

# CSE 2021 Computer Organization

## **Appendix C**

### **The Basics of Logic Design**

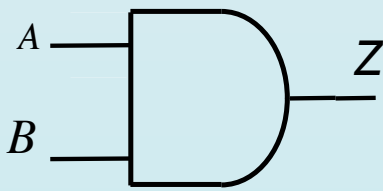
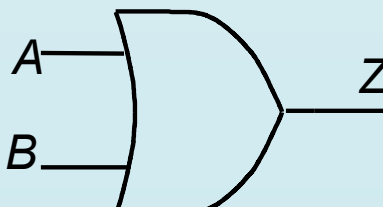
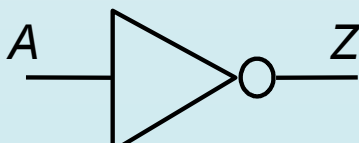
# Outline

- Fundamental Boolean operations
- Deriving logic expressions from truth tables
- Boolean Identities
- Simplifying logic expressions using Boolean identities
- Combinational and sequential circuits
- Verilog basics

# Boolean Algebra

- Boolean algebra is the basic math used in digital circuits and computers.
- A Boolean variable takes on only 2 values:  $\{0,1\}$  ,  $\{T,F\}$ ,  $\{Yes, No\}$ , etc.
- There are 3 fundamental Boolean operations:
  - AND, OR, NOT

# Fundamental Boolean Operations

AND	OR	NOT																																				
																																						
$Z=A*B \text{ (AB)}$	$Z=A+B$	$Z=\bar{A}$																																				
Truth Table	Truth Table	Truth Table																																				
<table><tr><th>A</th><th>B</th><th>Z</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Z	0	0	0	0	1	0	1	0	0	1	1	1	<table><tr><th>A</th><th>B</th><th>Z</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Z	0	0	0	0	1	1	1	0	1	1	1	1	<table><tr><th>A</th><th>Z</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	Z	0	1	1	0
A	B	Z																																				
0	0	0																																				
0	1	0																																				
1	0	0																																				
1	1	1																																				
A	B	Z																																				
0	0	0																																				
0	1	1																																				
1	0	1																																				
1	1	1																																				
A	Z																																					
0	1																																					
1	0																																					

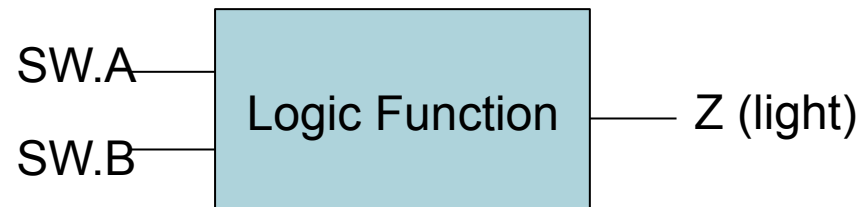
# Boolean Algebra

- A *truth table* specifies output signal logic values for every possible combination of input signal logic values
- In evaluating Boolean expressions, the *Operation Hierarchy* is: 1) NOT 2) AND 3) OR. Order can be superseded using ( ... )
- *Example:*  $A = T, B = F, C = T, D = T$
- What is the value of  $Z = (\bar{A} + B) \cdot (C + \bar{B} \cdot D)$ ?

$$\begin{aligned} Z &= (\bar{T} + F) \cdot (C + \bar{B} \cdot D) = (F + F) \cdot (C + \bar{B} \cdot D) \\ &= F \cdot (C + \bar{B} \cdot D) = F \end{aligned}$$

# Deriving Logic Expressions From Truth Tables

Light must be ON when both switches A and B are OFF, or when both of them are ON.



*Truth Table:*

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

- *What is the Boolean expression for Z?*

# Minterms and Maxterms

- Minterms
  - AND term of all input variables.
  - For variables with value 0, apply complements
- Maxterms
  - OR factor with all input variables
  - For variables with value 1, apply complements

A	B	Z	Minterms	Maxterms
0	0	1	$\bar{A} \cdot \bar{B}$	$A + B$
0	1	0	$\bar{A} \cdot B$	$A + \bar{B}$
1	0	0	$A \cdot \bar{B}$	$\bar{A} + B$
1	1	1	$AB$	$\bar{A} + \bar{B}$

# Minterms and Maxterms

- A function with  $n$  variables has  $2^n$  minterms (and Maxterms) – exactly equal to the number of rows in truth table
- Each minterm is true for exactly one combination of inputs
- Each Maxterm is false for exactly one combination of inputs

A	B	Z	Minterms	Maxterms
0	0	1	$\bar{A} \cdot \bar{B}$	$A + B$
0	1	0	$\bar{A} \cdot B$	$A + \bar{B}$
1	0	0	$A \cdot \bar{B}$	$\bar{A} + B$
1	1	1	$AB$	$\bar{A} + \bar{B}$

# Equivalent Logic Expressions

- Two equivalent logic expressions can be derived from Truth Tables:

1. *Sum-of-Products* (SOP) expressions:

- Several AND terms OR'd together, e.g.

$$\overline{A}\overline{B}C + \overline{A}B\overline{C} + ABC$$

2. *Product-of-Sum* (POS) expressions:

- Several OR terms AND'd together, e.g.

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

# Rules for Deriving SOP Expressions

1. Find each row in TT for which output is 1 (rows 1 & 4)
2. For those rows write a **minterm** of all input variables.
3. OR together all **minterms** found in (2):  
Such an expression is called a *Canonical SOP*

A	B	Z	Minterms	Maxterms
0	0	1	$\bar{A} \cdot \bar{B}$	$A + B$
0	1	0	$\bar{A} \cdot B$	$A + \bar{B}$
1	0	0	$A \cdot \bar{B}$	$\bar{A} + B$
1	1	1	$AB$	$\bar{A} + \bar{B}$

$$Z = \bar{A} \bar{B} + AB$$

# Rules for Deriving POS Expressions

1. Find each row in TT for which output is 0 (rows 2 & 3)
2. For those rows write a **maxterm**
3. AND together all **maxterm** found in (2):  
Such an expression is called a *Canonical POS*.

A	B	Z	Minterms	Maxterms
0	0	1	$\bar{A}.\bar{B}$	$A + B$
0	1	0	$\bar{A}.B$	$A + \bar{B}$
1	0	0	$A.\bar{B}$	$\bar{A} + B$
1	1	1	$AB$	$\bar{A} + \bar{B}$

$$Z = (A + \bar{B})(\bar{A} + B)$$

# CSOP and CPOS

- Canonical SOP:  $Z = \bar{A}\bar{B} + AB$
- Canonical POS:  $Z = (A + \bar{B})(\bar{A} + B)$
- Since they represent the same truth table, they should be identical

Verify that

$$Z = \bar{A}\bar{B} + AB \equiv (A + \bar{B})(\bar{A} + B)$$

- *CPOS and CSOP expressions for the same TT are logically equivalent. Both represent the same information.*

# Activity 1

Derive SOP and POS expressions for the following TT.

A	B	Carry
0	0	0
0	1	0
1	0	0
1	1	1



# **Boolean Identities**

# Boolean Identities

- Useful for simplifying logic equations.

	(a)	(b)
1	$\overline{\overline{A}} = A$	$\overline{\overline{A}} = A$
2	$A + \text{false} = A \quad (A + 0 = A)$	$A \cdot \text{true} = A \quad (A \cdot 1 = A)$
3	$A + \text{true} = \text{true} \quad (A + 1 = 1)$	$A \cdot \text{false} = \text{false} \quad (A \cdot 0 = 0)$
4	$A + A = A$	$A \cdot A = A$
5	$A + \overline{A} = \text{true} \quad (A + \overline{A} = 1)$	$A \cdot \overline{A} = \text{false} \quad (A \cdot \overline{A} = 0)$
6	$A + B = B + A$	$A \cdot B = B \cdot A$
7	$A + B + C = (A + B) + C = A + (B + C)$	$A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$
8	$A \cdot (B + C) = A \cdot B + A \cdot C$	$A + \overline{B \cdot C} = (\overline{A + B})(\overline{A + C})$
9	$\overline{A + B} = \overline{A} \cdot \overline{B}$	$\overline{A \cdot B} = \overline{A} + \overline{B}$
10	$A \cdot B + A \cdot \overline{B} = A$	$(A + B)(A + \overline{B}) = A$
11	$A + A \cdot B = A$	$A(A + B) = A$
12	$A(\overline{A} + B) = A \cdot B$	$A + \overline{A} \cdot B = A + B$
13	$A \cdot B + \overline{A} \cdot C + B \cdot C = A \cdot B + \overline{A} \cdot C$	$(A + B)(\overline{A} + C)(B + C) = (A + B)(\overline{A} + C)$

Duals

# Boolean Identities

Identities	Property
1-5	Single variable, foundations of Boolean manipulation
6	Commutative
7	Associative
8	Distributive
9	De Morgan's
10	Combining
11	Absorption
13	Consensus

# Boolean Identities

- The right side is the dual of the left side

1. Duals formed by replacing

AND  $\rightarrow$  OR

OR  $\rightarrow$  AND

0  $\rightarrow$  1

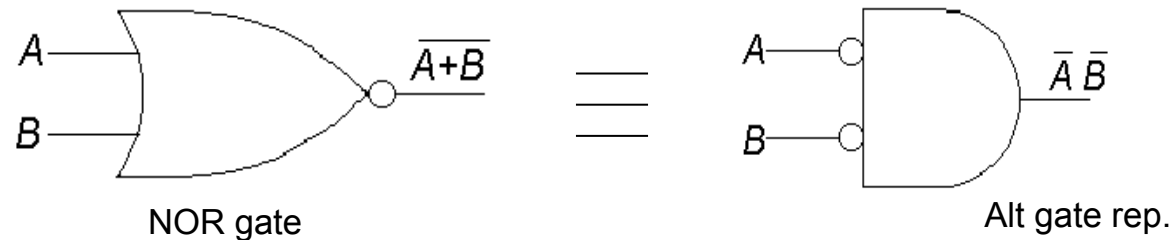
1  $\rightarrow$  0

2. The dual of any true statement in Boolean algebra is also a true statement.

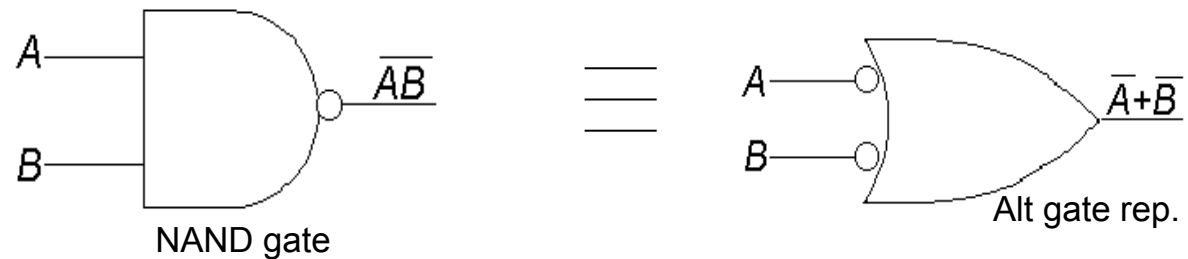
# Boolean Identities

- DeMorgan's laws very useful: 9a and 9b

$$\overline{A+B} = \overline{A}.\overline{B}$$



$$\overline{AB} = \overline{A} + \overline{B}$$



## Activity 2

Proofs of some Identities:

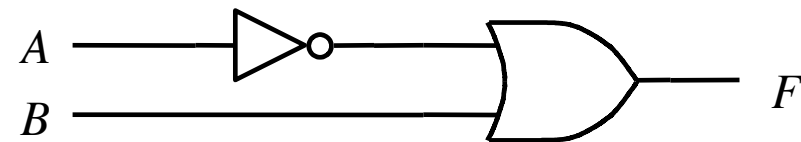
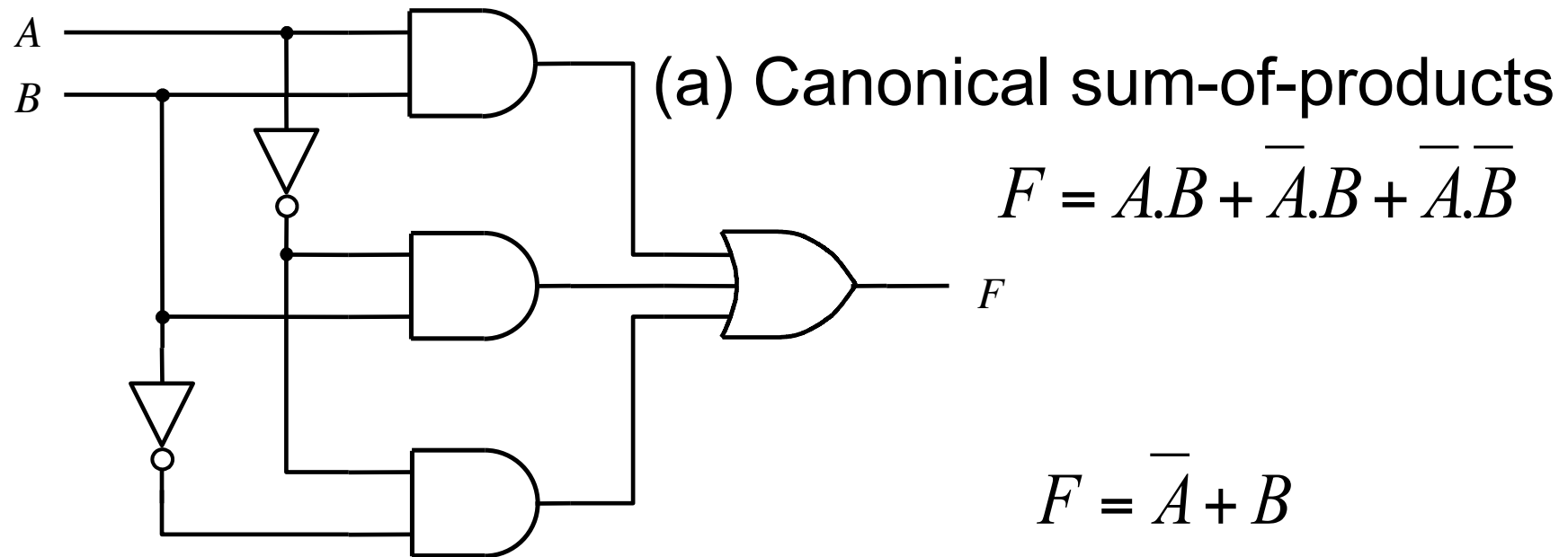
$$12b: \quad A + \overline{A}B = A + B$$

$$13a: \quad AB + \overline{A}C + BC = AB + \overline{A}C$$



# **Simplifying Logic Expressions Using Boolean Identities**

# Simplifying Logic Equations – Why?



(b) Minimal-cost realization

# Simplifying Logic Equations

- Simplifying logic expressions can lead to using smaller number of gates (parts) to implement the logic expression
- Can be done using
  - Boolean Identities (algebraic)
  - Karnaugh Maps (graphical)
- A *minimum SOP* (MSOP) expression is one that has no more AND terms or variables than any other equivalent SOP expression.
- A *minimum POS* (MPOS) expression is one that has no more OR factors or variables than any other equivalent POS expression.
- There may be several MSOPs of an expression

# Example of Using Boolean Identities

- Find an MSOP for

$$F = \overline{X}W + Y + \overline{Z}(Y + \overline{X}W)$$

$$= \overline{X}W + Y + \overline{Z}Y + \overline{Z}\overline{X}W$$

$$= \overline{X}W(1 + \overline{Z}) + Y(1 + \overline{Z})$$

$$= \overline{X}W + Y$$

## Activity 3

- Find an MSOP for

$$F = V\overline{W}XY + VWYZ + V\overline{X}YZ$$

# CSE 2021 Computer Organization

## **Combinational and Sequential Circuits**

# Digital Circuit Classification

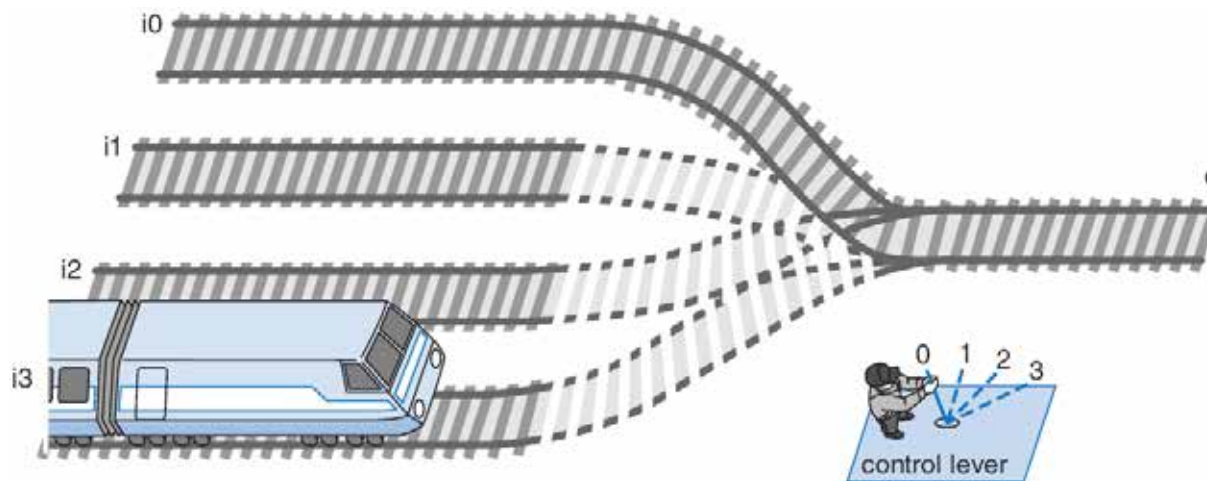
- Combinational circuits
  - Output depends only solely on the current combination of circuit inputs
  - Same set of input will always produce the same outputs
  - Consists of AND, OR, NOR, NAND, and NOT gates
- Sequential circuits
  - Output depends on the current inputs and state of the circuit (or past sequence of inputs)
  - Memory elements such as flip-flops and registers are required to store the “state”
  - Same set of input can produce completely different outputs

# CSE 2021 Computer Organization

## **Combinational Circuits**

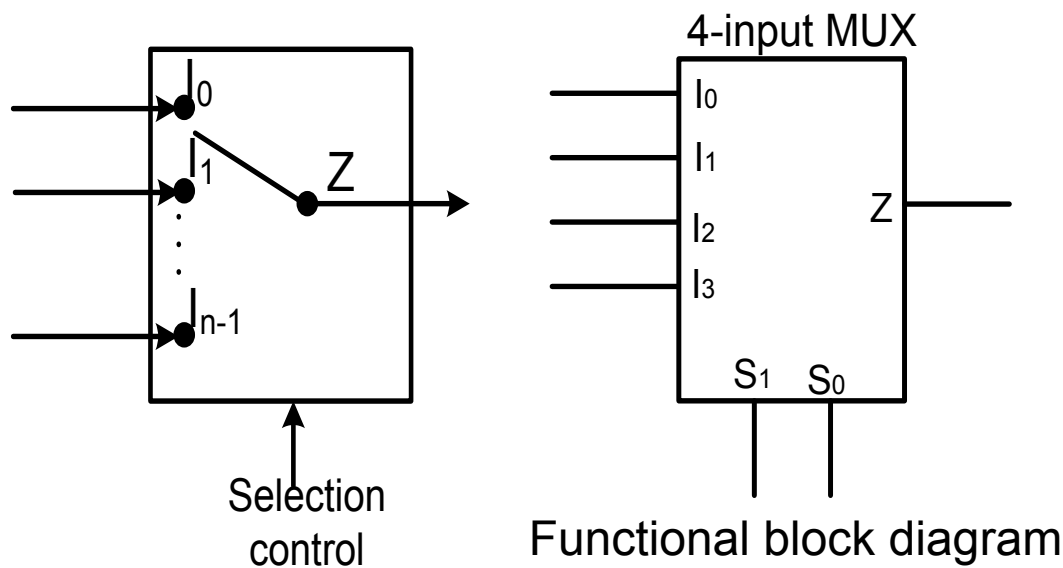
# Multiplexer

- A multiplexer (MUX) selects data from one of  $N$  inputs and directs it to a single output, just like a railyard switch
  - 4-input Mux needs 2 select lines to indicate which input to route through
  - $N$ -input Mux needs  $\log_2(N)$  selection lines



# Multiplexer (2)

- An example of 4-input Mux



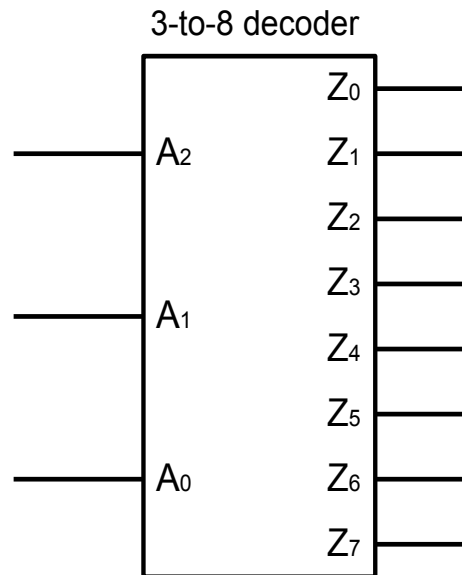
Actual truth table would have  $2^6$  rows corresponding to  $I_0$ ,  $I_1$ ,  $I_2$ ,  $I_3$ ,  $S_0$  and  $S_1$

$S_1$	$S_0$	$Z$
0	0	$I_0$
0	1	$I_1$
1	0	$I_2$
1	1	$I_3$

Condensed truth table

# Decoder

- A decoder is a circuit element that will decode an  $N$ -bit code.
- It activates an appropriate output line as a function of the applied  $N$ -bit input code



Functional block diagram

Truth Table

$A_2$	$A_1$	$A_0$	$Z_0$	$Z_1$	$Z_2$	$Z_3$	$Z_4$	$Z_5$	$Z_6$	$Z_7$
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

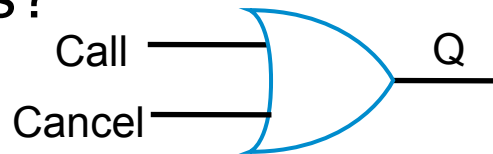
# CSE 2021 Computer Organization

## **Sequential Circuits**

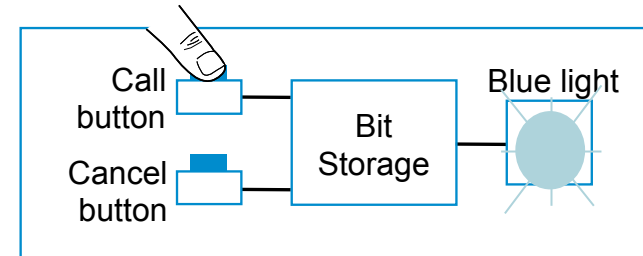
# Why Bit Storage ?

3.2

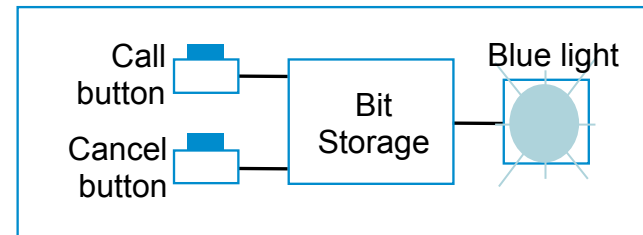
- Flight attendant call button
  - Press call: light turns on
    - **Stays on** after button released
  - Press cancel: light turns off
  - Logic gate circuit to implement this?



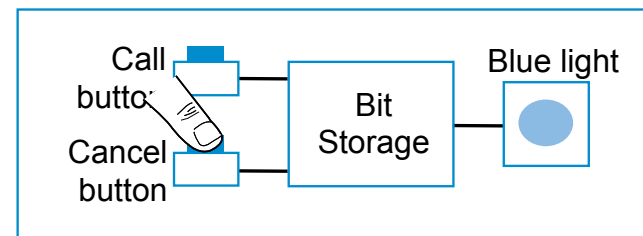
<sup>a</sup> Doesn't work.  $Q=1$  when  $Call=1$ , but doesn't stay 1 when  $Call$  returns to 0  
*Need some form of "memory" in the circuit*



1. Call button pressed – light turns on



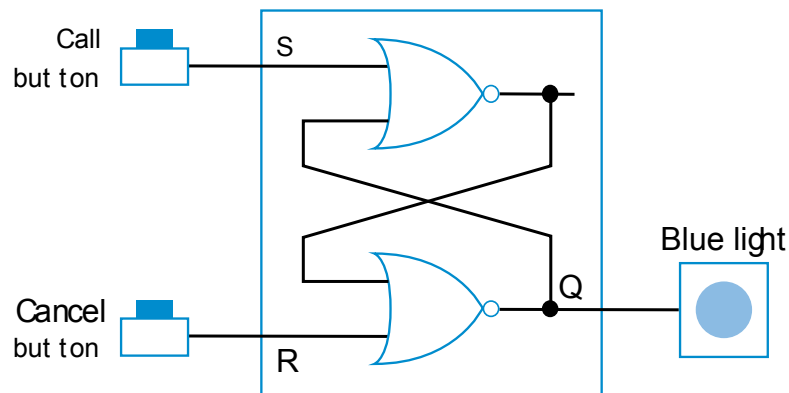
2. Call button released – light stays on



3. Cancel button pressed – light turns off

# Bit Storage Using SR Latch

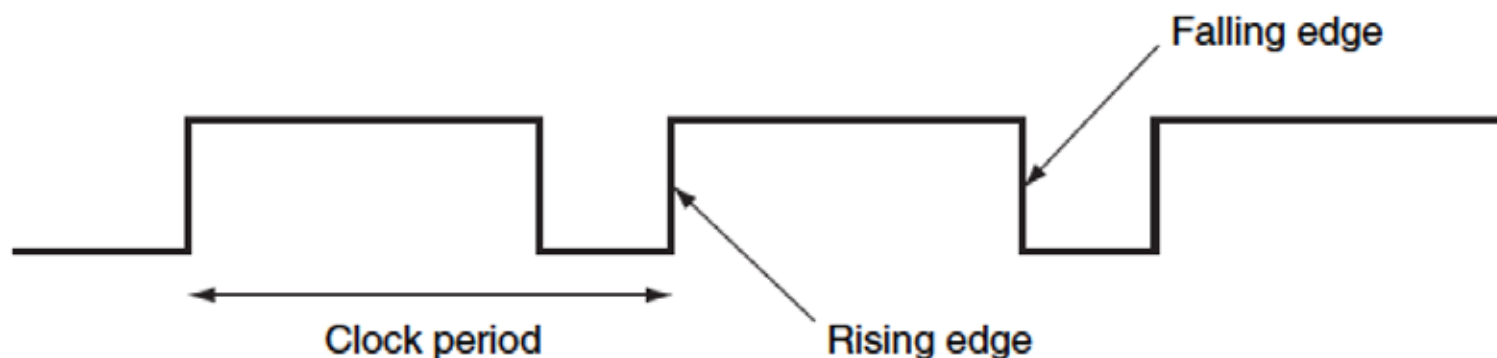
- Simplest memory elements are Latch and Flip-Flops
- SR (set-reset) latch is an ***un-clocked*** latch
  - Output  $Q=1$  when  $S=1$ ,  $R=0$  (set condition)
  - Output  $Q=0$  when  $S=0$ ,  $R=1$  (reset condition)
  - Problem -  $Q$  is undefined if  $S=1$  and  $R=1$



# Clocks

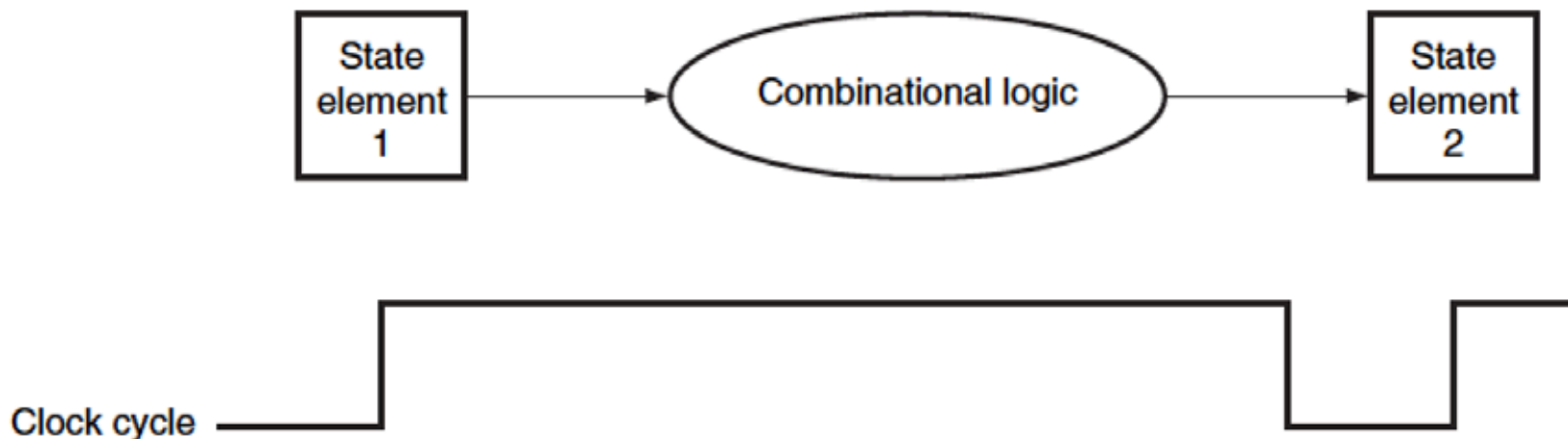
- **Clock period:** time interval between pulses
  - example: period = 20 ns
- **Clock frequency:** 1/period
  - example: frequency =  $1 / 20 \text{ ns} = 50 \text{ MHz}$
- **Edge-triggered clocking:** all state changes occur on a clock edge.

Freq	Period
100 GHz	0.01 ns
10 GHz	0.1 ns
1 GHz	1 ns
100 MHz	10 ns
10 MHz	100 ns



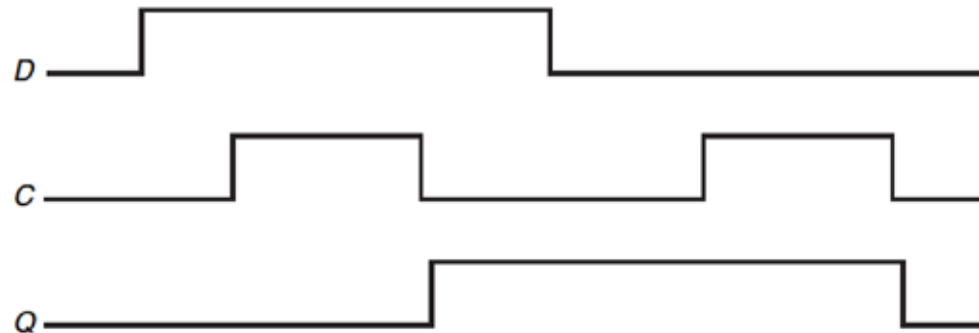
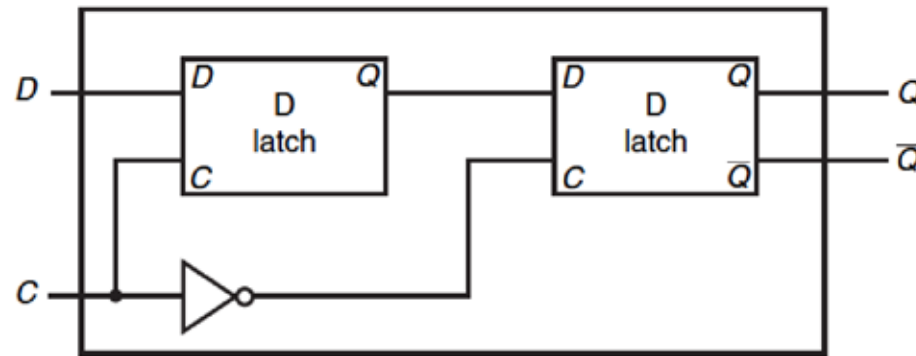
# Clock and Change of State

- Clock controls when the state of a memory element changes
- ***Edge-triggered clocking***: all state changes occur on a clock edge.



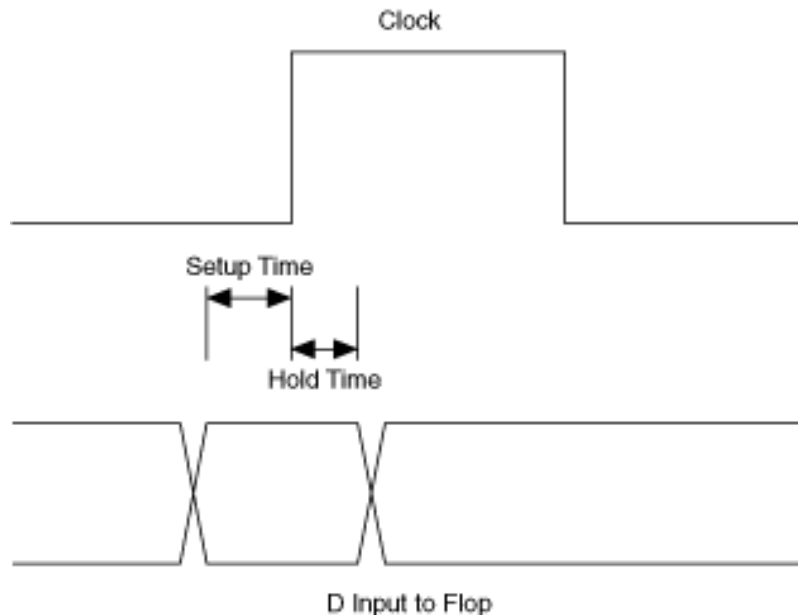
# Clock Edge Triggered Bit Storage

- **Flip-flop** - Bit storage that stores on clock edge, not level
- D Flip-flop
  - Two latches, master and slave latches.
  - Output of the first goes to input of second, slave latch has inverted clock signal (falling-edge trigger)



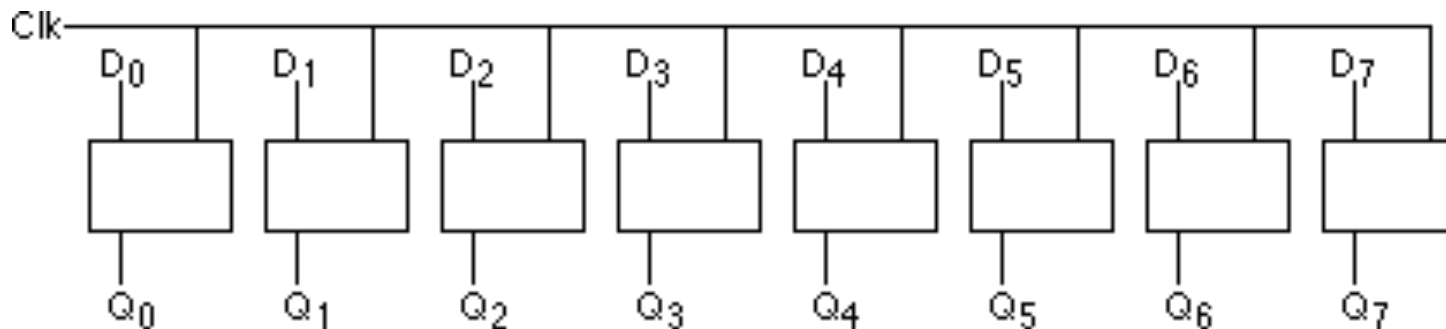
# Setup and Hold Time

- Setup time
  - The minimum amount of time the data signal should be held steady before the clock edge arrives.
- Hold time
  - The minimum amount of time the data signal should be held steady after the clock edge.



# N-Bit Register

- Cascade  $N$  number of D flip-flops to form a  $N$ -bit register
- An example of 8-bit register formed by 8 edge-triggered D flip-flops



# CSE 2021 Computer Organization

## **Verilog Basics**

# What is an HDL?

- A Hardware Description Language (HDL) is a software programming language used to model the intended operation of a piece of hardware.
- The difference between an HDL and “C”
  - Concurrency
  - Timing
- A powerful feature of the Verilog HDL is that we can use the same language for describing, testing and debugging the system.

# An Example

```
module pound_one;
reg [7:0] a,a$b,b,c; // register declarations
reg clk;

initial
begin
    clk=0; // initialize the clock
    c = 1;
    forever #25 clk = !clk;
end
/* This section of code implements
a pipeline */
always @ (posedge clk)
begin
    a = b;
    b = c;
end
endmodule
```

# Identifiers

- Identifiers are names assigned by the user to Verilog objects such as modules, variables, tasks etc.
- An identifier may contain any sequence of letters, digits, a dollar sign '\$' , and the underscore '\_' symbol.
- The first character of an identifier must be a letter or underscore; it cannot be a dollar sign '\$' , for example. We cannot use characters such as '-' (hyphen), brackets, or '#' in Verilog names (**escaped identifiers** are an exception).

# Escaped Identifiers

- The use of escaped identifiers allow any character to be used in an identifier.
  - Escaped identifiers start with a backslash (\) and end with white space (White space characters are space, tabs, carriage returns).
  - Gate level netlists generated by EDA tools (like DC) often have escaped identifiers
- Examples:
  - \clock = 0;
  - \a\*b = 0;
  - \5-6
  - \bus\_a[0]
  - \bus\_a[1]

**module** identifiers; /\* Multiline comments in Verilog look like C comments  
and // is OK in here. \*/

// Single-line comment in Verilog.

**reg** **legal\_identifier**, **two\_\_underscores**;

**reg** **\_OK,OK\_,OK\_\$,OK\_123,CASE\_SENSITIVE, case\_sensitive**;

**reg** **Vclock** ,\a\*b ; // Add white\_space after escaped identifier.

//reg \$\_BAD,123\_BAD; // Bad names even if we declare them!

**initial begin**

legal\_identifier = 0; // Embedded underscores are OK,

two\_\_underscores = 0; // even two underscores in a row.

\_OK = 0; // Identifiers can start with underscore

OK\_ = 0; // and end with underscore.

OK\$ = 0; // \$ sign is OK.

OK\_123 = 0; // Embedded digits are OK.

CASE\_SENSITIVE = 0; // Verilog is case-sensitive (unlike VHDL).

case\_sensitive = 1;

Vclock = 0; // An escaped identifier with \ breaks rules

\a\*b = 0; // but be careful to watch the spaces!

\$display("Variable CASE\_SENSITIVE= %d",CASE\_SENSITIVE);

\$display("Variable case\_sensitive= %d",case\_sensitive);

\$display("Variable Vclock = %d",Vclock );

\$display("Variable \\a\*b = %d",\a\*b );

**end**

**endmodule**

**An Example**  
44

## Simulation Result of the Example

Variable CASE\_SENSITIVE= 0

Variable case\_sensitive= 1

Variable /clock = 0

Variable \a\*b = 0

# Logic values

- Verilog has 4 logic Values:
  - '0' represents zero, low, false, not asserted.
  - '1' represents one, high, true, asserted.
  - 'z' or 'Z' represent a high-impedance value, which is usually treated as an 'x' value.
  - 'x' or 'X' represent an uninitialized or an unknown logic value--an unknown value is either '1' , '0' , 'z' , or a value that is in a state of change.

# Data Types

- Three data type classes:
  - Nets
    - Physical connections between devices
    - Example: **wire** a, b;
  - Registers
    - Storage devices, variables.
    - Example: **reg** a; **reg** [7:0] bus;
  - Parameters
    - Constants
    - Example: **parameter** width=32;  
**parameter** A\_string =“hello”;

## **Code Structure**

---

**Design Entities**

**Verilog Module Basics**

# Design Entities

- The **module** is the basic unit of code in the Verilog language.

- Example

```
module holiday_1(sat, sun, weekend);  
    input sat, sun;  
    output weekend;  
    assign weekend = sat | sun;  
endmodule
```

# Verilog Module

- **Modules contain**
  - **declarations**
  - **functionality**
  - **timing**

```
module name (port_names);  
  
    module port declarations  
  
    data type declarations  
  
    procedural blocks  
  
    continuous assignments  
  
    user defined tasks & functions  
  
    primitive instances  
  
    module instances  
  
    specify blocks  
  
endmodule
```

syntax:

```
module module_name (signal, signal,... signal ) ;  
    ; //content of module  
    .  
    .  
    ..  
    .  
endmodule
```

# Module Port Declarations

- Scalar (1bit) port declarations:
  - *port\_direction port\_name, port\_name ... ;*
- Vector (Multiple bit) port declarations:
  - *port\_direction [port\_size] port\_name, port\_name ... ;*
- *port\_direction* : input, inout (bi-directional) or output
- *port\_name* : legal identifier
- *port\_size* : is a range from [msb:lsb]

```
input a, into_here, george; // scalar ports
input [7:0] in_bus, data;    // vectored ports
output [31:0] out_bus;      // vectored port
inout [maxsize-1:0] a_bus; // parameterized port
```

# Module Instances

- A module may be instantiated within another module.
- There may be multiple instances of the same module.

syntax for instantiation:

***module\_name instance\_name*** (signal, signal,...);

```
module example (a,b,c,d);  
input a,b;  
output c,d;  
.  
.  
endmodule
```

```
example ex_inst_1(in_1, in_2, w, z);  
example ex_inst_2(in_1, in_2, , z); // skip a port
```

# Gate-level Primitives

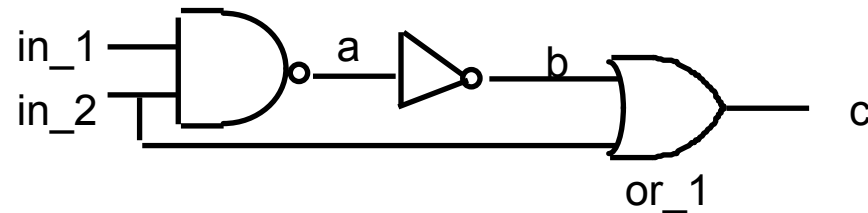
- Verilog has pre-defined primitives that implement basic logic functions.
- Structural modeling with the primitives is similar to schematic level design.

<b>and</b>	<b>nand</b>	<b>or</b>	<b>nor</b>	<b>xor</b>	<b>xnor</b>
<b>buf</b>	<b>not</b>	<b>bufif0</b>	<b>bufif1</b>	<b>notif0</b>	<b>notif1</b>

```
module
gate_level_ex(in_1,in_2,c);
output c;
input in_1,in_2;

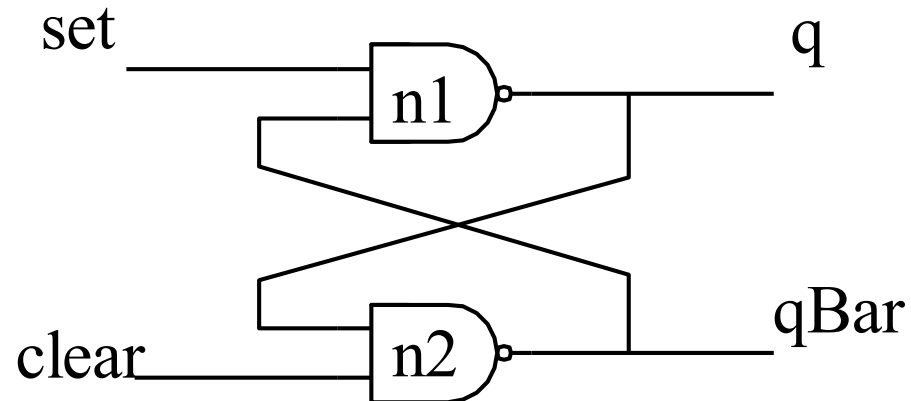
nand (a, in_1, in_2);
not (b, a);
or or_1(c, in_2, b);

endmodule
```



# Activity 4

Given the circuit below, develop a Verilog module for the circuit



# User-Defined Primitives

- We can define primitive gates (a **user-defined primitive** or **UDP**) using a truth-table specification. The first port of a UDP must be an output port, and this must be the only output port (we may not use vector or inout ports).

- An example

```
primitive Adder(Sum, InA, InB);  
    output Sum;  
    input InA, InB;  
    table // inputs : output  
    00 : 0;  
    01 : 1;  
    10 : 1;  
    11 : 0;  
    endtable  
endprimitive
```

# Operators

## ■ Verilog operators (in increasing order of precedence)

- ?: (conditional)
- || (logical or)
- && (logical and)
- | (bitwise or)
- ~| (bitwise nor)
- ^ (bitwise xor)
- ^~ ~^ (bitwise xnor, equivalence)
- & (bitwise and)
- ~& (bitwise nand)
- == (logical) != (logical) === (case) !== (case)
- < (lt)
- <= (lt or equal)
- > (gt)
- >= (gt or equal)
- << (shift left)
- >> (shift right)
- + (addition)
- - (subtraction)
- \* (multiply)
- / (divide)
- % (modulus)

# CSE 2021 Computer Organization

**Procedural Assignment**  
**Continuous Assignment**  
**Control Statement**

# Procedures

- A Verilog **procedure** is an **always** or **initial** statement, a task , or a function .
- The statements within a sequential block (statements that appear between a **begin** and an **end** ) that is part of a procedure execute sequentially in the order in which they appear, but the procedure executes concurrently with other procedures.

# Procedural Blocks

- There are two types of procedural blocks:
  - initial blocks - executes only once
  - always blocks - executes in a loop
- Multiple Procedural blocks may be used, if so the multiple blocks are concurrent.
- Procedural blocks may have:
  - Timing controls - which delays when a statement may be executed
  - Procedural assignments
  - Programming statements

# Procedural Statement Groups

- When there is more than one statement within a procedural block the statements must be grouped.
- Sequential grouping: statements are enclosed within the keywords **begin** and **end**.
- An example

always

begin

a = 5;           // executed 1st

c = 4;           // executed 2nd

wake\_up = 1; // executed 3rd

end

# Timing Controls (procedural delays)

- **#delay** - simple delay
  - Delays execution for a specific number of time steps.  
**#5** reg\_a = reg\_b;
- **@ (edge signal)** - edge-triggered timing control
  - Delays execution until a transition on **signal** occurs.
  - **edge** is optional and can be specified as either **posedge** or **negedge**.
  - Several **signal** arguments can be specified using the keyword **or**.
  - An example : always @ (posedge clk) reg\_a = reg\_b;
- **wait (expression)** - level-sensitive timing control
  - Delays execution until **expression** evaluates true.
  - **wait (cond\_is\_true)** reg\_a = reg\_b;

# Procedural assignments

- Assignments made within procedural blocks are called procedural assignments.
  - Value of the RHS of the equal sign is transferred to the LHS
  - LHS must be a register data type (reg, integer, real). NO NETS!
  - RHS may be any valid expression or signal

```
always @ (posedge clk)
begin
    a = 5;           // procedural assignment
    c = 4*32/6;      // procedural assignment
    wake_up = $time; // procedural assignment
end
```

# Continuous Assignment

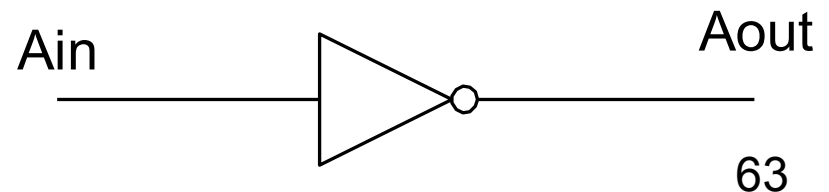
- Continuous assignment assigns a value to a **wire** in a similar way that a real logic gate drives a real wire.
- The main use for continuous assignments is to model combinatorial logic.

**syntax:** Explicit continuous assignment:

***assign net\_name = expression;***

where ***net\_name*** is a **net** that has been previously declared

```
module continuous (Ain, Aout);  
    input Ain;  
    output Aout;  
    assign Aout = ~Ain //continuous assignment.  
endmodule
```



## Illustration of Assignment Statements

**module** assignments

//... Continuous assignments go here.

**always** // beginning of a procedure

**begin** // beginning of sequential block

//... Procedural assignments go here.

**end**

**endmodule**

# Control Statements

- Two types of programming statements:
  - Conditional
  - Looping
- Programming statements only used in procedural blocks

# if and if-else

**syntax:**

***if(expression) statement***

If the expression evaluates to true then execute the statement

***if(expression) statement1  
else statement2***

If the expression evaluates to true then execute statement1,  
if false, then execute statement2.

```
module if_ex(clk);  
    input clk;  
    reg red,blue,pink,yellow,orange,color,green;  
    always @ (posedge clk)  
        if (red || (blue && pink))  
            begin  
                $display ("color is mixed up");  
                color <= 0; // reset the color  
            end  
        else if (blue && yellow)  
            $display ("color is greenish");  
        else if (yellow && (green || orange))  
            $display ("not sure what color is");  
        else $display ("color is black");  
endmodule
```

# for

**syntax:**

***for (assignment\_init; expression; assignment)  
statement or statement\_group***

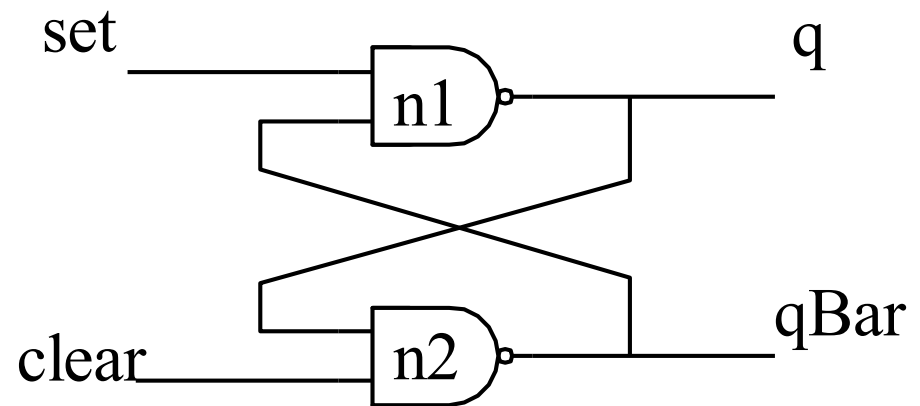
- The ***assignment\_init*** is executed once at the start of the loop.
- Loop executes as long as ***expression*** is true.
- The ***assignment*** is executed at the completion of each loop.

```
module for_ex1 (clk);  
input clk;  
reg [31:0] mem [0:9]; // 10x32 memory  
integer i;  
always @ (posedge clk)  
    for (i = 9; i >= 0; i = i-1)  
        mem[i] = 0; // init the memory to zeros  
endmodule
```

# Simulating the Verilog Code

- Verilog code of NAND Latch

```
Module simple_latch (q, qBar, set, clear);  
  input set, clear;  
  output q, qBar;  
  nand #2 n1(q,qBar,set);  
  nand #2 n2(qBar,q,clear);  
endmodule
```



# Testbench

- A testbench generates a sequence of input values (we call these **input vectors** ) that test or **exercise** the verilog code.
- It provides stimulus to the statement that will monitor the changes in their outputs.
- Testbenches do not have a port declaration but must have an instantiation of the circuit to be tested.

# A testbench for NAND Latch

```
Module test_simple_latch;
  wire q, qBar;
  reg set, clear;
  simple_latch SL1(q,qBar,set,clear);
  initial
    begin
      #10 set = 0; clear = 1;
      #10 set = 1;
      #10 clear = 0;
      #10 clear = 1;
      #10 $stop;
      #10 $finish;
    end
  initial
    begin
      $monitor ("%d set= %b clear= %b q=%b qBar=%b", $time,
                set, clear, q, qBar);
    end
endmodule
```