int a, char *b)
{
    if(a > 0)
    { *b = ‘a’ ; }
}
int main()
{.....
  func1();
  printf(“\abc”);
}
```

How We Manipulate Variables Depends on Their Data-Type

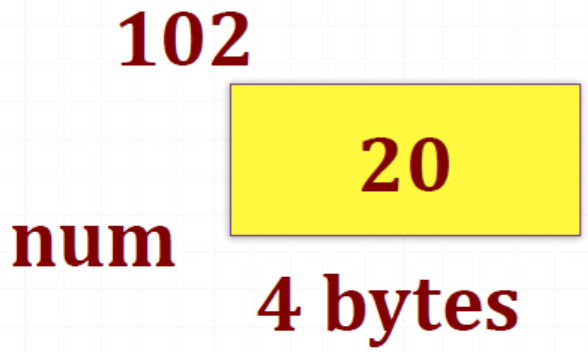


Basic Data Object in Memory

A region in memory that contains a value and is associated with a name/identifier

Attributes of a Data-object/variable:

- Name/identifier
- Value
- Address: Location in memory
- Size
- A unique *data-type*
- Lifetime
- Scope



Declarations and Definitions

```
char c='a';              /* 1 byte */
short s;                 /* 2 bytes */
int a;                   /* usually 4 bytes - signed */
unsigned int a=0;        /* usually 4 bytes*/
float f;                 /* usually 4 bytes use sizeof(float)*/
```

Accessing Value, Lvalue and Rvalue

- To access/change the value of a basic type:

```
y=x;
x=10;
y= x>y ? x : y;
```

102

20

x

Accessing Location

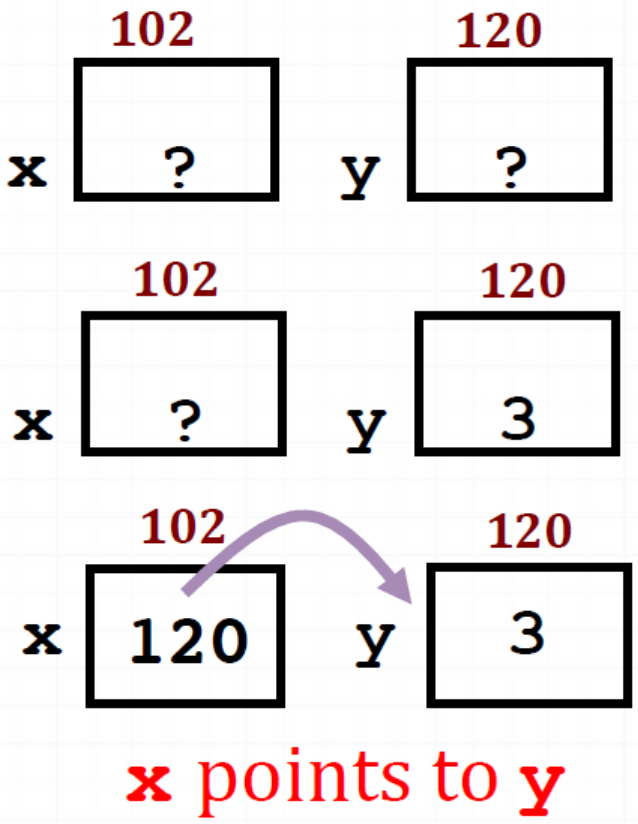
To access the location/address, use the address operator '&'

```
&x   (is 102)
```

Pointers

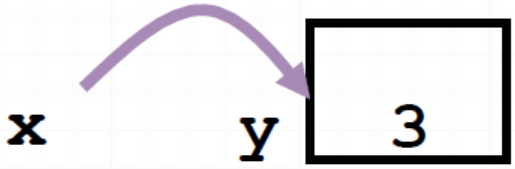
- Pointer: A variable that contains the address of a variable

```
int *x, y;

y = 3;

x = &y;

sizeof(x)=
```



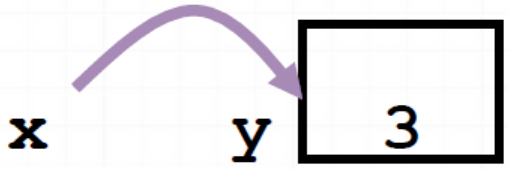
Using Pointers to Change the Value of a Variable

Use dereference * operator to left of pointer name

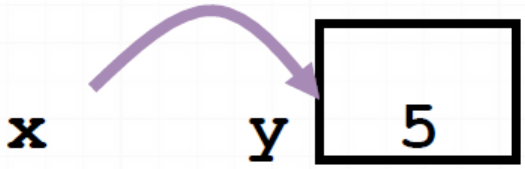



```
*x = 5;
```

- Two ways of changing the value of any variable
- Why this is useful will be clear when we discuss functions and pointers




```
*x = 5;
```



Two Important Facts About Pointers

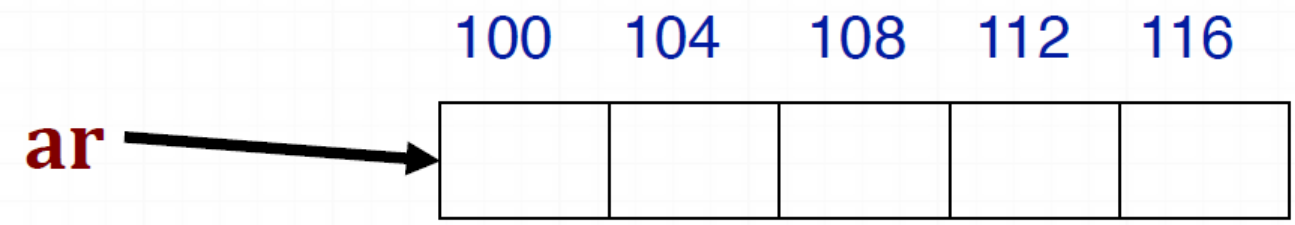
1) A pointer can only point to one type –(basic or derived) such as `int`, `char`, a `struct`, another pointer, etc

2) After declaring a pointer: `int *ptr;`

   `ptr` doesn't actually point to anything yet. We can either:

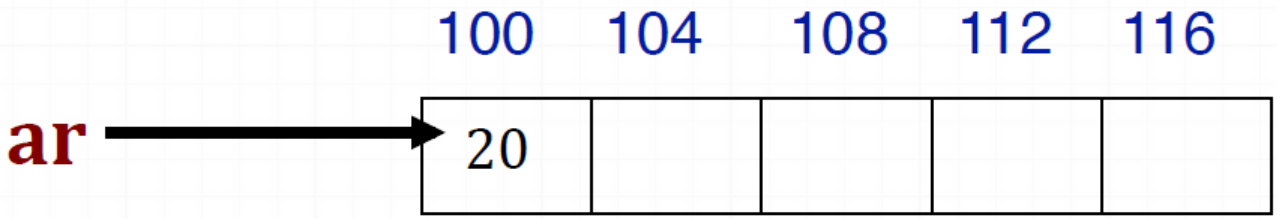
   - make it point to something that already exists, or
   - allocate room in memory for something new that it will point to… (next lecture)

Array Basics




```
int ar[5];    // declares a 5-element integer array
int ar[] = {795, 635}; //declares and fills a 2-element integer array.
```

Array Basics

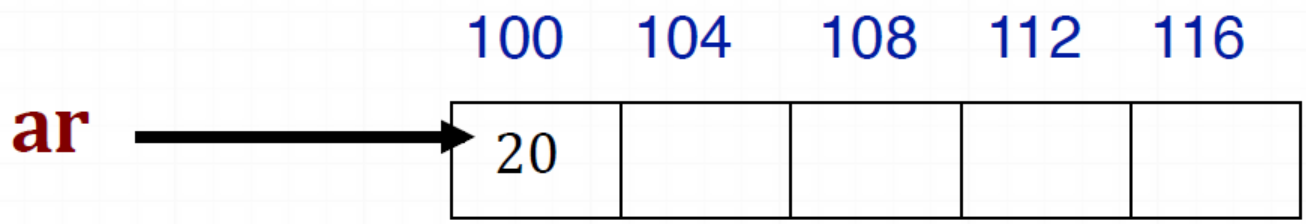


- Accessing elements:

```
ar[i]; // returns the ith element
```

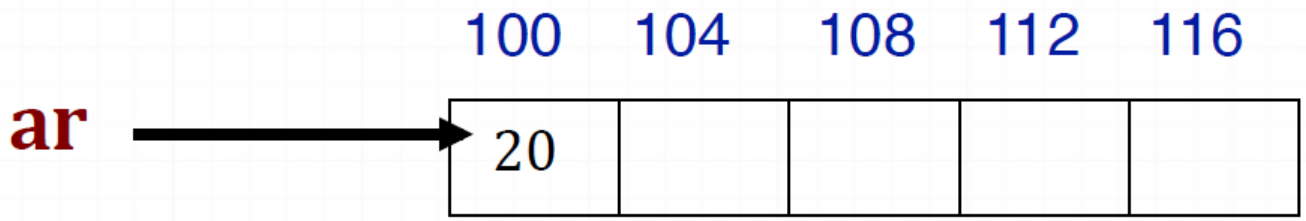
- How are arrays in C different from Java?
- Pointers and arrays are very similar

Array Basics



- `ar` is a pointer to the first element
- `ar[0]` is the same as `*ar`
- `ar[2]` is the same as `*(ar+2)`

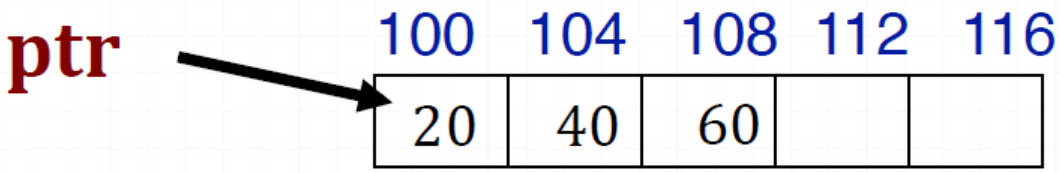
Arrays and Pointers



- Use pointers to pass arrays to functions
- Use *pointer arithmetic* to access arrays more conveniently

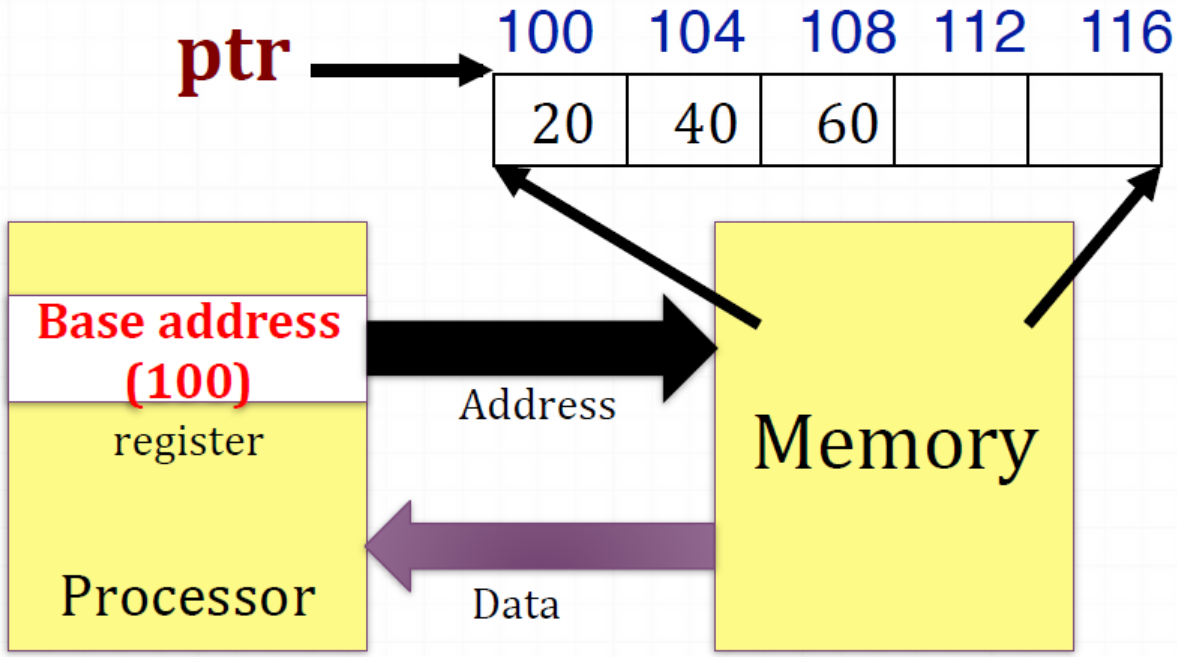
Pointers Arithmetic

- Since a pointer is just a memory address, we can add to it to traverse an array.
- `ptr+1` will return a pointer to the next array element.



Arrays: Fast Data Access

- Using pointer arithmetic, easy to compute the address of any array element in memory
- How are array elements accessed on an ARM?



Arrays

- Pitfall: An array in C does not know its own length, & bounds not checked!
  - Consequence: We can accidentally access off the end of an array.
  - Consequence: We must pass the array and its size to a procedure which is going to traverse it.
- Segmentation faults and bus errors:
  - These are VERY difficult to find, so be careful.

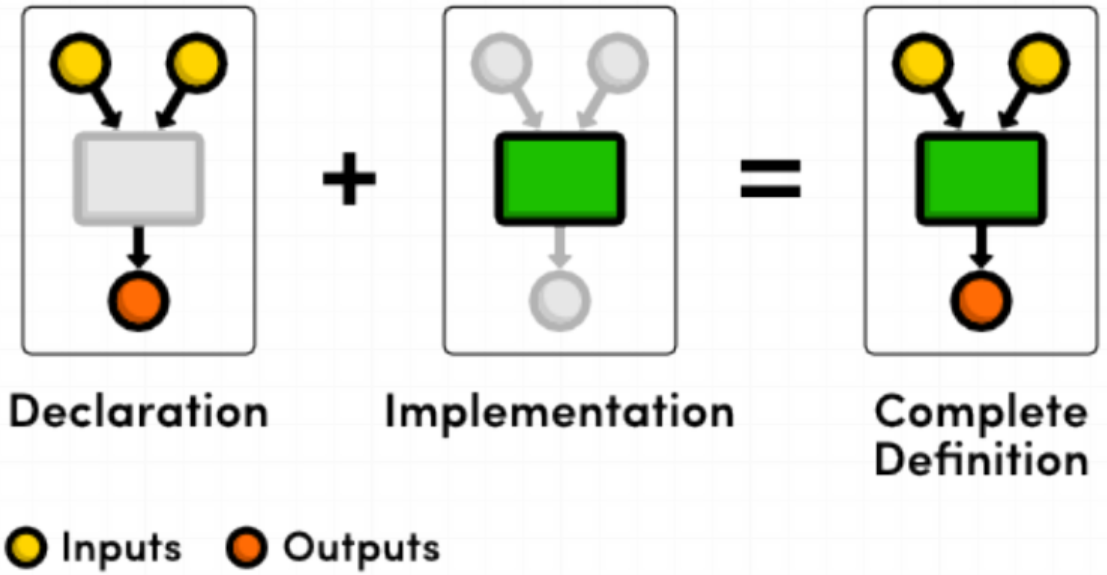
Pointer Arithmetic

- What if we have an array of large structs (objects)?
  - C takes care of it: In reality, `ptr+1` doesn't add 1 to the memory address, but rather adds the size of the array element.
  - C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes: 1 byte for a char, 4 bytes for an int, etc.

- Procedural thought process

```
main () /* High level Outline */
{
   . . .
   get_input(arg1) /*Comment: Step 1 */
   perform_step_2(arg2);
   perform_step_3();
   store_result(); /* Print output or store in a file */

}
```

Overview of Functions



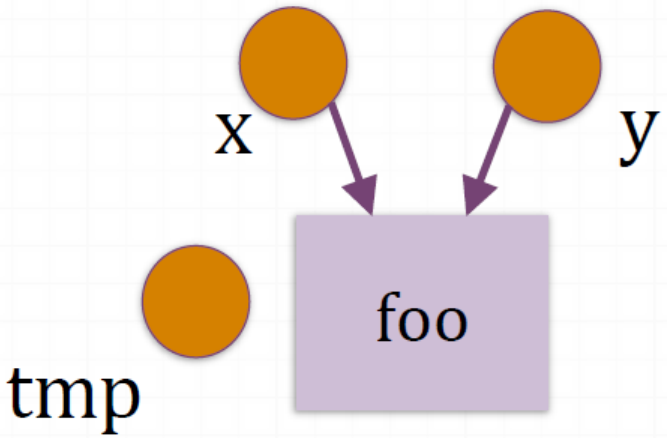
Functions make code easy to

- Maintain
- Debug
- Reuse

Functions Overiew

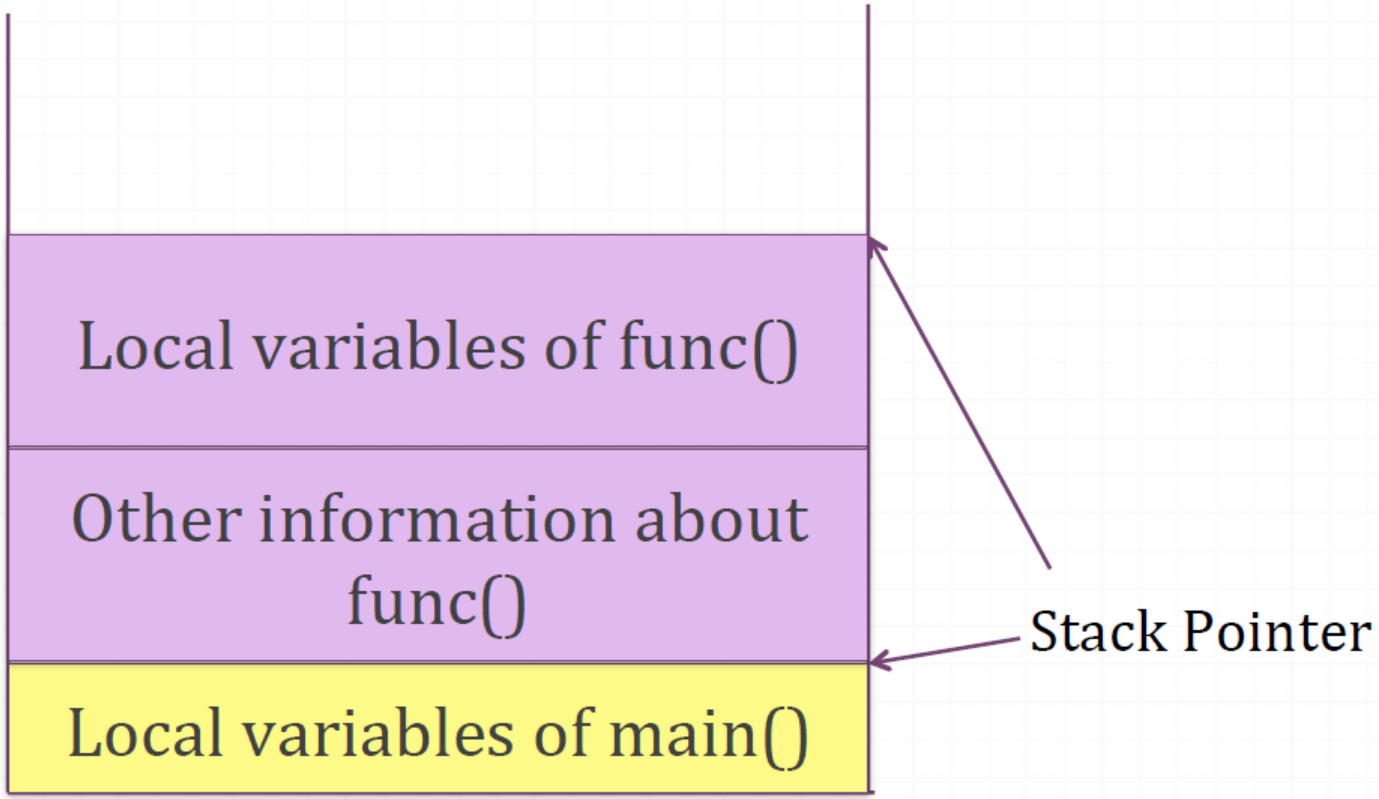
```
//Definition

void foo(int x, int y) {
    int tmp;
    tmp= x;
    x= y;
    y= tmp;  }
```

What does foo do?



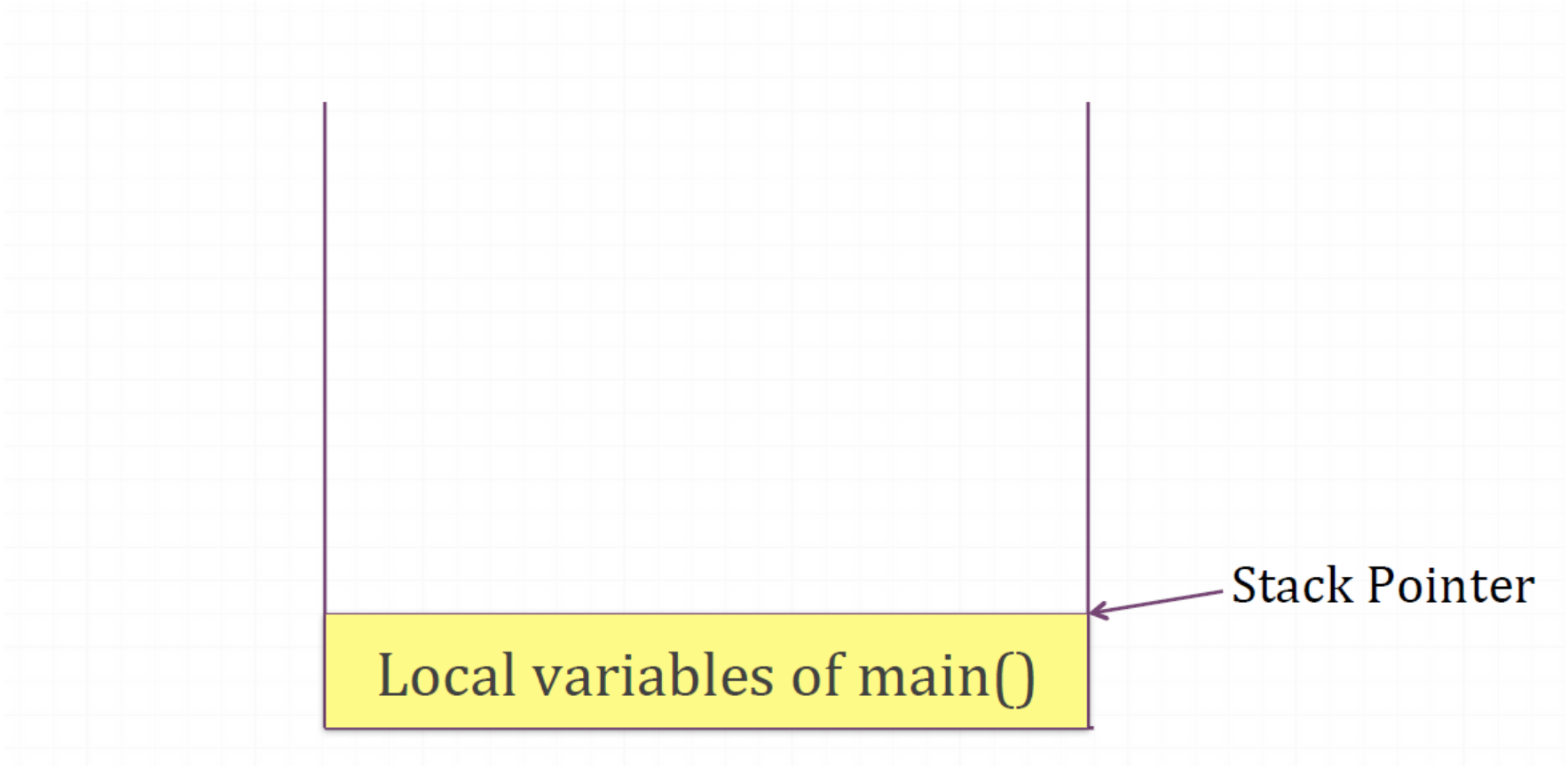
Stack Allocation: Function Local Variables and Parameters

- When program execution starts



What if main calls the function func()?

Stack Allocation: Function Local Variables and Parameters



Possible Lifetimes of Data

1. Execution time of program
2. Time between explicit creation and explicit deletion
3. Execution time of a function (time between function call and function return)

*Variables whose lifetime is the execution time of function are managed using the stack structure

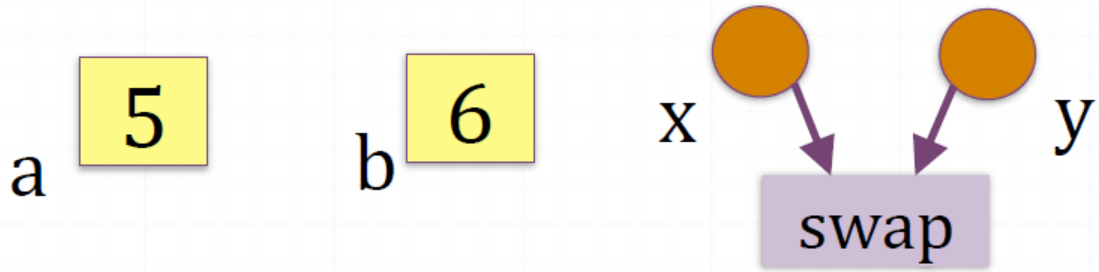
Specifying Scope and Lifetime

Scope and lifetime are often implicit but sometimes we have to use specific keywords:

- `static int a=0;` /*Defines lifetime*/
- `extern int a;` /*Extends scope to multiple files*/

Functions: Call by Value

```
main() {
   . . .
   swap(a, b);
   . . . . }
```



Q: Are the value of variables ‘a’ and ‘b’ interchanged after swap is called?

A. Yes, because that’s what is implemented by the ‘swap’ routine

B. No, because the inputs to swap are only copies of ‘a’ and ‘b’

Functions: Call by Reference

```
void swap(int *x, int *y) {
    . . .
}
```

Q: What should the modified swap function do?

A. Swap the addresses in ‘x’ and ‘y’

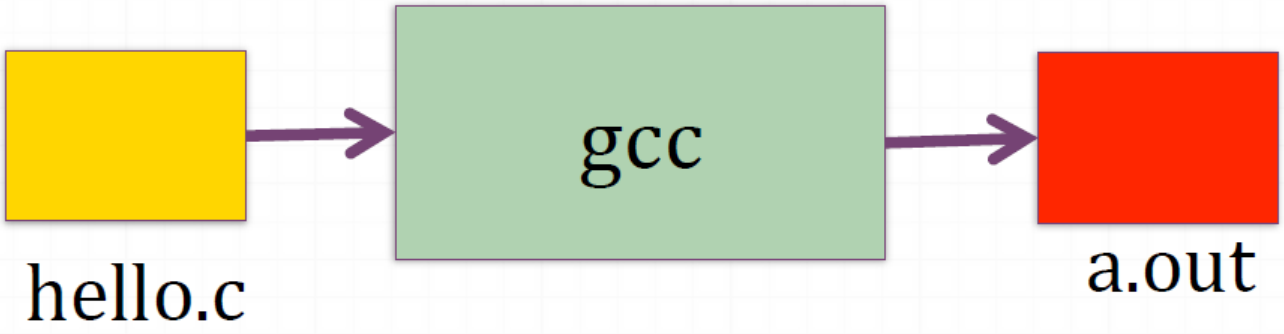
B. Swap the values pointed to by ‘x’ and ‘y’

C. Both the above operations are equivalent

What is Really in an Executable?

```
%  gcc hello.c
% ./a.out    (executable loaded in memory and processed)
```

Also referred to as "running" the C program



"Source": Program in C

"Executable": Equivalent program in machine language

What Does the Executable Contain?

- Instructions or “code”
- Data

What is data?

- Any information that instructions operate on (objects in memory)

C Runtime Environment

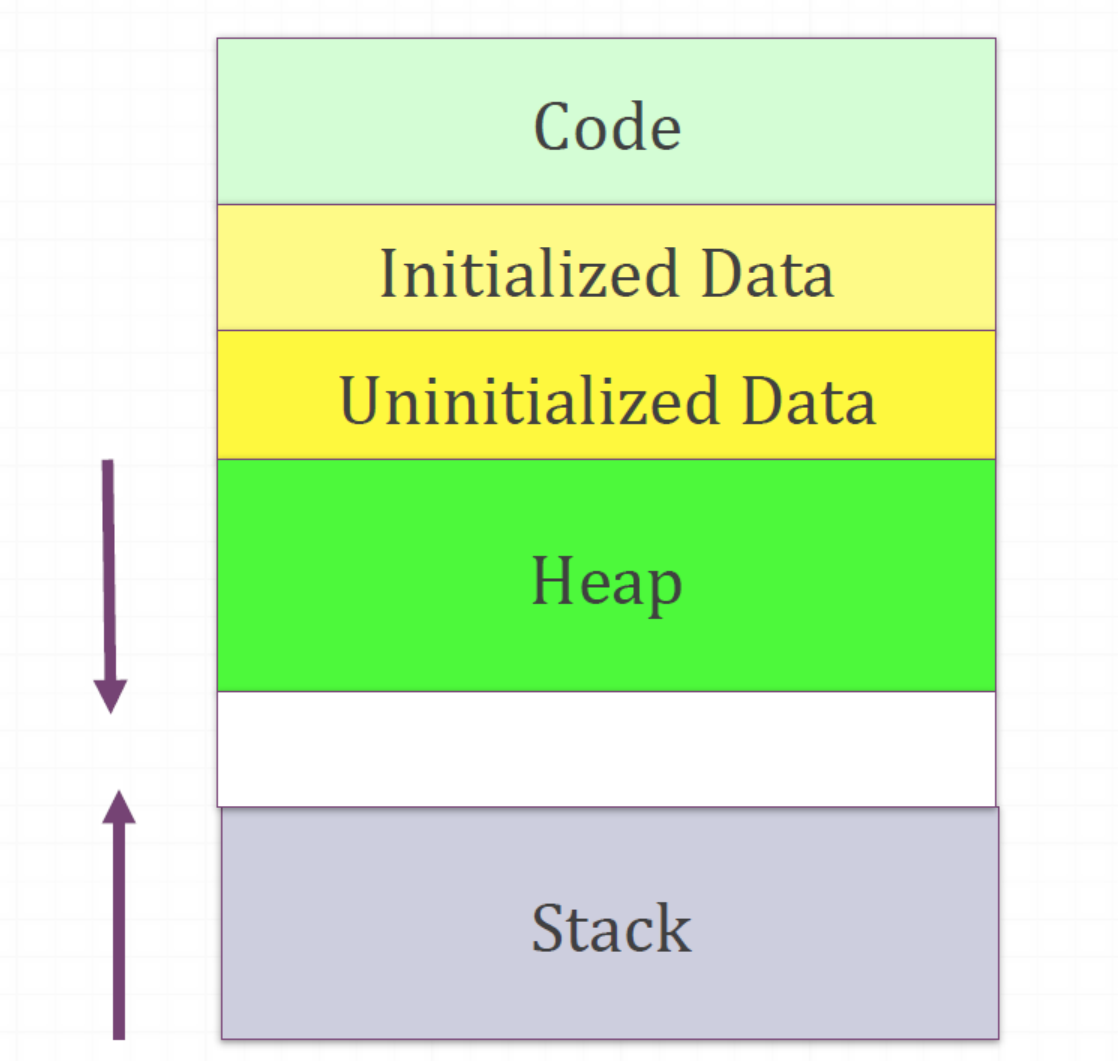
- "Code" (instructions in machine language)
- "Data" (initialized and uninitialized - static allocated)

Both code and data don't change in size

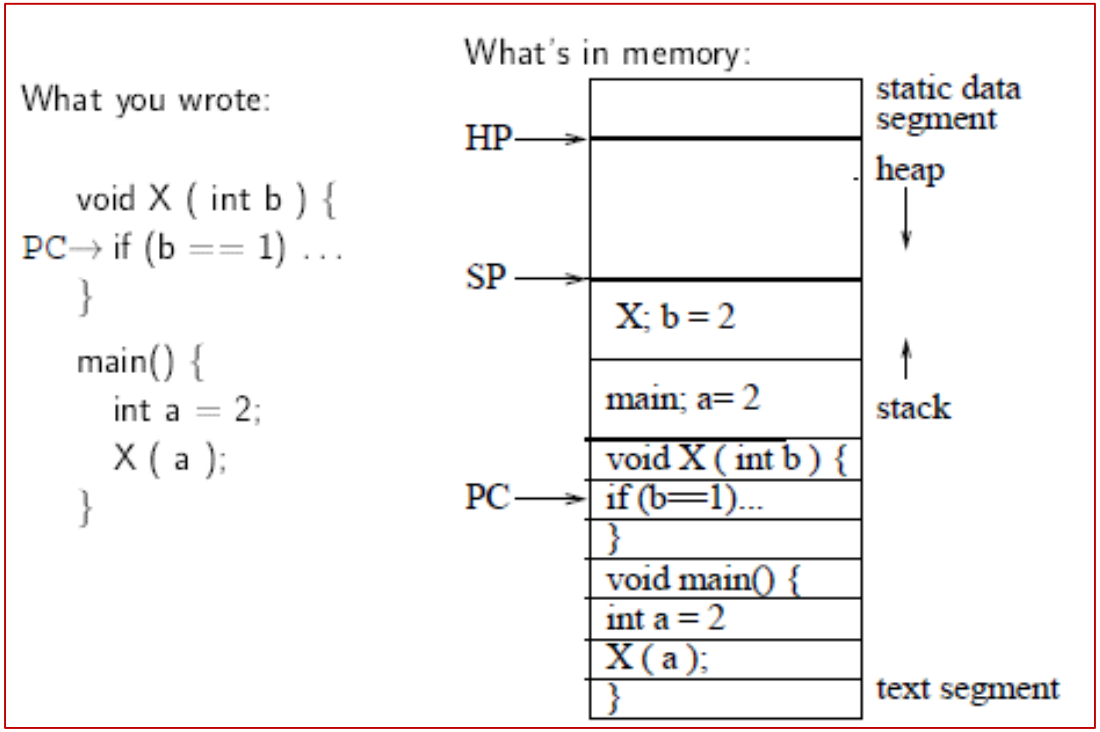
- "Heap" (for dynamically allocated data)
- "Stack" (for function local variables)

Heap and stack change in size as the program executes

C Runtime Environment



Example Process State in Memory



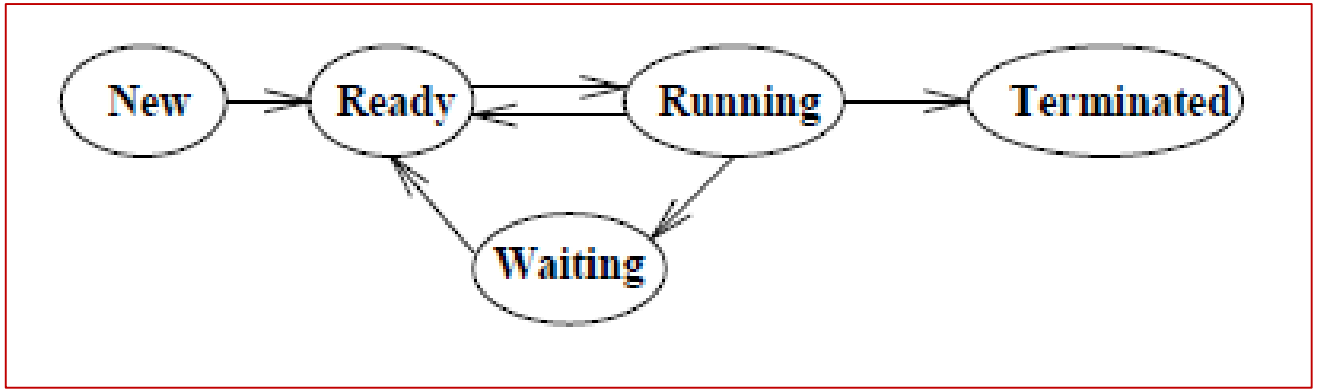
- **Process state consists of at least:**
  - **The code for running the program**
  - **The program counter (PC) indicating the next instruction**
  - **An execution stack with the program's call chain (the stack), the stack pointer (SP)**
  - **The static data for the running program**
  - **Space for dynamic data (the heap), the heap pointer (HP)**
  - **Values of CPU registers**
  - **A set of OS resources in use (e.g., open files)**
  - **Process execution state (e.g., ready, running, etc.)**

Process Execution State

- **Each process has an execution state which indicates what it is currently doing**

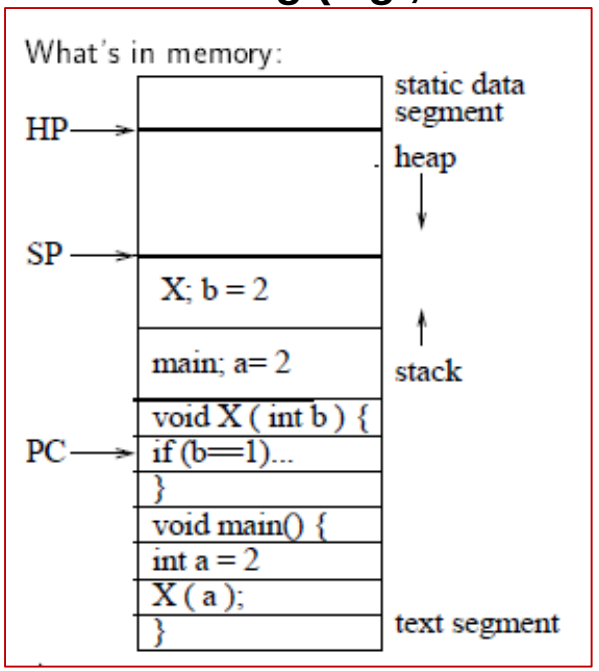
New	the OS is setting up the process state
Ready	ready to run, but waiting for the CPU
Running	executing instructions on the CPU
Waiting	waiting for an event to complete (e.g., I/O)
Terminated	the OS is destroying this process

- As the program executes, it moves from state to state, as a result of the program actions (e.g., system calls), OS actions (scheduling), and external actions (interrupts)



OS Process Data Structures – Process Control Block

- The OS uses a Process Control Block (PCB), its own dynamic data structure, to keep track of all the processes (this data structure is in addition to the process state in memory). The PCB represents the execution state and location of each process when it is not executing (e.g., when it is waiting)
- The PCB contains
  - Process state (running, waiting, etc.)
  - Process number
  - Program Counter (PC)
  - Stack Pointer
  - General Purpose Registers
  - Memory Management Information (HP, etc.)
  - Username of owner
  - List of open files
  - Queue pointers for state queues (e.g., the waiting queue)
  - Scheduling information (e.g., priority)
  - I/O status
  - …



- When a process is created, the OS allocates and initializes a new PCB, and then places the PCB on a state queue
- When a process terminates, the OS deallocates the PCB

C Structures : Overview

- A `struct` is a data structure composed of simpler data types.
  - Like a class in Java/C++ but without methods or inheritance.

```
struct point {
    int x;
    int y;
}
void PrintPoint(struct point p)
{
    printf("(%d,%d)", p.x, p.y);
}
```

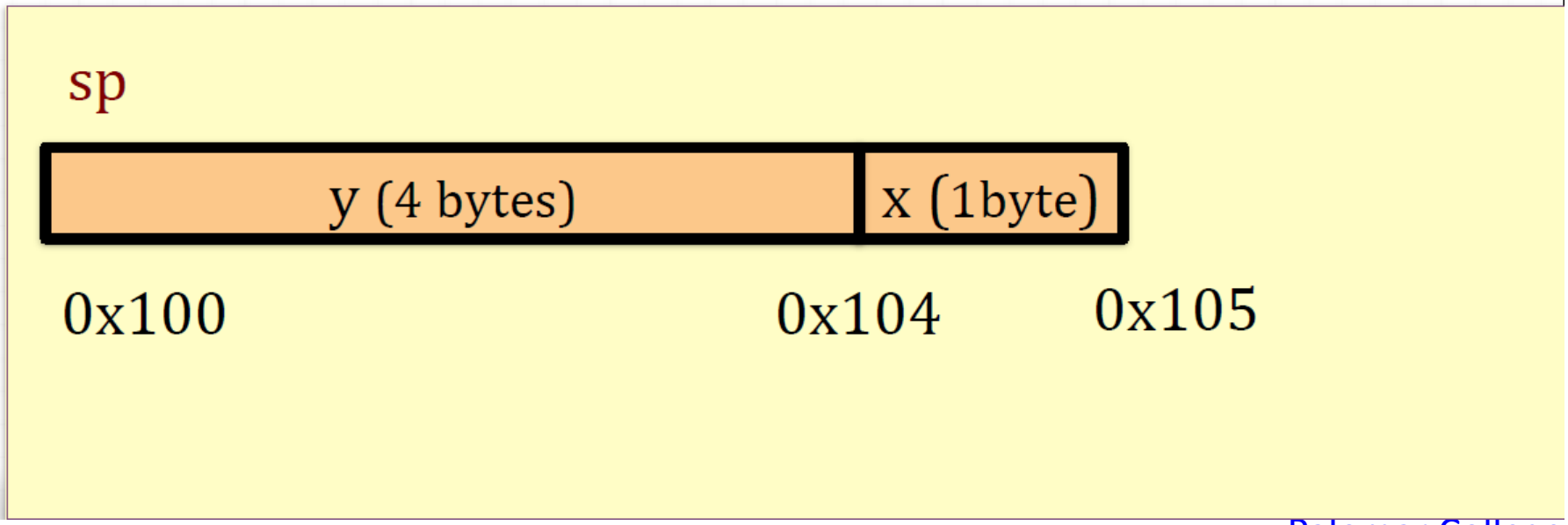
Pointers to Structures

- The C arrow operator (->) dereferences and extracts a structure field with a single operator.
- The following are equivalent:

```
struct point *p;

printf( "x is %d\n" , (*p).x);
printf( "x is %d\n" , p->x);
```

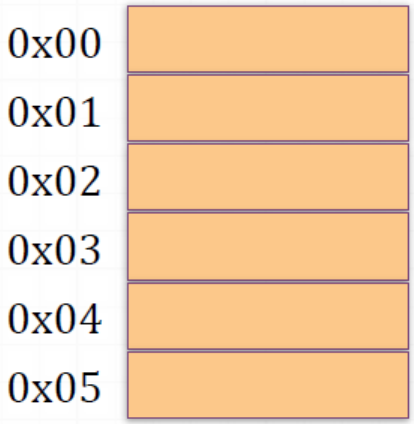
Representation in Memory

```
struct p {
   int y;
   char x;
};
struct p sp;
```

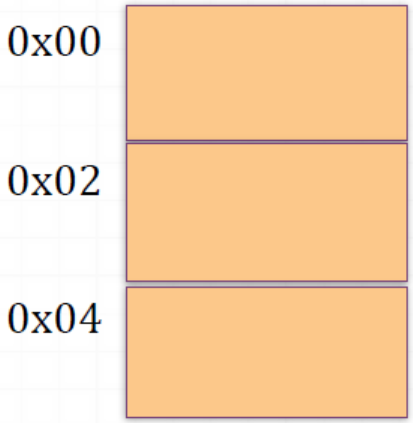


Alignment Fundamentals

- Processors do not always access memory in byte sized chunks, instead in 2, 4, 8, even 16 or 32 byte chunks
- Boundaries at which data objects are stored affects the behavior of read/write operations into memory

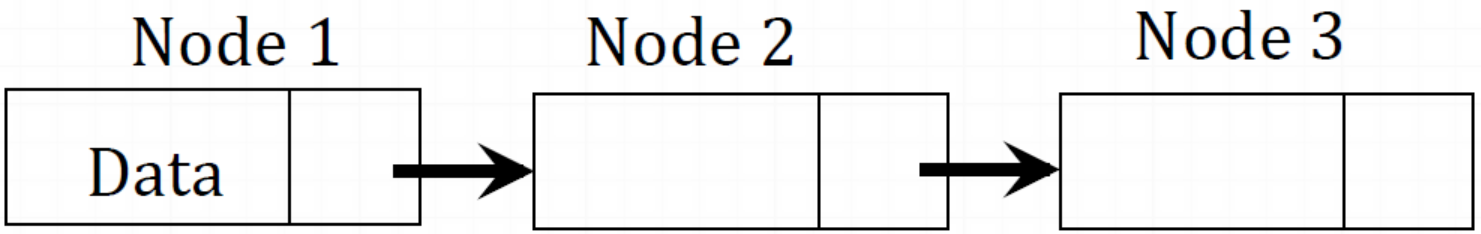


Programmer's view of memory



Processor's view of memory

Linked Lists



A generic linked list has:

- Multiple data objects (structs) also called nodes
- Each node linked to the next node in the list i.e. it knows the address of the next node
- Nodes located at different memory locations (unlike arrays where they are contiguous)

Advantages compared to arrays

- Multiple data members (of different data types) can be stored in each node
- Nodes can be easily inserted or removed from the list without modifying the whole list

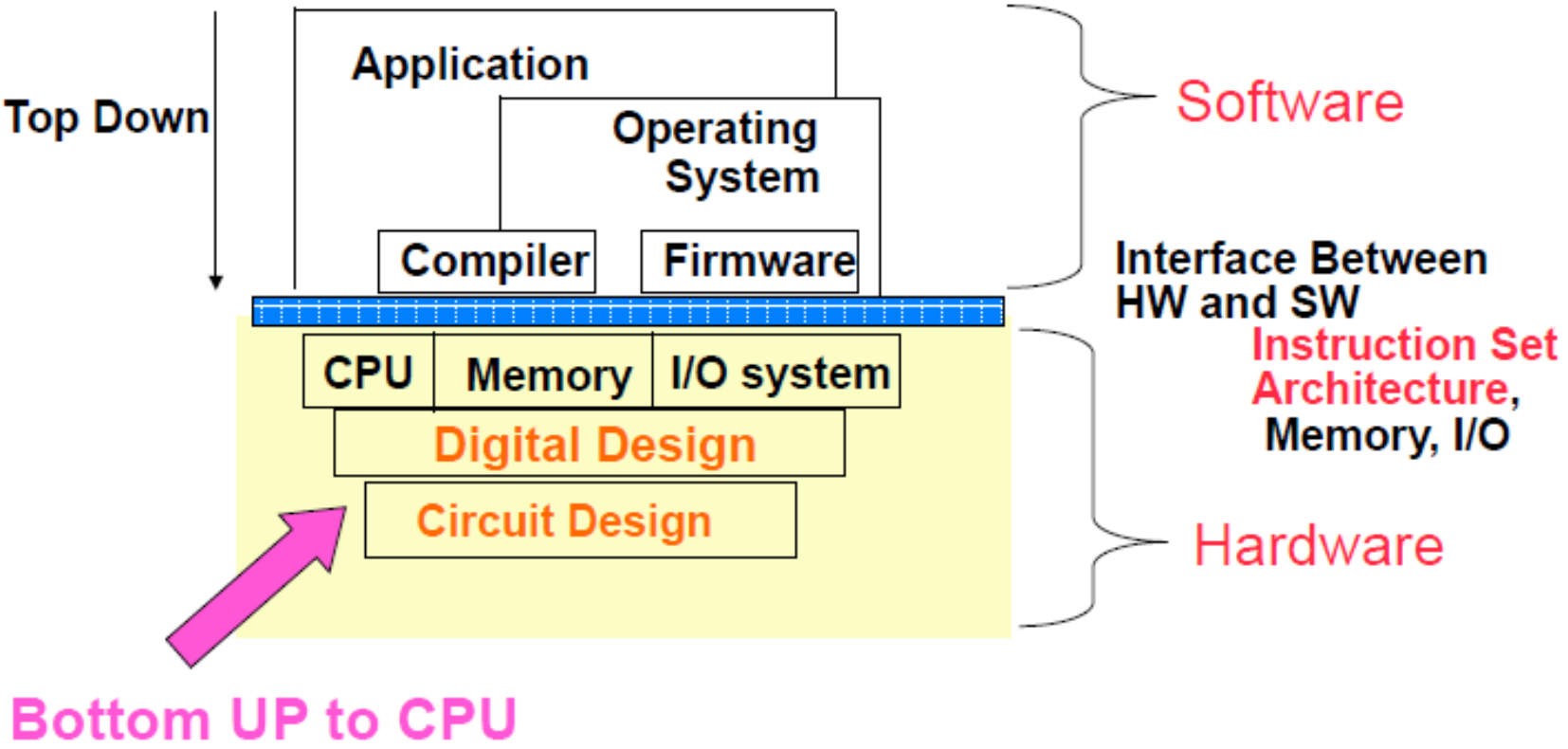
Dynamic Memory Management

- **Java uses "new" keyword to allocate objects dynamically**
  - **Unused objects are automatically deleted**
- **C uses `malloc`/`free` to manage memory dynamically**

```
int main() {
  struct s *my_s = malloc(sizeof(struct s));
  (*my_s).val = 5; // dereference ptr to my_s
  my_struct->weight = 8.6; // shortcut
  …
  free(my_s); // must free mem when finished
}
```

Boolean Algebra

Boolean Algebra Operations



Basic Boolean Operators & Logic Gates

- Inverter (NOT Gate)
- AND Gate
- OR Gate
- Exclusive-OR (XOR) Gate

- NAND Gate = AND Gate + Inverter
- NOR Gate = OR Gate + Inverter
- Exclusive-NOR Gate = XOR Gate + Inverter

The basic logic gates are the inverter (or NOT gate), the AND gate, the OR gate and the exclusive-OR gate (XOR). If you put an inverter in front of the AND gate, you get the NAND gate etc.

Truth Table

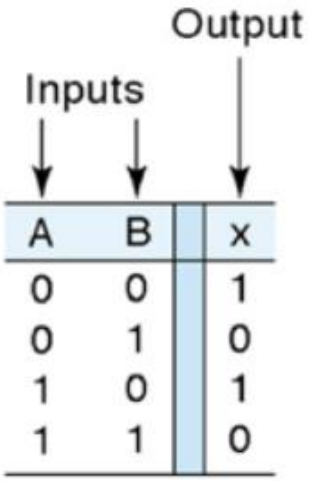
One of the common tool in specifying a gate function is the truth table. All possible combination of the inputs A, B … etc, are enumerated, one row for each possible combination. Then a column is used to show the corresponding output value.

If two ligic circuits share identical truth table, they are functionally equivalent.

Shown here are example of truth tables for logic gate with 2, 3 and 4 inputs.

Truth Tables

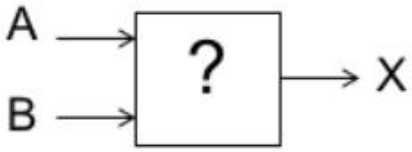
- Truth Tables specifies how a logic circuit's output depends on the logic levels present at the inputs.

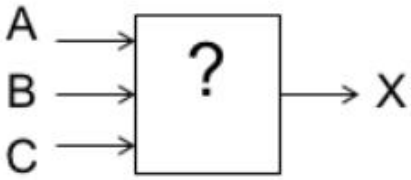


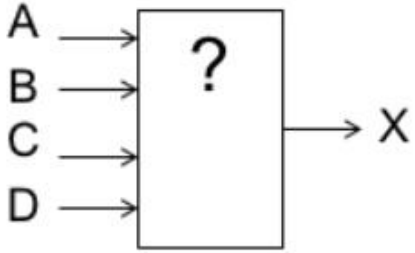
A	B	x
0	0	1
0	1	0
1	0	1
1	1	0

A	B	C	x
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

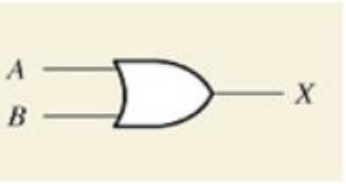
A	B	C	D	x
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1







Different Representations – OR Gate



Distinctive shape symbol

Rectangular outline symbol

Symbol or Schematic

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

Truth table

0 = LOW
1 = HIGH

Truth Table & Boolean Expression

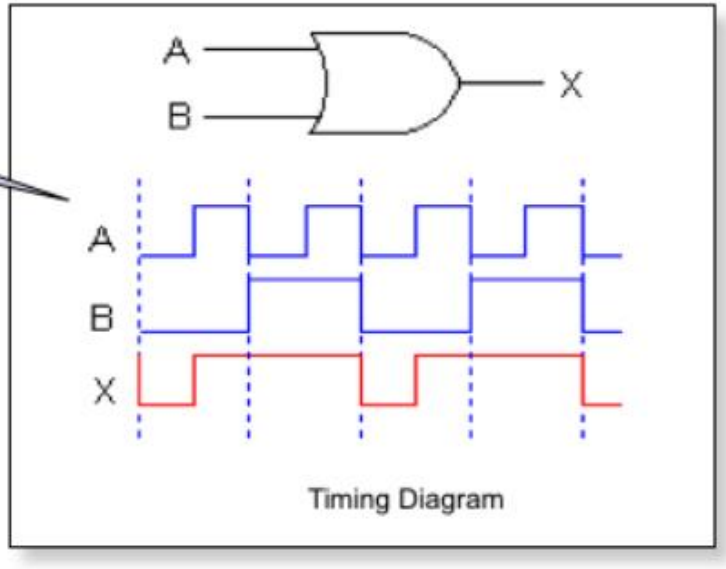
X = A + B

Boolean expression

Boolean: A or B
Multiple bits: A | B

Python

Timing Diagram



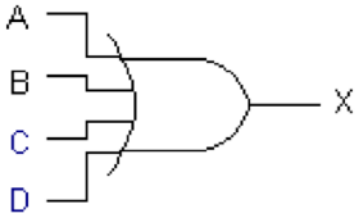
Timing Diagram

The output of an OR gate is HIGH whenever one or more inputs are HIGH

Five Different Representations of the OR Gate

Here we show five different representation of the OR gate or OR function. They are:

1. Sch
2. Tru
3. Boc
4. Tim
5. Exp

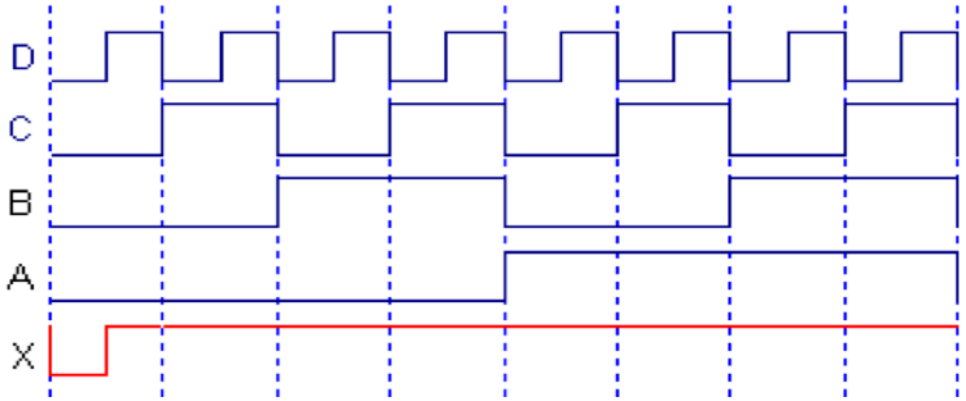


gic symbol

g language (e.g. Python

$X = A+B+C+D$

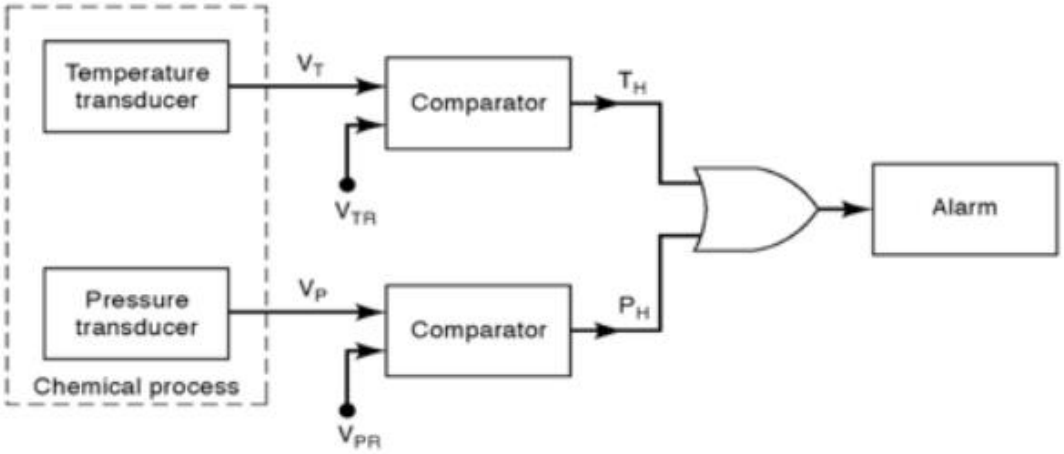
A	B	C	D	X
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

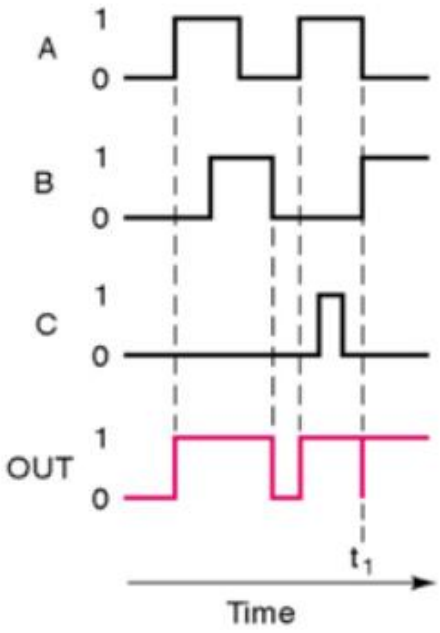


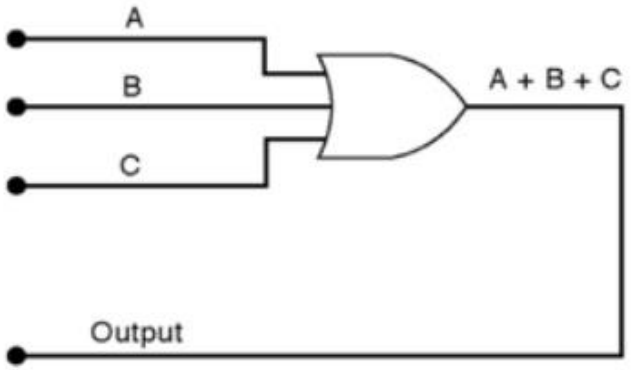
In summary, OR operation produces as result of 1 whenever any input is 1. Otherwise 0.
An OR gate is a logic circuit that performs an OR operation on the circuit's input.
The expression x=A+B is read as "**x equals A OR B**"

Two More Examples of OR Function

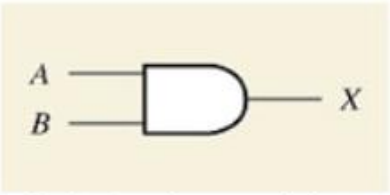
- The merging nature of OR gate.



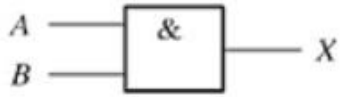




The AND Operation & AND Gate



Distinctive shape symbol



Rectangular outline symbol

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

Truth table

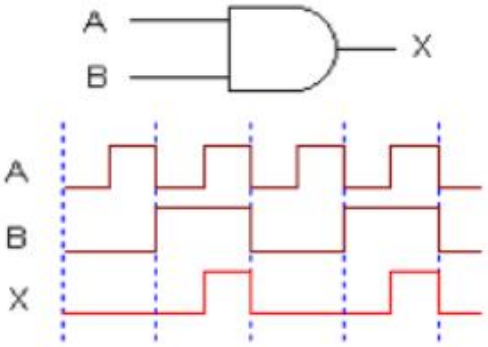
0 = LOW
1 = HIGH

X = AB

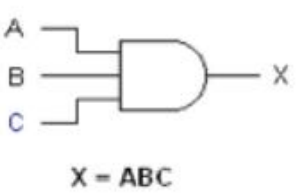
Boolean expression

Boolean: A and B
Multiple bits: A & B

Python



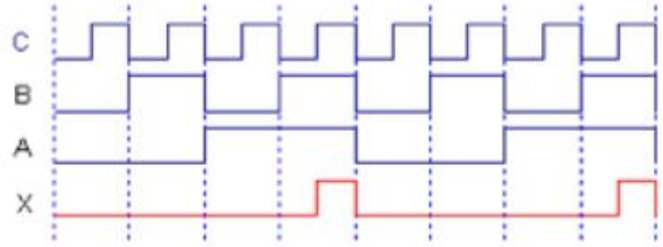
Timing Diagram



X = ABC

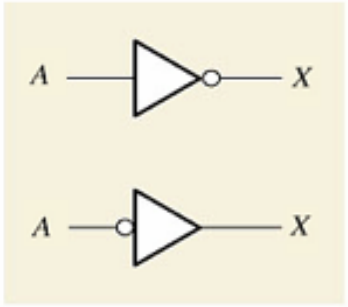
A	B	C	X
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

3-Input AND Gate

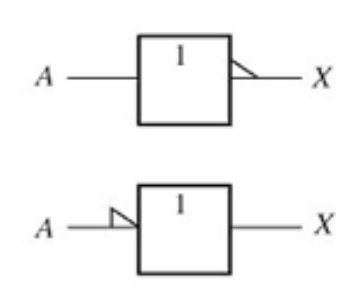


The output of an AND gate is HIGH only when all inputs are HIGH.

The NOT Operation & Inverter



Distinctive shape symbols



Rectangular outline symbols

A	X
0	1
1	0

Truth table

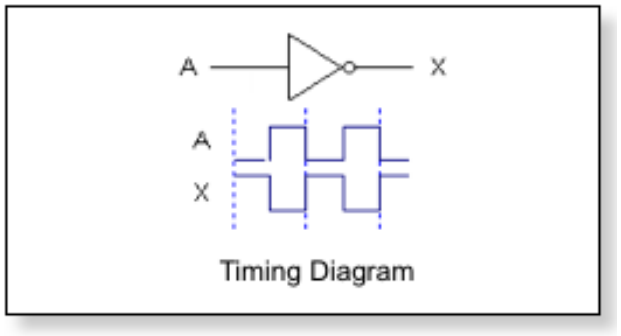
$X = \overline{A}$

Boolean expression

0 = LOW
1 = HIGH

Boolean: not A
Multiple bits: ~A

Python



Timing Diagram

The output of an inverter is always the complement (opposite) of the input.

Describing Logic Circuits Algebraically

Using symbolic diagram or truth table to specify or describe logic gates and logic functions is cumbersome. A much better way is to use algebraic expression. Here a "dot" represents the AND operation, and a "+" represents and OR operation. Furhtermore, a bar over a variable or a '/' in front of the variable represents an inversion (NOT function).

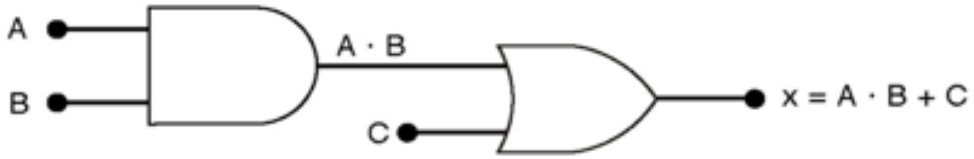
The convention is that AND has precedence over OR.

Precedence rules in Boolean algebra:

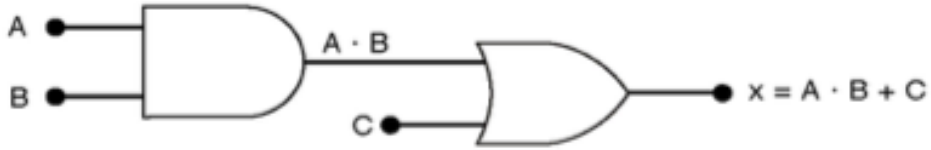
1. First, perform all inversions of single terms
2. Perform all operations with parentheses
3. Perform an AND operation before an OR operation unless parentheses indicate otherwise
4. If an expression has a bar over it, perform the operations inside the expression first and then invert the result

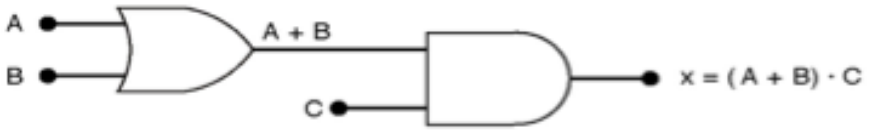
Describing Logic Circuits Algebraically

- Any logic circuit, no matter how complex, can be completely described using the three basic Boolean operations: OR, AND, NOT.
- Example: logic circuit with its Boolean expression



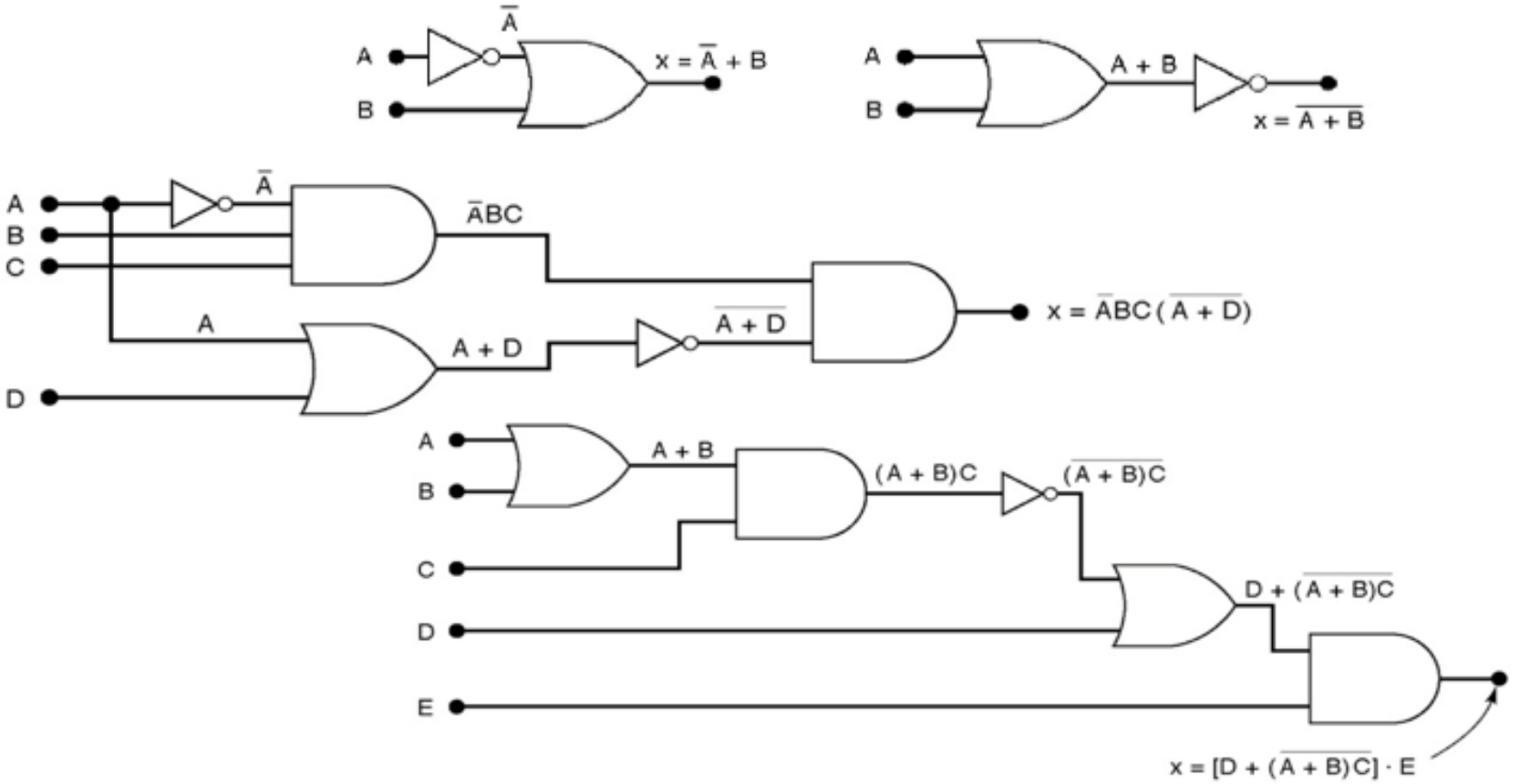
- How to interpret A•B+C?
  - Is it A•B ORed with C ? Is it A ANDed with B+C ?
- Order of precedence for Boolean algebra: AND before OR. Parentheses make the expression clearer, but they are not needed for the case on the preceding slide.
- Therefore the two cases of interpretations are :



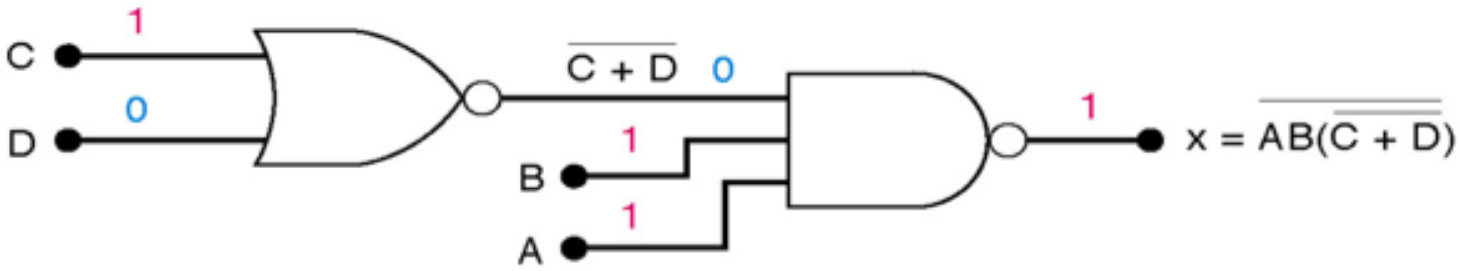


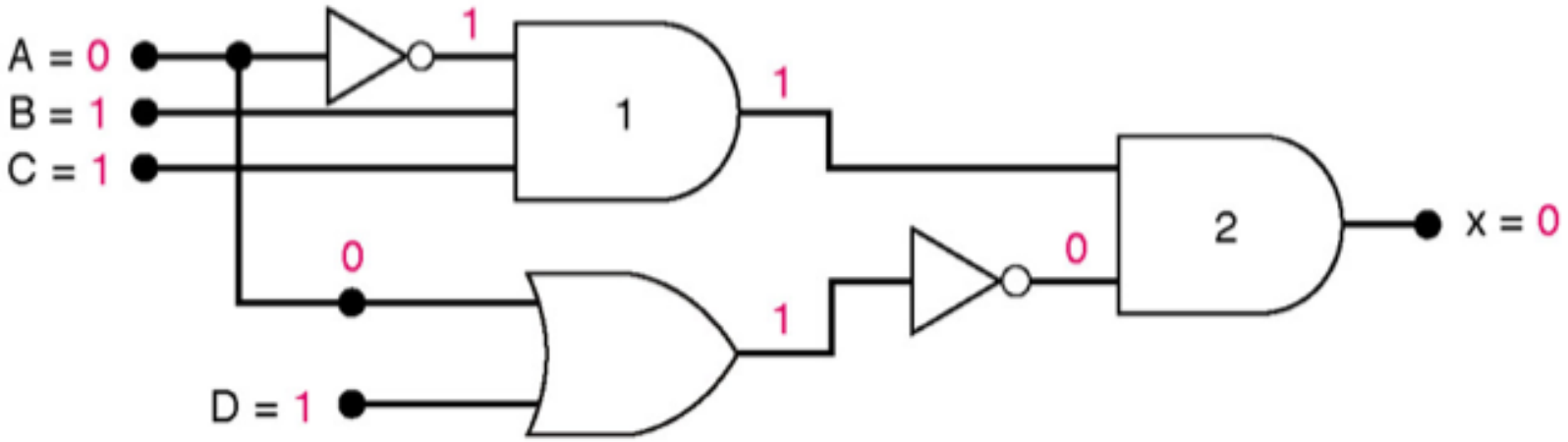
Circuits Contain INVERTERS

- Whenever an INVERTER is present in a logic-circuit diagram, its output expression is simply equal to the input expression with a bar over it.



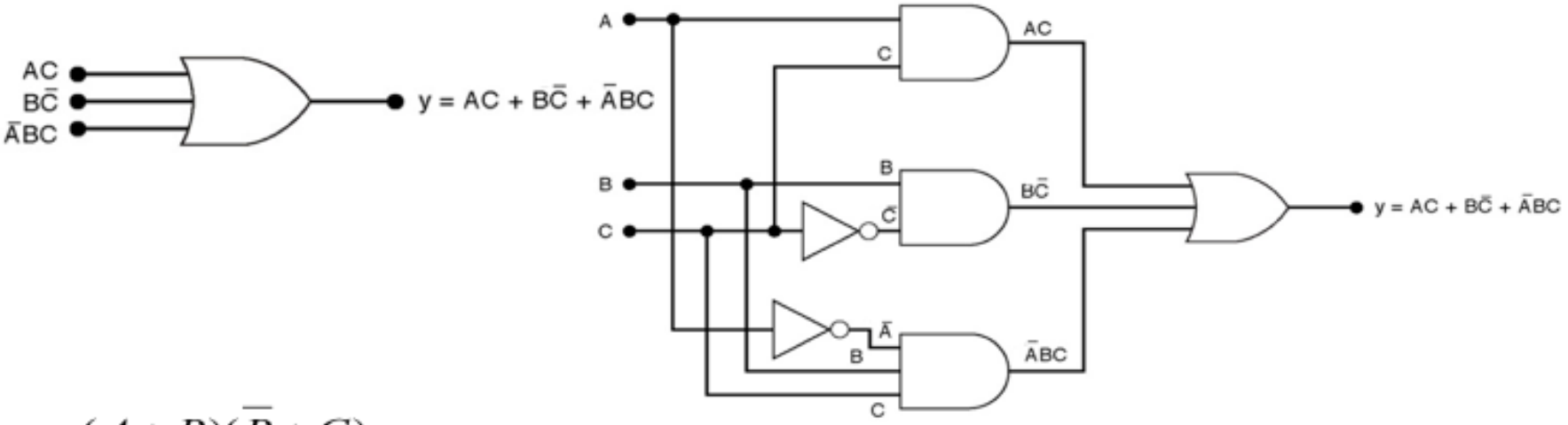
Determine Output Value from a Diagram



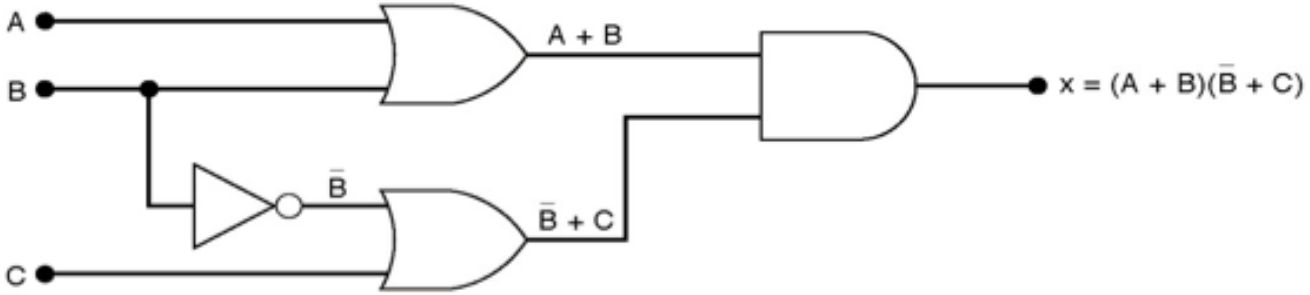


Implementing Circuits From Boolean Expressions

- When the operation of a circuit is defined by a Boolean expression, we can *draw a logic-circuit* diagram directly from that expression.



$$x = (A + B)(\overline{B} + C)$$

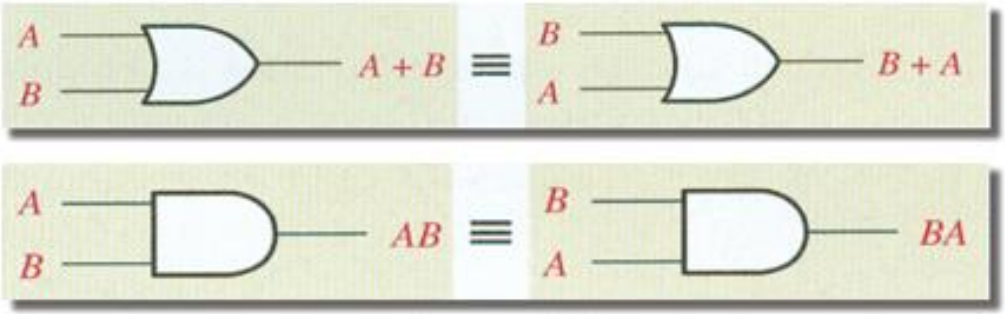


Laws of Boolean Algebra

- Commutative Laws

$A + B = B + A$

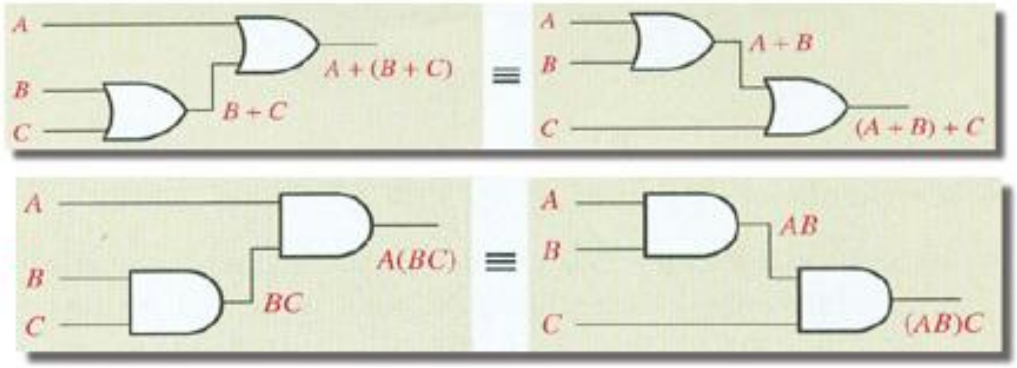
$A \cdot B = B \cdot A$



- Associative Laws

$A + (B + C) = (A + B) + C$

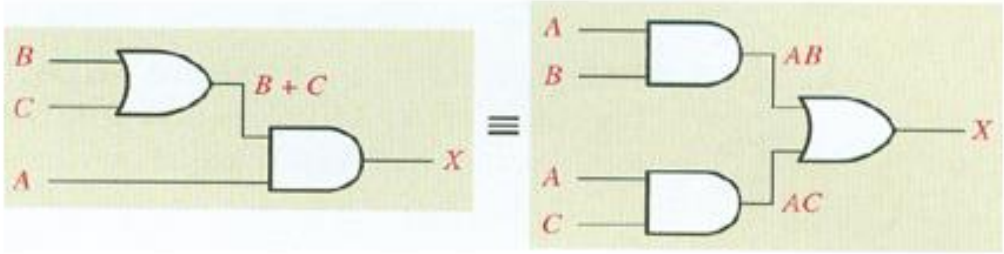
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$



- Distributive Law

$A \cdot (B + C) = A \cdot B + A \cdot C$

$A\,(B + C) = A\,B + A\,C$

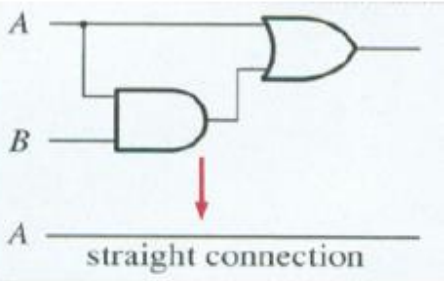


Rules of Boolean Algebra

1. $A + 0 = A$	7. $A \cdot A = A$
2. $A + 1 = 1$	8. $A \cdot \overline{A} = 0$
3. $A \cdot 0 = 0$	9. $\overline{\overline{A}} = A$
4. $A \cdot 1 = A$	10. $A + AB = A$
5. $A + A = A$	11. $A + \overline{A}B = A + B$
6. $A + \overline{A} = 1$	12. $(A + B)(A + C) = A + BC$

- Rules 1 to 9 are obvious.
- Rule 10: A + AB = A

A	B	AB	A + AB
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1



There are also a number of rules to help simplification of Boolean expression.

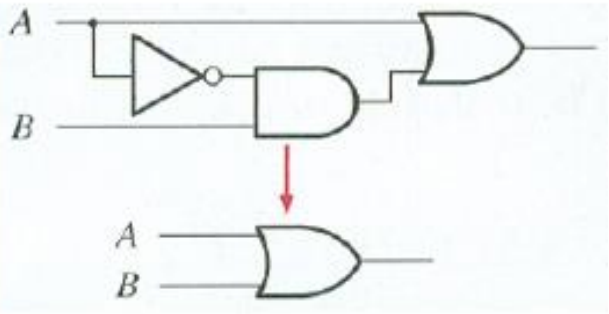
The first 9 rules listed here are obvious.

Rule 10: Less obvious, but it is clearly shown here that it is true.

Rules 11 and 12 of Boolean Algebra

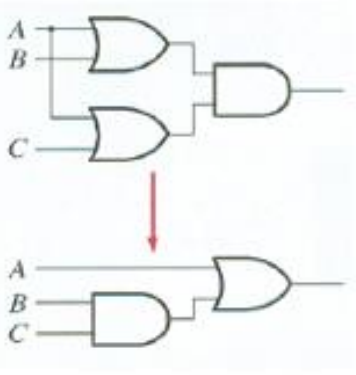
- Rule 11: $A + \overline{A}B = A + B$

A	B	$\overline{A}B$	$A + \overline{A}B$	$A + B$
0	0	0	0	0
0	1	1	1	1
1	0	0	1	1
1	1	0	1	1



- Rule 12: (A + B)(A + C) = A + BC

A	B	C	A + B	A + C	(A + B)(A + C)	BC	A + BC
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	1	1	1	1
1	0	0	1	1	1	0	1
1	0	1	1	1	1	0	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1



Using Boolean Algebra to Simplify Expressions

$$y = A\overline{B}D + A\overline{B}\,\overline{D} \rightarrow y = A\overline{B}$$

$$z = (\overline{A} + B)(A + B) \rightarrow z = B$$

$$x = ACD + \overline{A}BCD \rightarrow x = ACD + BCD$$

DeMorgan's Theorems

- Theorem 1 $\overline{(x+y)} = \overline{x} \cdot \overline{y}$
- Theorem 2 $\overline{(x \cdot y)} = \overline{x} + \overline{y}$

Remember:

"Break the bar, change the operator"

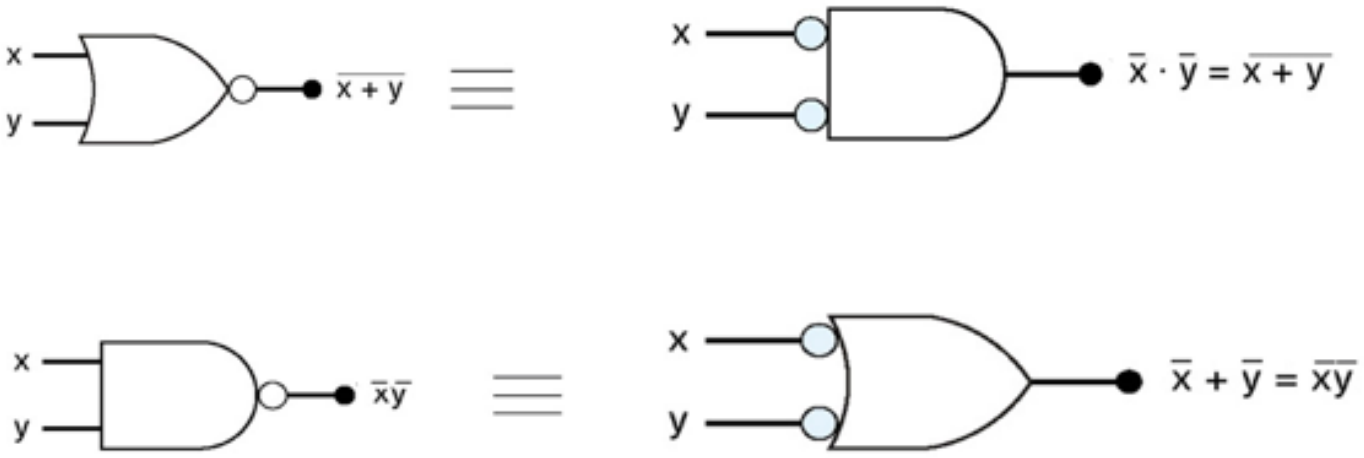
- DeMorgan's theorem is very useful in digital circuit design
- It allows ANDs to be exchanged with ORs by using invertors
- DeMorgan's Theorem can be extended to any number of variables.

$$F = \overline{X.Y} + \overline{P.Q} \longleftarrow \text{2 NAND plus 1 OR}$$

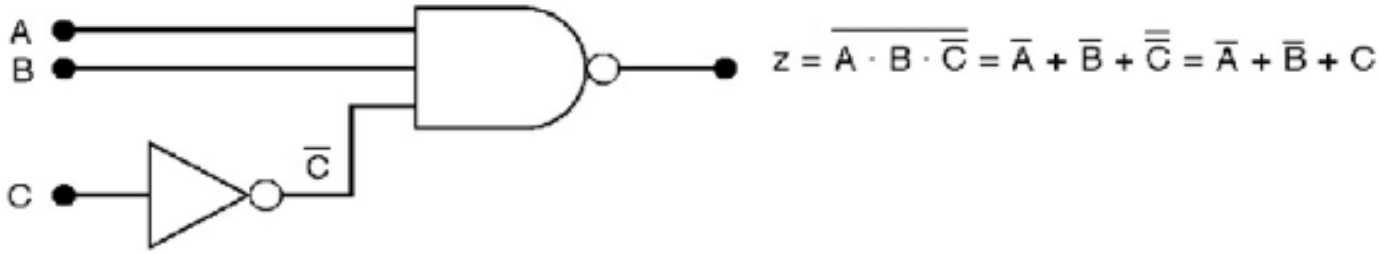
$$= \overline{X} + \overline{Y} + \overline{P} + \overline{Q} \longleftarrow \text{1 OR with some input invertors}$$

$$z = \overline{(\overline{A}+C)} \cdot \overline{(B+\overline{D})} \longrightarrow z = A\overline{C} + \overline{B}D$$

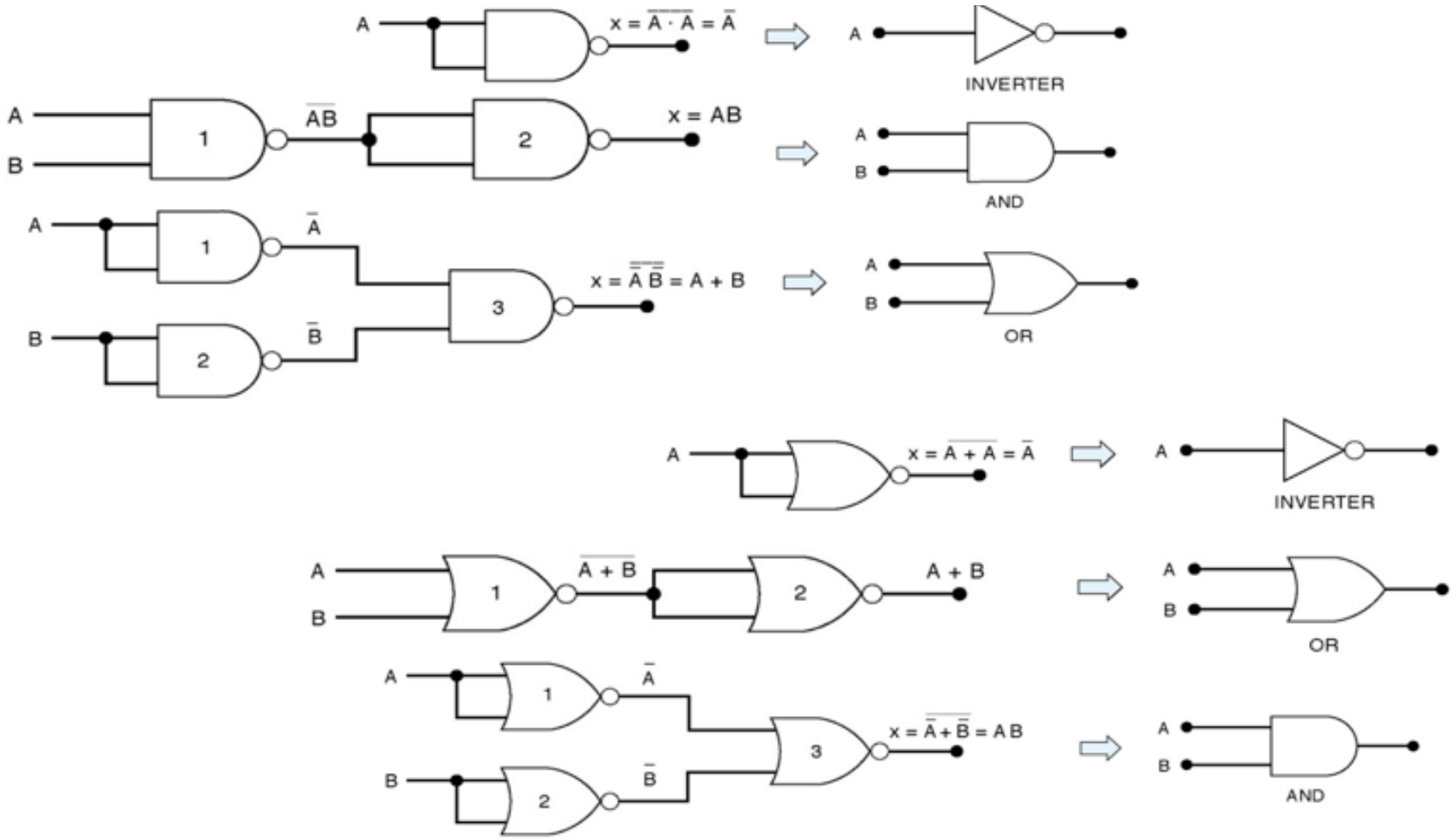
Implications of DeMorgan's Theorems I



- Determine the output expression for the circuit below and simplify it using DeMorgan's Theorem



Universality of NAND and NOR Gates



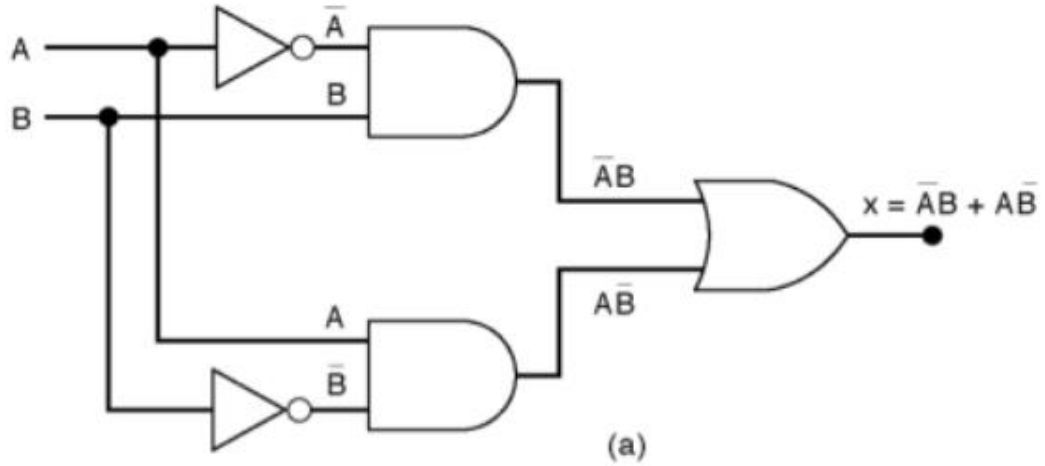
Implications

Let us assume that we ONLY have 2-input NAND gate. From this ,we can get an inverter, an AND gate, and, thanks to De'Morgan, we can also get an OR gate. In other words, if we have a 2-input NAND gate, we can build the three basic logic operators: NOT, AND and OR. As a result, we can build ANY logic circuit and implement any Boolean expression. Taken to limit, give me as many NAND gate as I want, in theory I can build a Pentium processor. This shows the universality of the NAND gate.

Similarly, one can do the same for NOR gates.

Exclusive-OR (XOR)

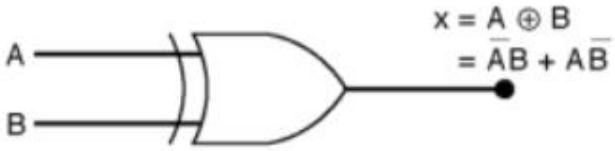
- Exclusive-OR (XOR) produces a HIGH output whenever the two inputs are at opposite levels.



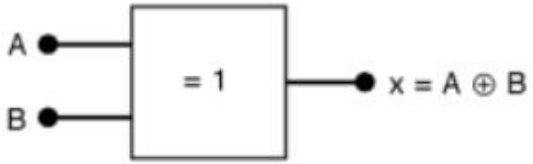
(a)

A	B	x
0	0	0
0	1	1
1	0	1
1	1	0

XOR gate symbols



(b)

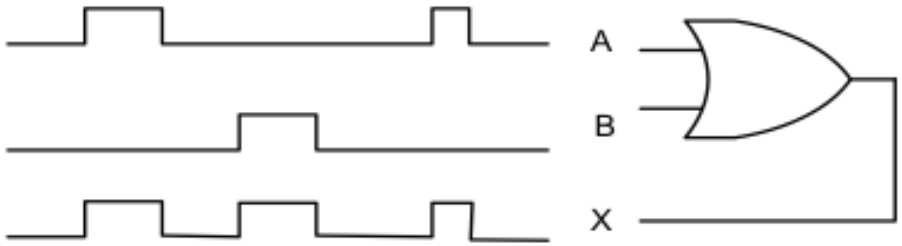


(c)

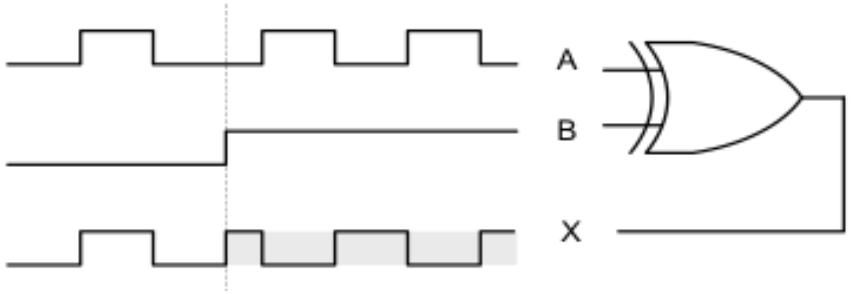
Functional View of Gates

- AND gate function act as enable/disable circuits:-

- OR gate performs signal merging function:-



- XOR gate performs selectable inversion function:-



What Does it Mean?

Now let us take a functional view of logic operators.

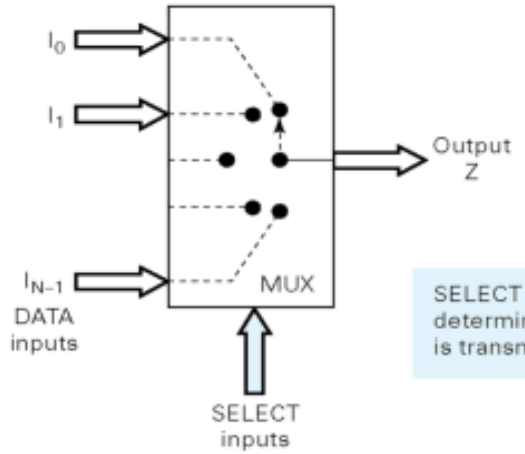
The AND operator can be interpreted as having an enabling or disabling function. (We sometimes call this a 'gating function" as if it perform a gate keeping (i.e blocking) function.) Input B here is a the gating control – if B = 1, it lets A through, otherwise if B = 0, it blocks A.

The OR operator can be interpreted as a merging function. It combines both A and B high level and merge them to form output X.

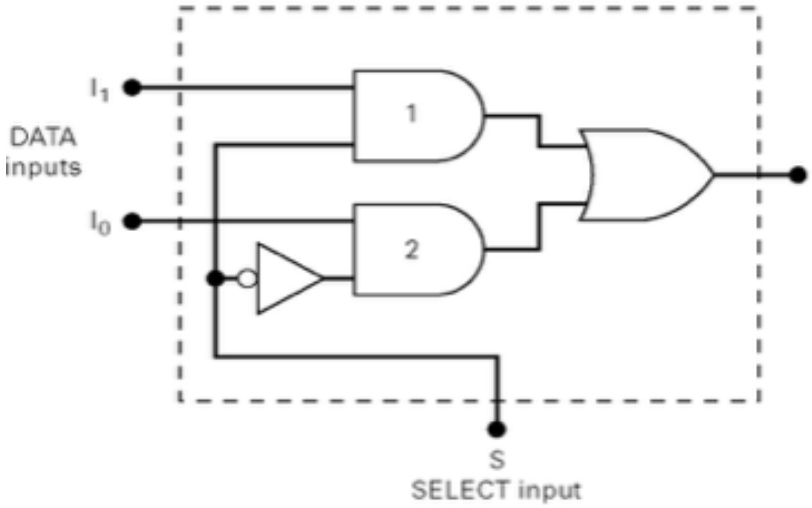
The XOR gate can be viewed as a selectable inverter. If B = 0, A is pass to the output. If B = 1, A is inverted. So B determines inversion, or no inversion of A.

Multiplexers (Data Selectors)

- Multiplexer (MUX) selects one of many input to send to output

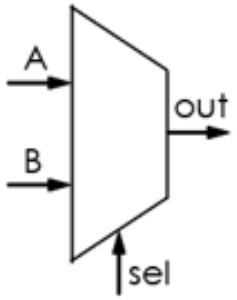


SELECT input code determines which input is transmitted to output Z.

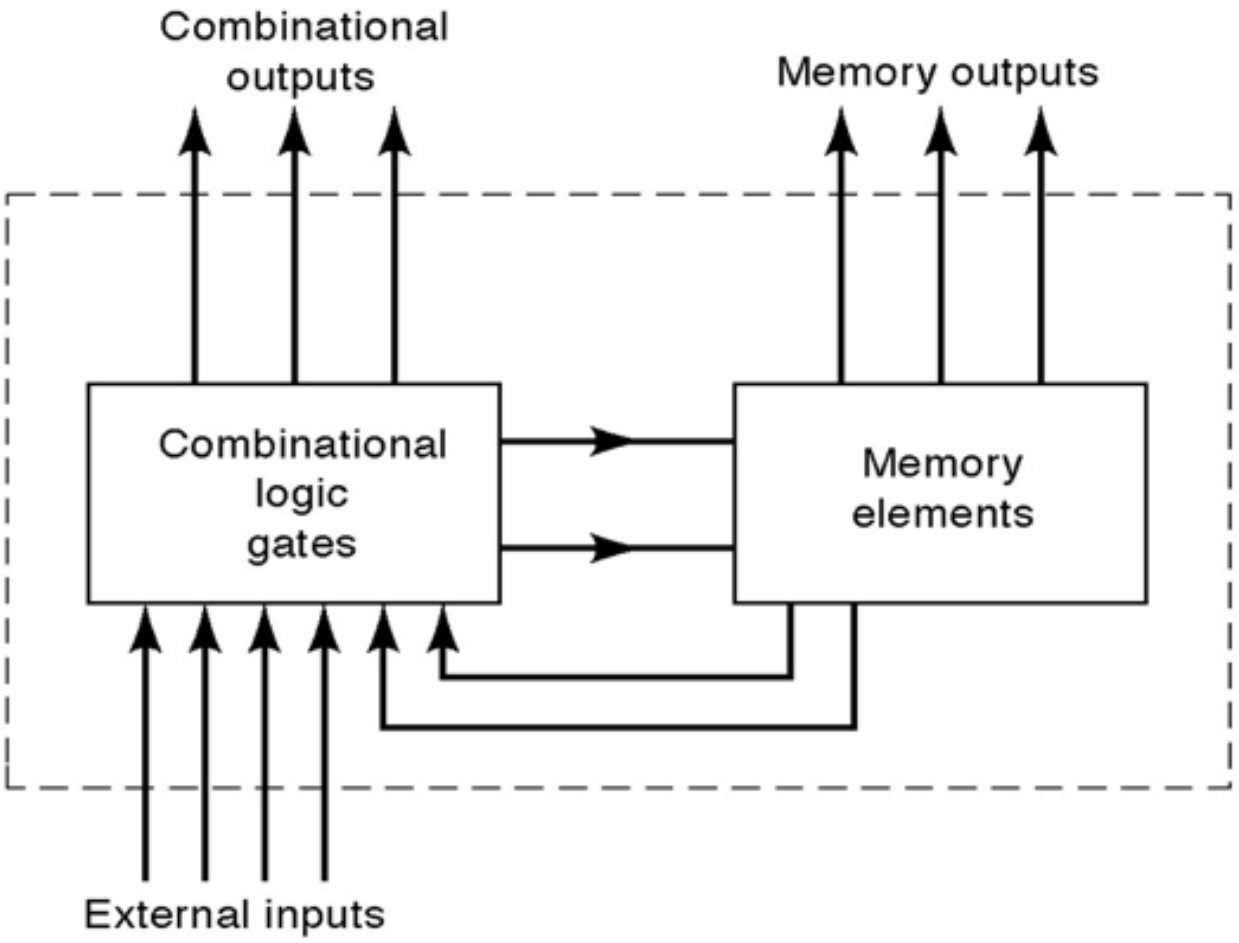


$Z = I_0 \cdot \overline{S} + I_1 \cdot S$

S	Output
0	$Z = I_0$
1	$Z = I_1$



General Digital System Diagram



General Digital System Diagram

Combinational logic gates evaluate Boolean expressions. They can do computation, decoding (e.g. mapping of one binary number to another binary number), selection (such as multiplexers and de-multiplexers) and any function that can be expressed in Boolean expression form. One key characteristic of combinational logic is that outputs completely determined by the input values at a given time. As soon as input changes, soon afterwards, the output will change if necessary. Since the logic gates themselves have delay, the change may happen with some delay.

In other words, combinational logic gates DO NOT HAVE MEMORY (or storage). Its outputs only depend on current inputs and not previous inputs.

Another class of digital circuits, which can be built with gates, have memory. The currently output depends on previous inputs as well as current inputs. These can be built from NAND (or NOR) gates alone. The "memory" property comes from feedback – feeding the outputs of gates back to the inputs.

On this course, we will NOT concern ourselves with how to implement such a storage components using NAND gates. We will just use memory circuits as a building block.

Any digital circuit, no matter how complex, can be model by the combination of the combinational circuit connected with some memory circuits as shown here. This model may look simple, but it is universal!

Properties of Sequential Circuits

- So far we have seen **combinational logic**
  - the output(s) depends only on the current values of the input variables
- Here we will look at **sequential logic** circuits
  - the output(s) can depend on present and also past values of the input and the output variables
- Sequential circuits exist in one of a defined number of states at any one time
  - they move "sequentially" through a defined sequence of transitions from one state to the next
  - The output variables are used to describe the state of a sequential circuit either directly or by deriving state variables from them

Properties of Sequential Circuits

Circuits whose outputs depends on current inputs and past history are called sequential circuits. This is because the circuit will follow some sequences, hence the past is taken into account.

In sequential circuits, the output could take on different values. These are known as states. A sequential circuit will go through these different states in a certain sequence. The exact sequence depends on what happens to the input.

Adding memory into a sequential circuit adds the time domain into the mix. Straight forward Boolean expression is no longer sufficient because a Boolean variable has no notion of time. Therefore when we use Boolean equations to describe behaviour of sequential circuits, we need to somehow introduce the time element.

When a sequential circuit goes from one state (as defined by some of its output) to another state, we call this a "state transition".

Asynchronous and Synchronous Sequential Logic

- Synchronous
  - the timing of all state transitions is controlled by a common clock signal
  - changes in all variables occur simultaneously
- Asynchronous
  - state transitions occur independently of any clock and normally dependent on the timing of transitions in the input variables
  - changes in more than one output do not necessarily occur simultaneously
- Clock
  - A clock signal is a square wave of fixed frequency
  - Often, transitions will occur on one of the edges of clock pulses
    - i.e. the rising edge or the falling edge

Asynchronous and Synchronous Sequential Logic

In sequential circuits, if the transition from a state to another state is governed by a single regular, repetitive, clock signal, we call this a **synchronous circuit**. All changes in such circuits are synchronous (and governed) by the clock signal. Almost all digital circuits in the real-world are synchronous. For example, when you buy a computer with an Intel processor running at 3GHz, this number refers to the clock signal frequency that determines how the processor perform its Boolean evaluation in the logic gates on the chip.

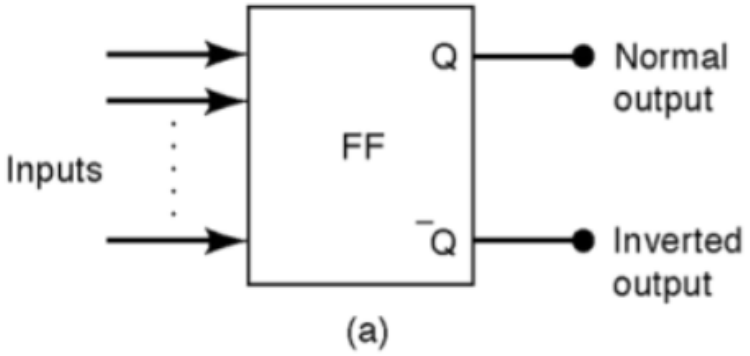
Occasionally, some digital circuits goes from one state to another state NOT governed by a clock signal. These are called **asynchronous circuits**.

In theory, we can implement any logic function in either the synchronous or asynchronous way. However, using a clock signal to mark when state transitions should occur makes life much easier for designer, and generally results in most faster circuits at the system level.

On this course, we will only consider sequential circuits that are synchronous to a single clock signal.

Flip-Flops

- Flip-flops are the fundamental element of sequential circuits
- Flip-flops are essentially 1-bit storage devices
  - outputs can be set to store either 0 or 1 depending on the inputs
  - even when the inputs are de-asserted, the outputs retain their prescribed value
- Flip-flops often have 2 complimentary outputs: Q and $\overline{Q}$



(a)

Output states	
Q = 1, $\overline{Q}$ = 0:	called HIGH or 1 state; also called SET state
Q = 0, $\overline{Q}$ = 1:	called LOW or 0 state; also called CLEAR or RESET state

(b)

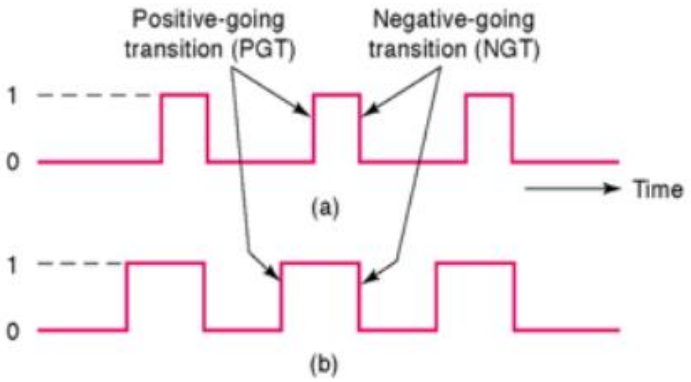
The most basic sequential circuit is a flip-flop (FF). The circuit can store only two states: '0' or '1'. Unlike combinational logic gates, a flip-flop has memory. It is effectively a 1-bit storage element.

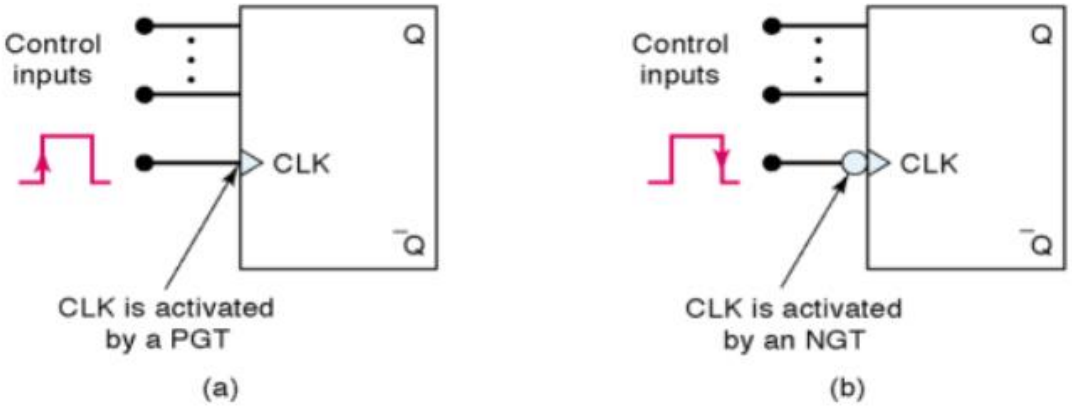
Many FFs have two outputs Q and $\overline{Q}$.

All FFs we considered on this course have a clock and a data signal input.

Clock Signals and Clocked FFs

- Digital systems can operate
  - Asynchronously: output can change state whenever inputs change
  - Synchronously: output only change state at clock transitions (edges)
- Clock signal
  - Outputs change state at the transition (edge) of the input clock
  - Positive-going transitions (PGT)
  - Negative-going transitions (NGT)

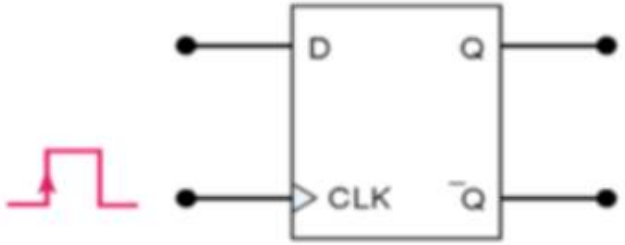




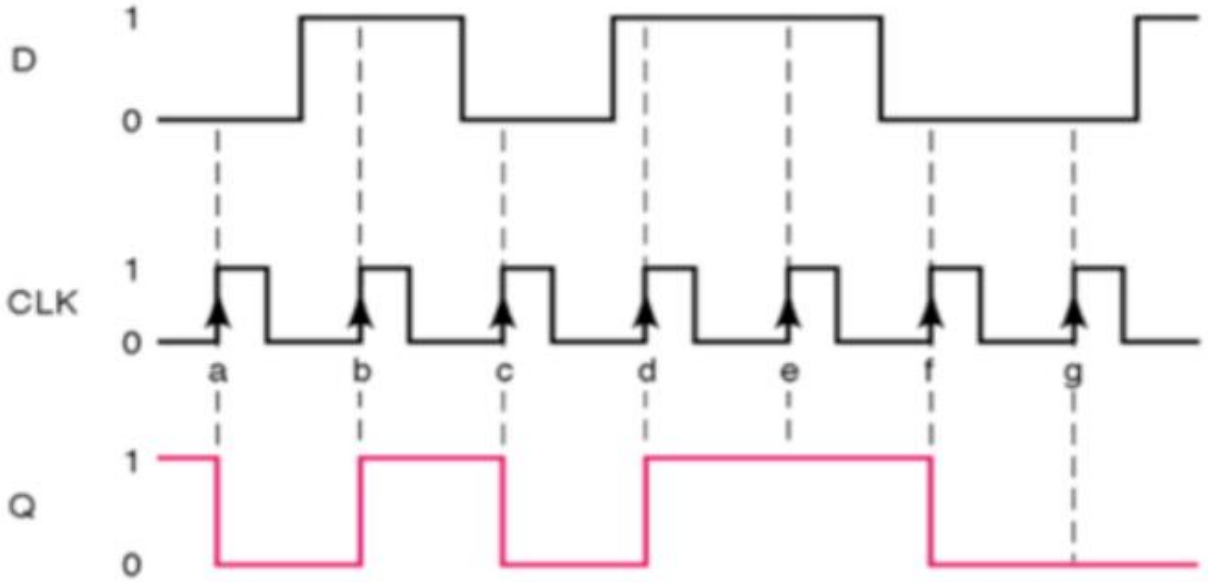
All synchronous circuits use a clock to determine when they change states. The change could occur on the rising edge (positive going) or the falling edge (negative going) of the clock signal. Such "edge-triggered" flip-flops have a small triangle next to the clock wire. Further, we use a circuit to indicate a FF that is triggered on the falling (negative) edge of the clock.

Clocked D Flip-Flop

- D-FF that triggers only on positive-going transitions:



D	CLK	Q
0	↑	0
1	↑	1



Clocked D Flip-Flop

In all textbooks, they go through many different TYPES of flip-flops. Actually, the only useful type and is found almost universally in all digital circuits is the D-type flip-flop (D-FF). (D stands for data.)

The truth table specifying the behaviour of a D-FF is shown here. The D-FF stores the current state (i.e. Q is set or '1' or reset or '0'). D input can change as much as it likes, Q does not respond.

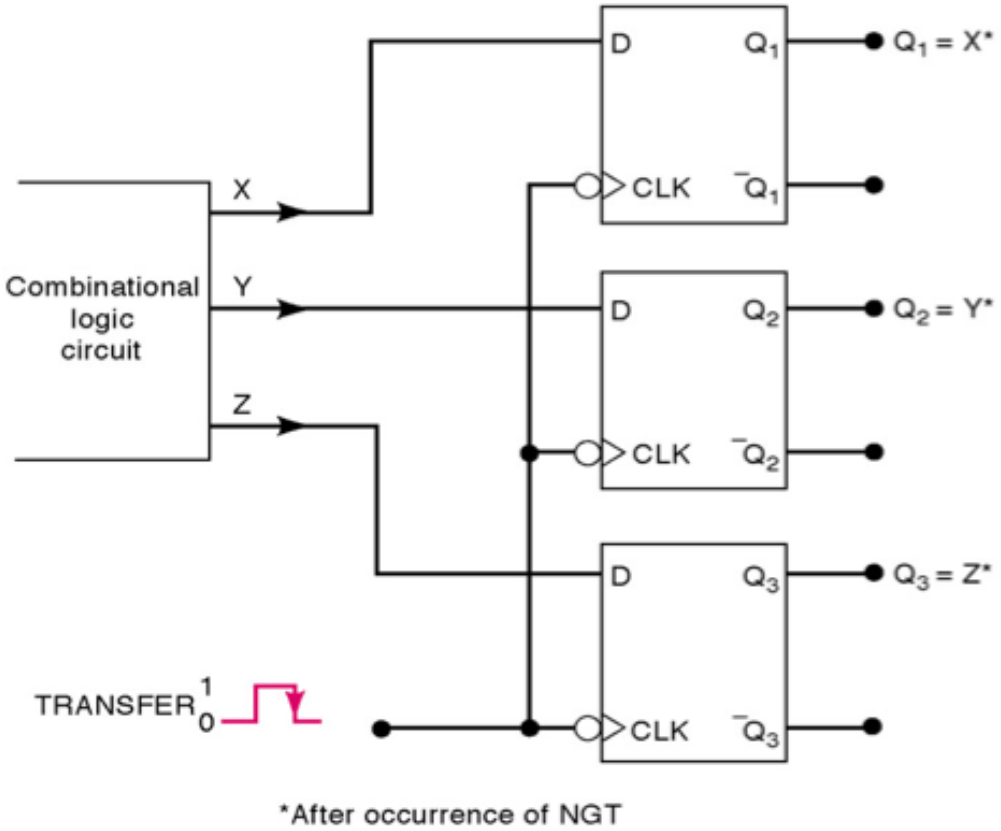
However, when CLK input goes from low to high (positive edge), the logic value of D is sampled and stored in the D-FF, and Q changes.

This is like taking a photograph. The scene (i.e input D) may change, and it does not affect the picture (Q output) until you press the trigger button (the CLK signal).

You can therefore view the D-FF in two ways:

1. It capture the data input on the command of CLK, and keep this data at the output Q for use by other circuits.
2. It blocks any changes on the data input and ignore them. These changes (such as glitches from combinational gates) will be ignore until the active edge of the clock comes along.

Parallel Data Transfer Using D-FF

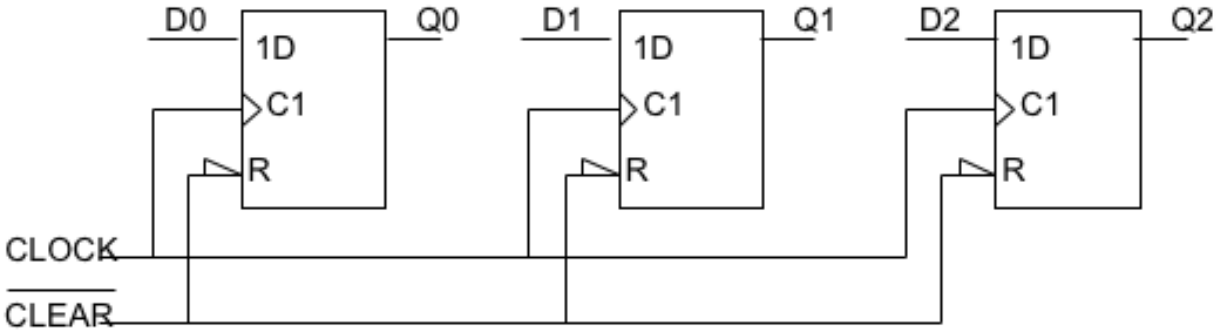


*After occurrence of NGT

D-FFs are usually used to "register" or "latch" logic values from combinational logic circuits. Shown here are three signals X, Y, Z from logic gates. The logic gates perform Boolean computation and, after some delay, reach final values. These values are not visible on the outputs Q1, Q2 and Q3 until a negative edge occurs on the CLK input. At that moment, the values of X, Y and Z are captured by the three D-FFs, and stored. These values are presented on the Q outputs (and its inverse).

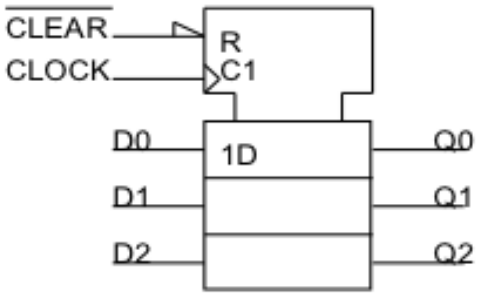
Registers

- A register is a digital electronic device capable of storing several bits of data
  - Normally made from D-type flip-flops with asynchronous RESET inputs
  - Operates on the bits of the data word in parallel (parallel in / parallel out)
- Operation
  - Data on each data input is stored in the flip-flop on the rising edge of CLOCK
  - The data can be read from the Q outputs
  - New data can be reloaded by re-CLOCKing the register
  - The register can be cleared (zeroed) by asserting the CLEAR inputs



A 3-bit register

Symbol

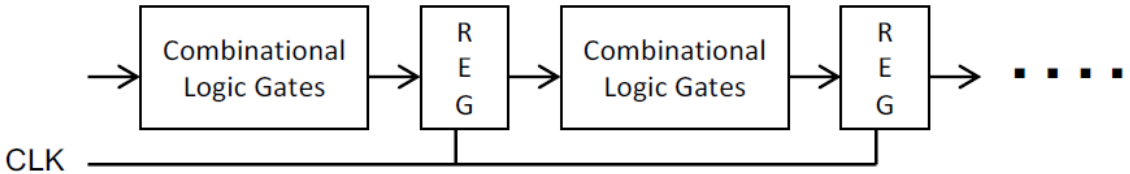


Registers

If you combine a number of D-FFs together as shown here (for three D-FFs), all driven by the same CLOCK signals, but have separate D inputs, we form a useful digital building block called a **register**.

You will find registers in all microprocessors and microcontrollers. They are used to store a temporary variables whose values are then fed to adders or other computational circuits (which are the combinational logic gates).

In many digital systems, you will find combinational logic gates and registers interleaving each other, one after another like this:

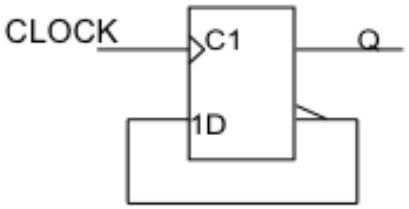


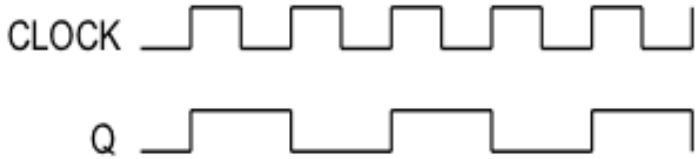
This way of organising digital circuit is known as "pipelining". It is the same idea as a production pipeline where the registers act as shelves (storage units) between teams of workers or machines doing the assembly (logic gates). The CLK signal makes the transition form one stage of the pipeline to the next.

Shown above is a compact symbol for a three bit registers. The CLEAR signal is low active meaning that it is normally '1'. When it goes low, the content of the D-FF are all reset (i.e. zeroed).

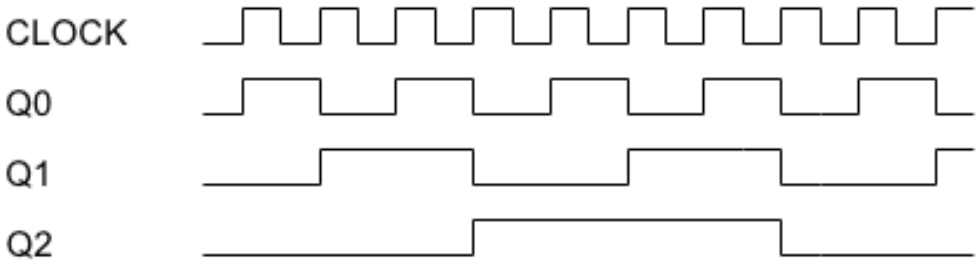
Divide Frequency By 2 Circuits

- Consider a D-type flip-flop with $\overline{Q}$ connected to D





- D is always the inverse of Q hence Q will always toggle on a rising clock edge
- The frequency of Q is half the frequency of CLOCK
- Example: 3-bit counter



- $f_{Q1} = f_{Q0}/2, \quad f_{Q2} = f_{Q1}/2$

Decimal	Q2	Q1	Q0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Divide Frequency By 2 Circuits

If we connect the not_Q output back to the D input of the D-FF, we have a frequency divide-by-2 circuit. The D-FF acts as a toggle: on each rising edge of the clock, the output changes states (i.e. toggles). This results in Q having a frequency half that of CLOCK.

If you now use the Q output of one D-FF as the CLOCK input to the next, we can implement a binary counter as shown here (Q0 is the least significant bit or LSB).

We now divide the frequency of CLOCK, by 2, then by 4 and then by 8 etc.