CHW 362 : Computer Architecture & Organization

Instructors:
Dr Ahmed Shalaby http://bu.edu.eg/staff/ahmedshalaby14#

bu.edu.eg/staff/ahmedshalaby14#

Benha University

Staff Search: Go

Login

- Benha University
- Home
- النسخة العربية
- My C.V.
- About
- Courses
- Publications
- Reports
- Published books
- Workshops / Conferences
- Supervised PhD
- Supervised MSc
- Supervised Projects
- Education
- Language skills
- Academic Positions
- Administrative Positions
- Memberships and awards

You are in:Home

Dr. Ahmed Shalaby

Academic Position: Lecturer

Current Administrative Position:

Ex-Administrative Position:

Faculty: Computers and Informatics

Department: Computer Science

Edu-Mail: ahmed.shalaby@fci.bu.edu.eg

Alternative Email: a.shalaby@ieee.org

Mobile:

Scientific Name:

Publications [Titles(4) :: Papers(2) :: Abstracts(4) **]**

Inlinks(6) :: Courses Files(4) | **Total points :49**

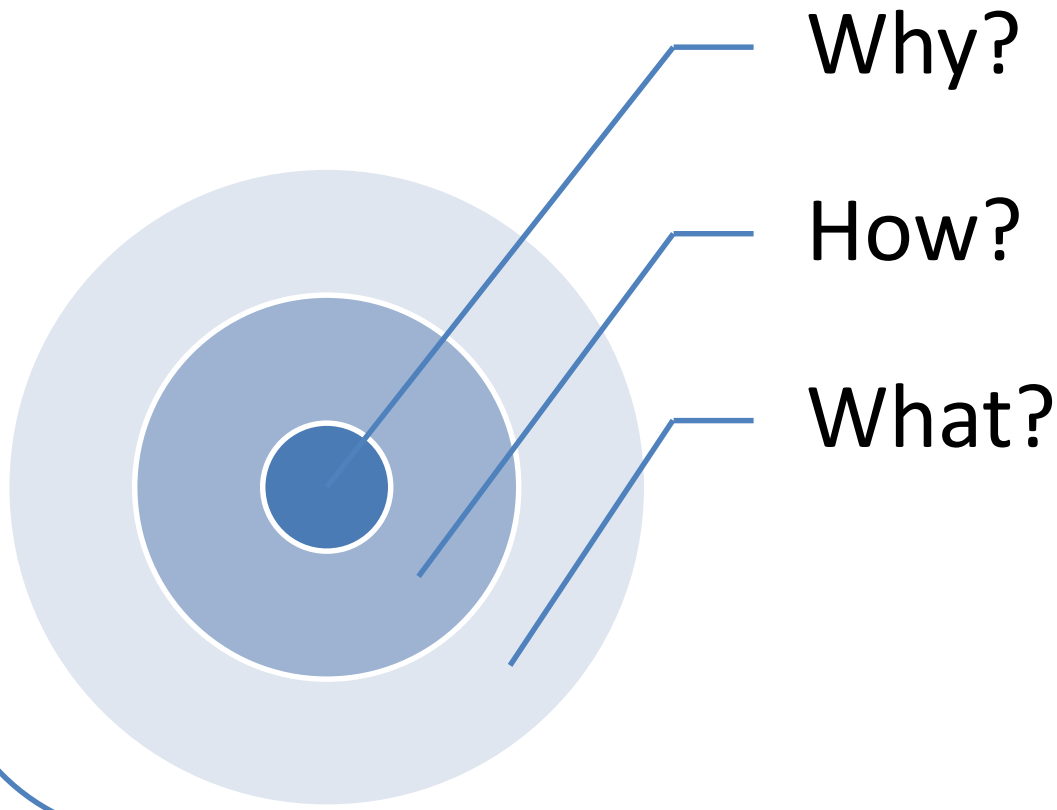
News

IOT Project Update [2016-11-14]
Smart Water Management Irrigation System (SWMIS) project moves to field test phase on November 2016. more

Research Interests

System on Chip, Network on Chip, VLSI, Embedded System, High Efficiency Video Coding (HEVC)

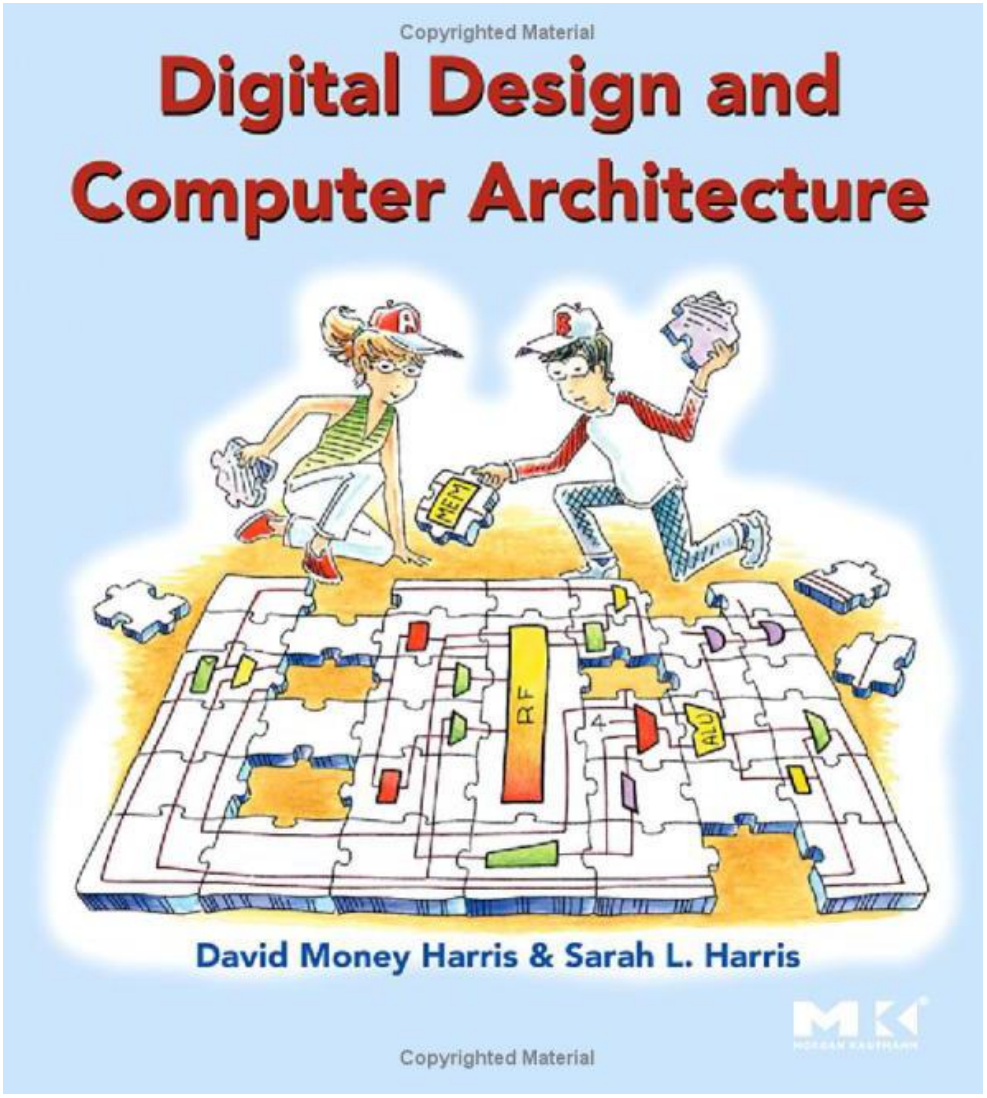
Study: CHW 362 : Computer Architecture & Organization



What ? Computer Architecture

- **computer architecture** defines how to command a processor.
- **computer architecture** is a set of rules and methods that describe the functionality, organization, and implementation of computer system.

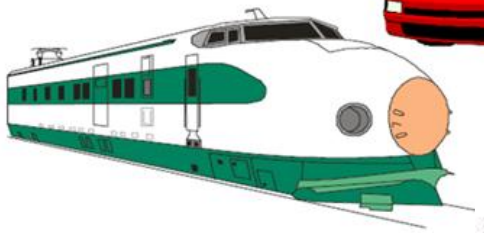
How ? Course Book



How ? Course Content

Lec #	Subject	Week #
Lec1	Chapter 1: From Zero to One	Week #1
Lec2	Chapter 2: Combinational Logic Design	Week #2
Lec 3	Chapter 3: Sequential Logic Design	Week #3
Lec 4	Chapter 5 : Digital Building Blocks	Week #4
Lec 5	Chapter 5: Digital Building Blocks Cont …	Week #5
Lec 6	Chapter 4 : Hardware Description Language	Week #6
Lec 7	Hardware Description Language Cont …	Week #7
	MidTerm	Week #8
Lec 8	Chapter 6: Computer Architecture	Week #9
Lec 9	Chapter 6: Computer Architecture Cont …	Week #10
Lec 10	Chapter 7 : Microarchitecture	Week #11
Lec 11	Chapter 7: Microarchitecture Cont …	Week #12
Lec 12	Chapter 6, 7 revision	Week #13

Why ? Computer Architecture



Digital System Implementation Spectrum

Hardware
ASIC

Reconfigurable Architectures

- CPLD
- FPGA
- Customized Processors
- Coarse Grain
- Reconfigurable Array

Software

- µProcessor
- µController
- DSP

flexibility trade-off efficiency

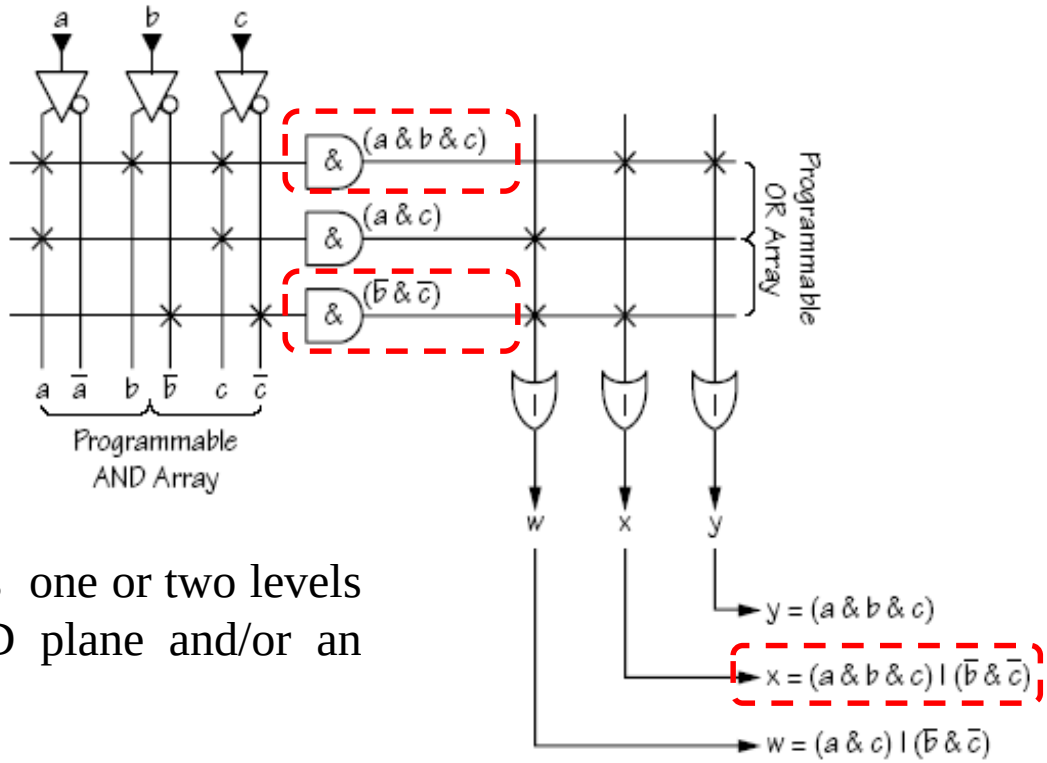
Reconfigurable Architectures

- Filed Programmable Devices
  - ❑ Simple
    - **Programmable logic Devices (PLD)**
  - ❑ Complex
    - **Complex Programmable Logic Devices (CPLD)**
    - **Field Programmable Gate Array (FPGA)**

Programmable logic Devices

Logic Functions

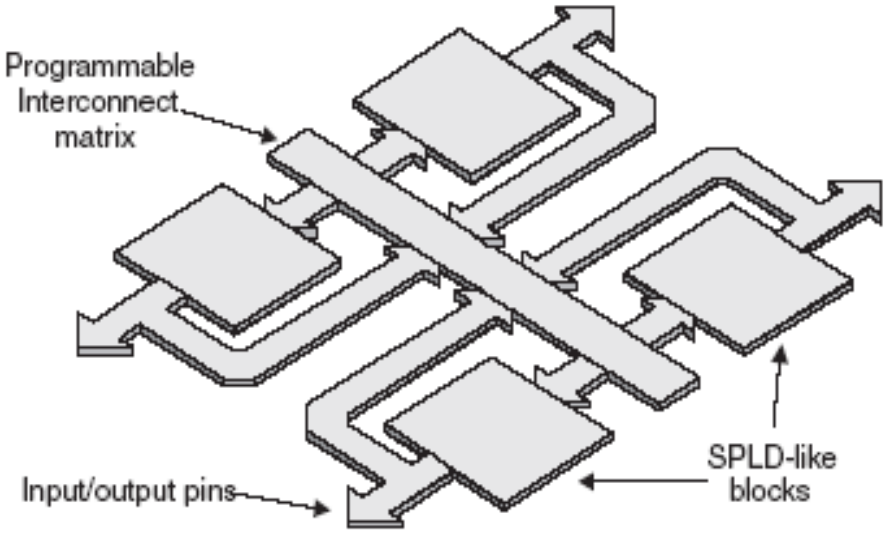
- $y = (a \,\&\, b \,\&\, c)$
- $x = (a \,\&\, b \,\&\, c) \mid (\overline{b} \,\&\, \overline{c})$
- $w = (a \,\&\, c) \mid (\overline{b} \,\&\, \overline{c})$

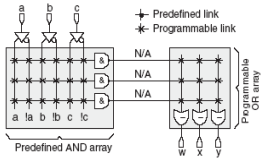


Relatively small FPD that contains one or two levels of programmable logic—an AND plane and/or an OR plane.

Complex PLD

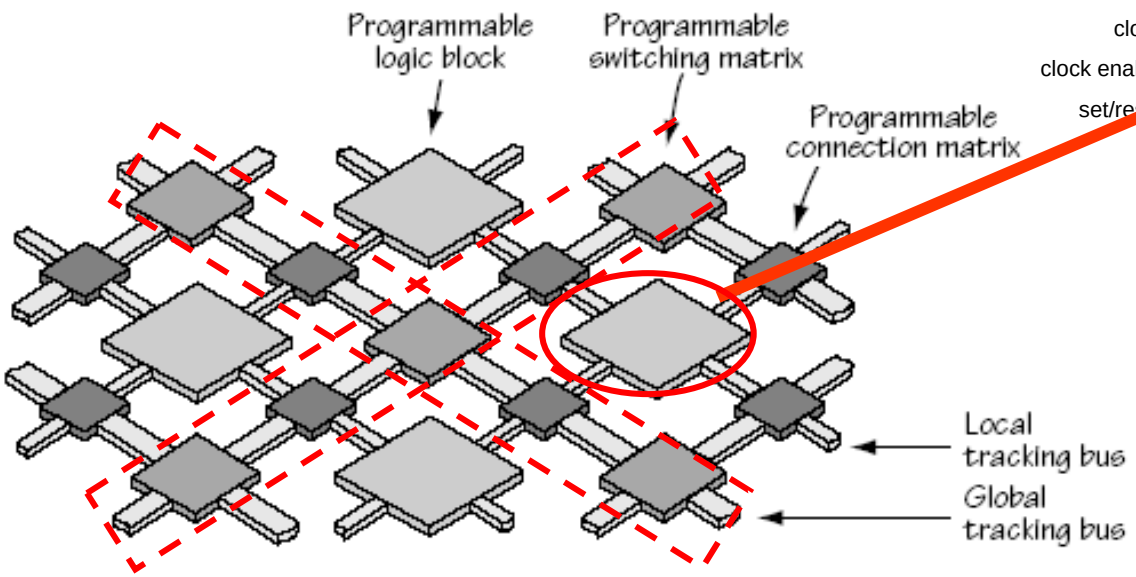
- Arrangement of multiple SPLD-like blocks on a single chip.
- Programmable PLD Blocks, Programmable Interconnects

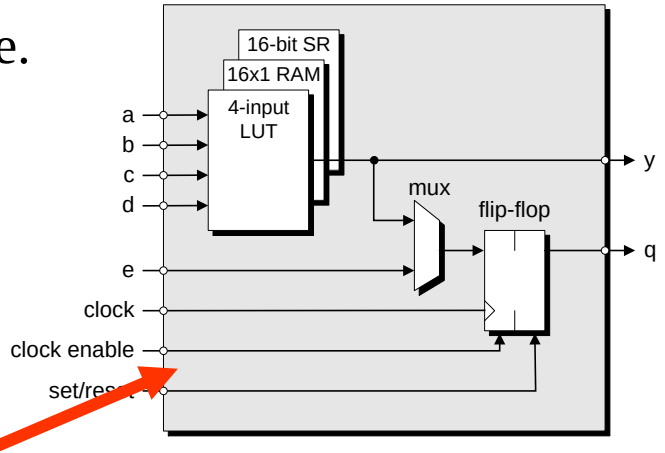




FPGA

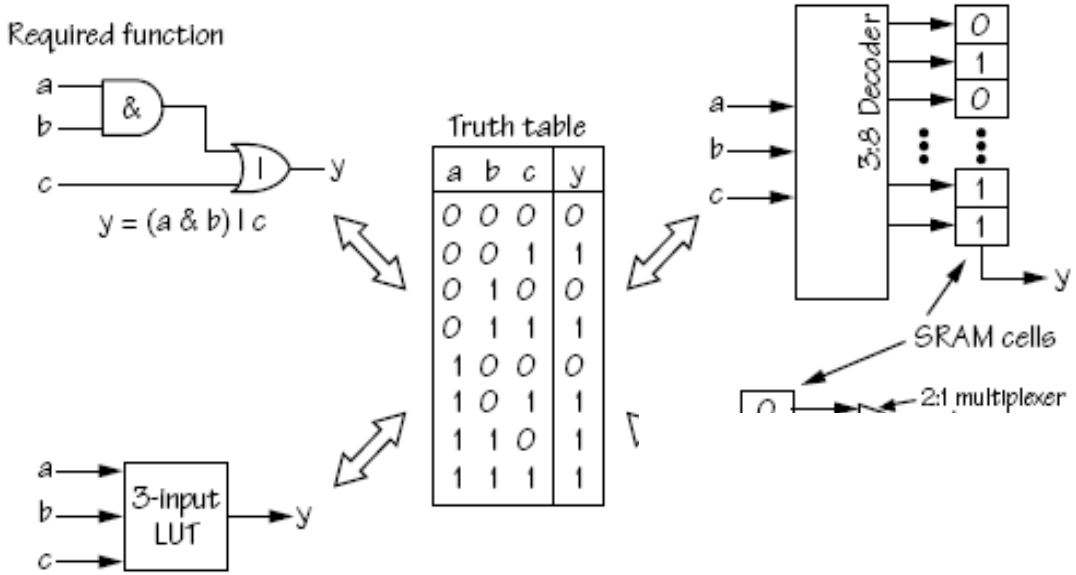
- Logic Functions implemented in Look Up Table.
- Flip-Flops (Register).
- Multiplexers





FPGA - LUT

- LUT contains Memory Cells to implement small logic functions
- Each cell holds ‘0’ or ‘1’ .
- Programmed with outputs of Truth Table
- Inputs select content of one of the cells as output
- Configured by re-programmable SRAM memory cells



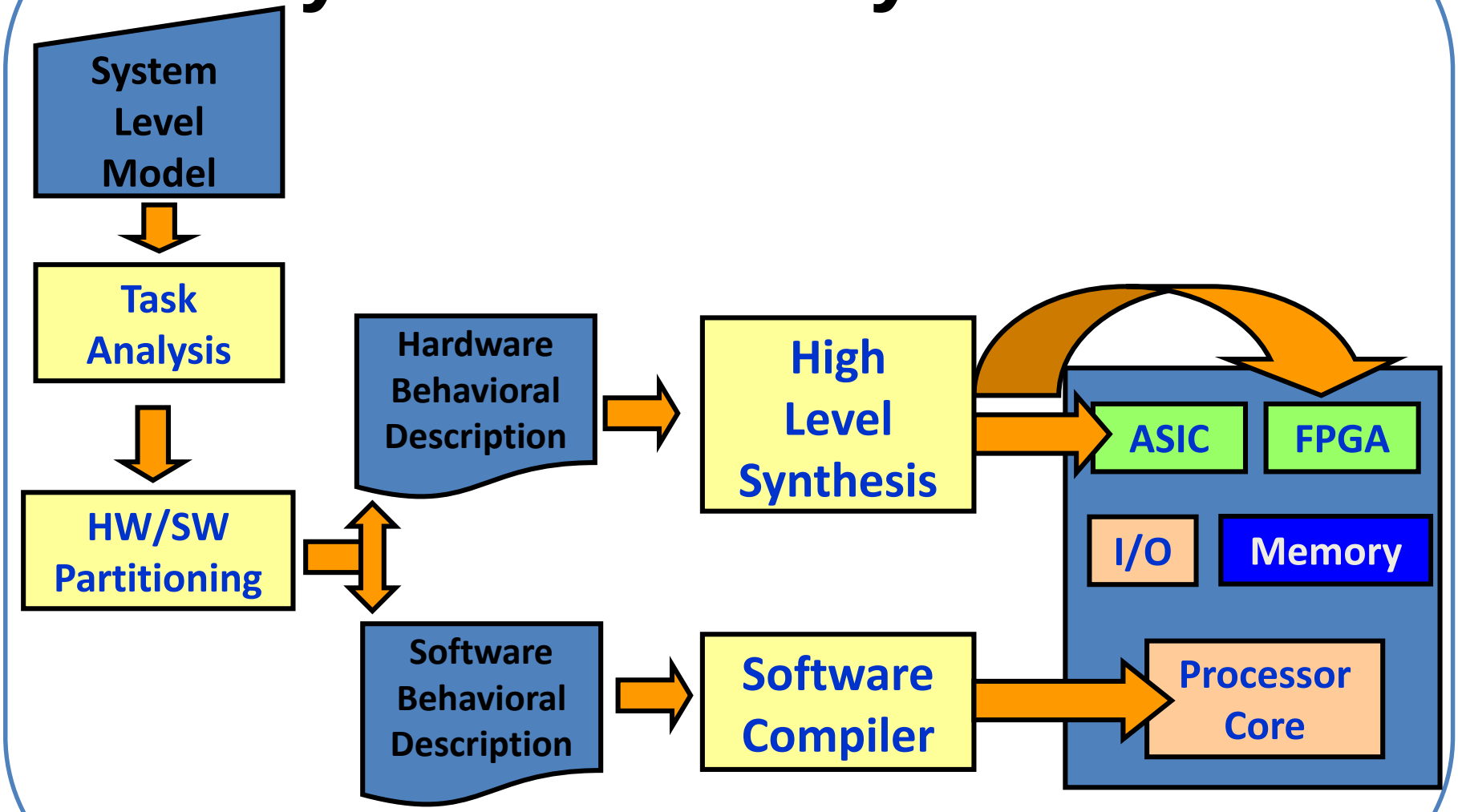
FPGA – Design Flow

- Reading Specs and Define the full definition of the problem
- Detailed Specs and architecture of the project.
- Behavioral design
  - High level Description of Logic Design.
  - Schematic.
  - State machine.
  - Flow chart.
  - Block diagram.
- Function Verification (Pre-Synthesis Simulation).
- Synthesis - Target FPGA Device.
  - Place
  - Route
- Timing Verification (Post-Synthesis Simulation).
- Device Programing.

Synthesis

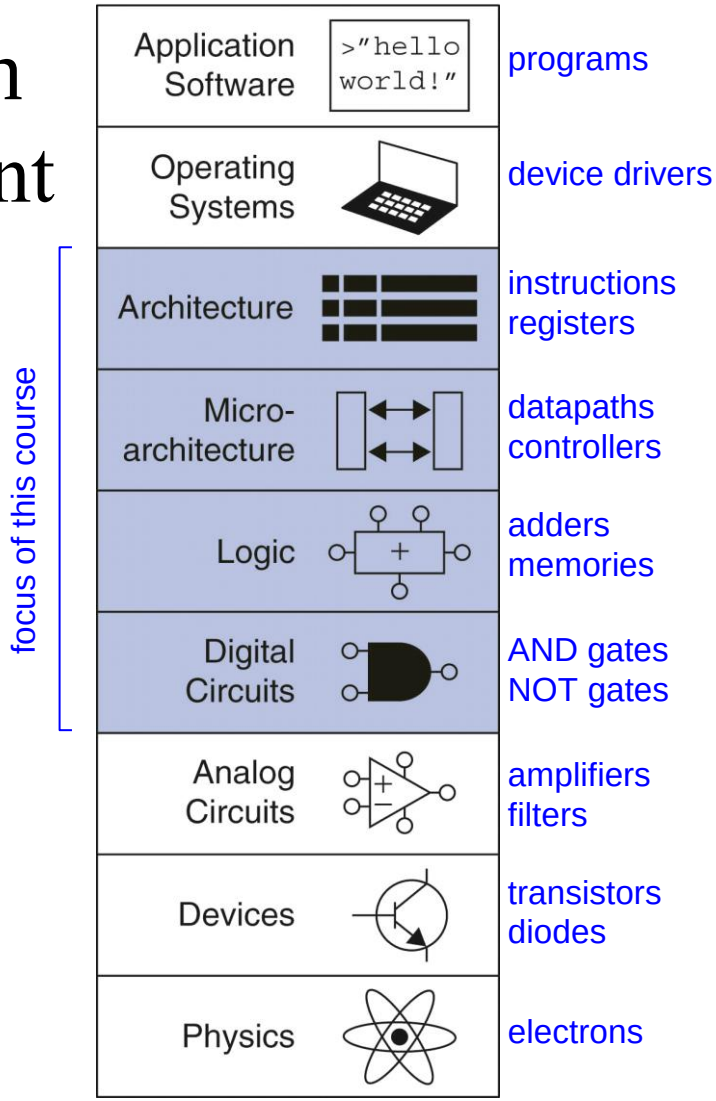
- Synthesis is an automatic method of converting a higher level of abstraction (RTL) to a lower level of abstraction (gate level Netlists).
- Synthesis produces technology-specific implementation from technology-independent HDL description.
- Not all HDL can be used for synthesis. There are the HDL subset for synthesis and synthesis style description.
- Synthesis is very sensitive to how the HDL is written
  - Good design is still the responsibility of the designer.
  - Junk in - junk out

System Level Synthesis

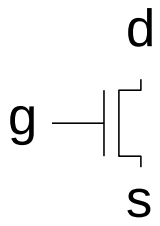
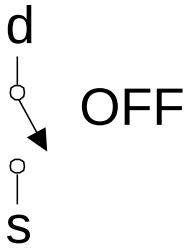
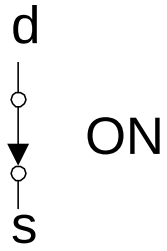
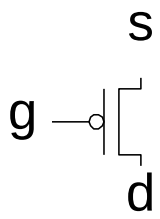
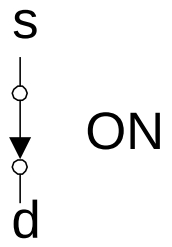
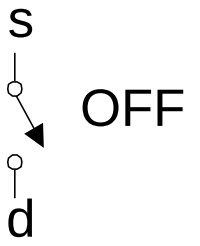


Abstraction

- Hiding details when they aren't important



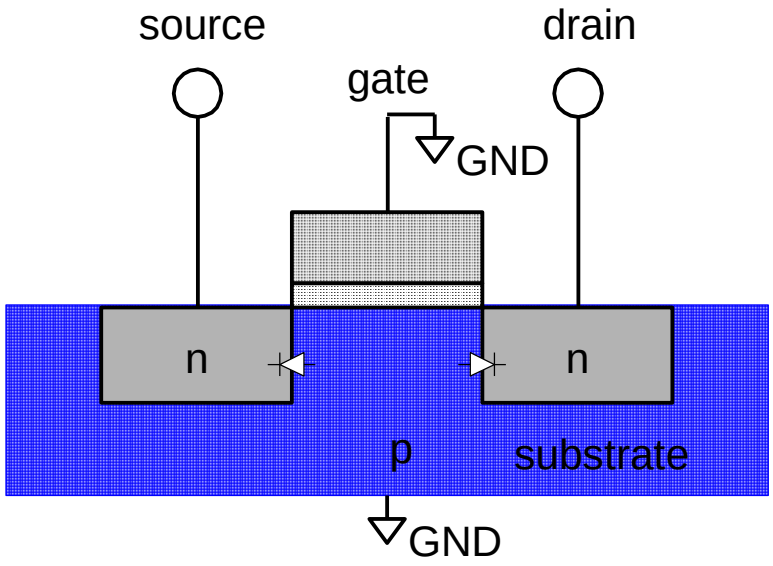
Transistor Function

		g = 0	g = 1
nMOS			
pMOS			

Transistors: nMOS

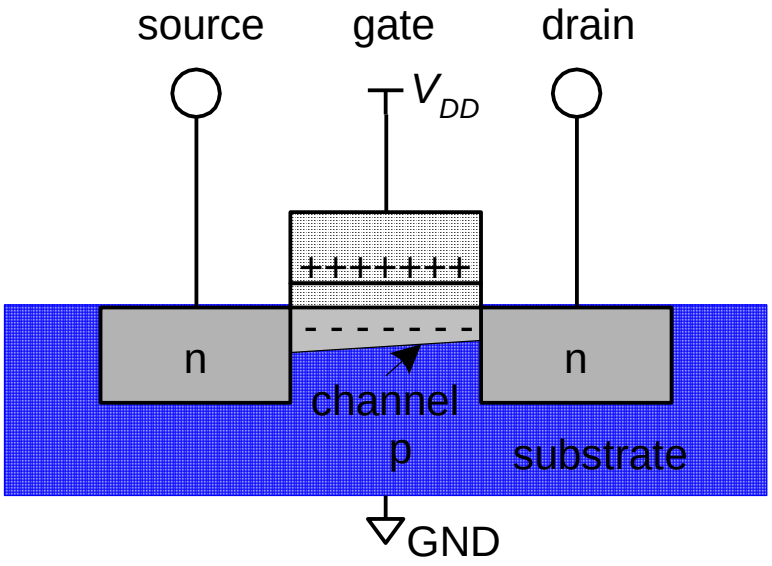
Gate = 0

OFF (no connection between source and drain)



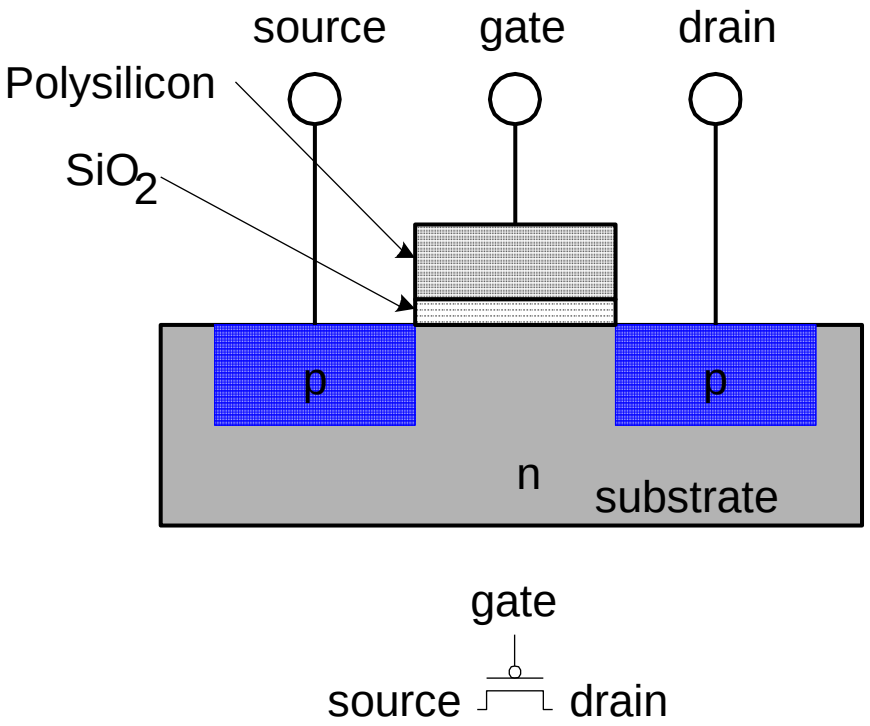
Gate = 1

ON (channel between source and drain)



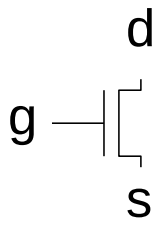
Transistors: pMOS

- pMOS transistor is opposite
  - ON when Gate = 0
  - OFF when Gate = 1

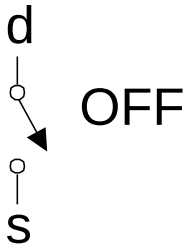


Transistor Function

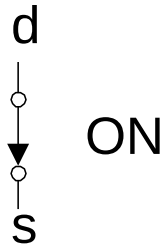
nMOS



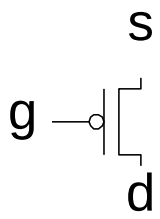
g = 0

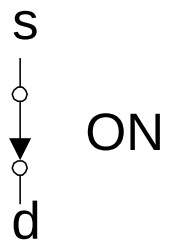


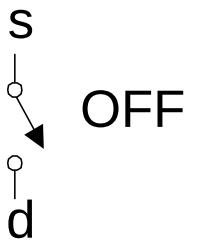
g = 1



pMOS

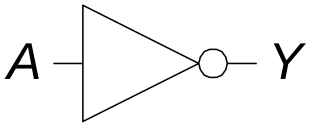






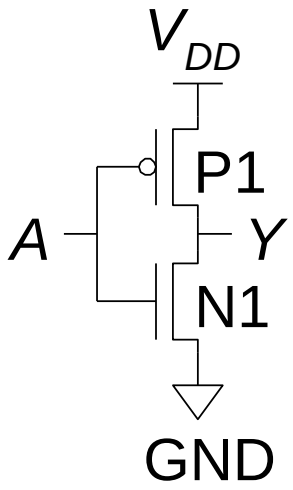
CMOS Gates: NOT Gate

NOT



$Y = \overline{A}$

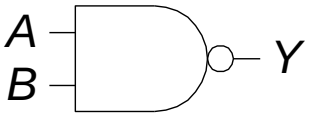
A	Y
0	1
1	0



A	P1	N1	Y
0	ON	OFF	1
1	OFF	ON	0

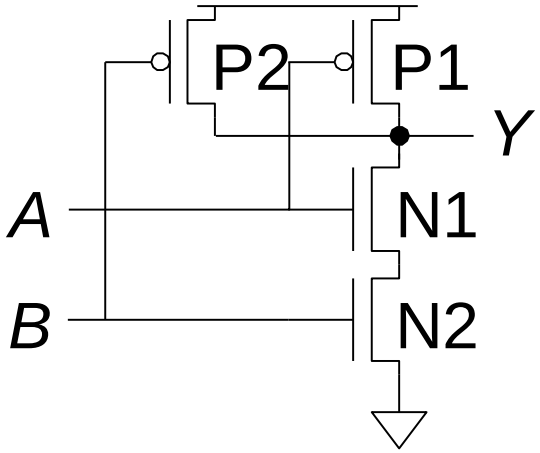
CMOS Gates: NAND Gate

NAND



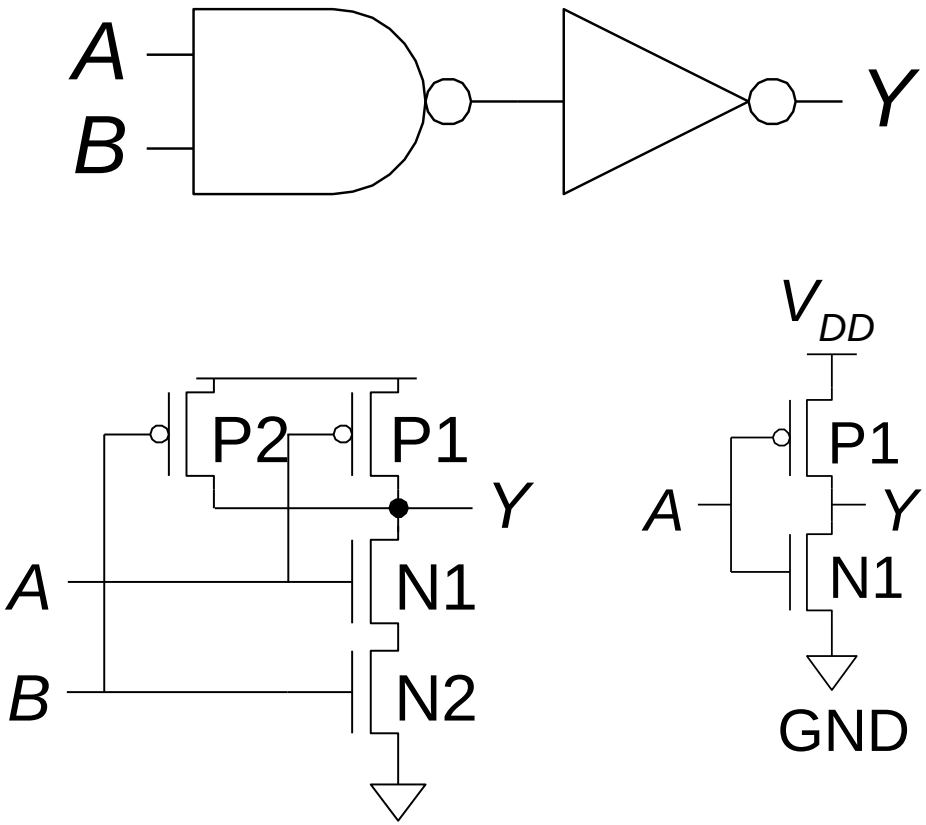
$Y = \overline{AB}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0



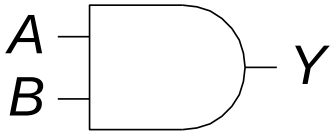
A	B	P1	P2	N1	N2	Y
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	1
1	0	OFF	ON	ON	OFF	1
1	1	OFF	OFF	ON	ON	0

AND Gate



Two-Input Logic Gates

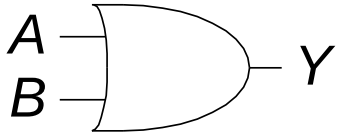
AND



$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR

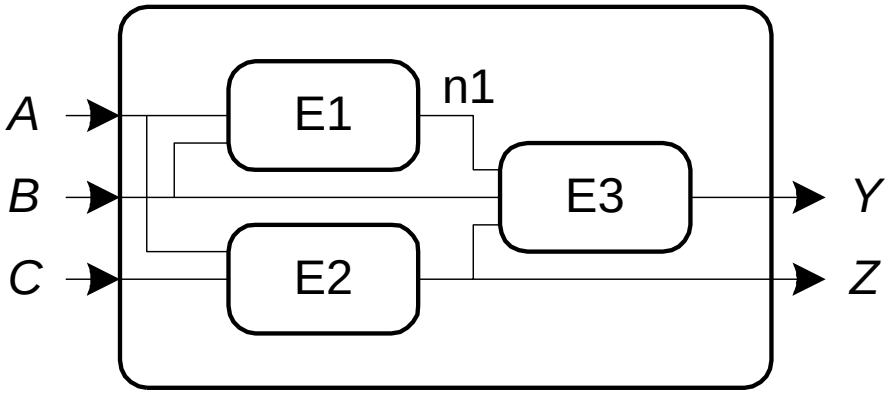


$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

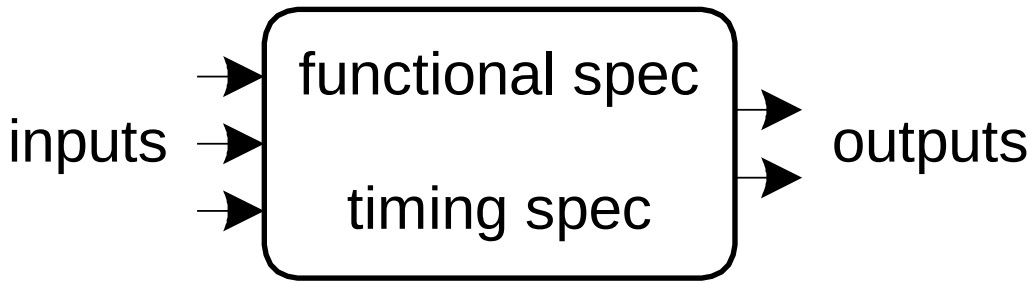
Circuits

- Nodes
  - Inputs: *A*, *B*, *C*
  - Outputs: *Y*, *Z*
  - Internal: n1
- Circuit elements
  - E1, E2, E3
  - Each a circuit



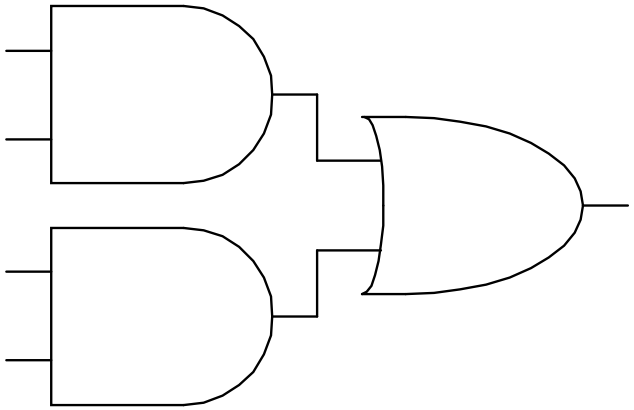
Types of Logic Circuits

- **Combinational Logic**
  - Memoryless
  - Outputs determined by current values of inputs
- **Sequential Logic**
  - Has memory
  - Outputs determined by previous and current values of inputs

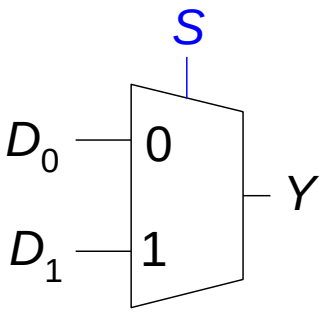


Combinational Composition

- Every element is combinational
- Every node is either an input or connects to *exactly one* output
- The circuit contains no cyclic paths
- **Example:**

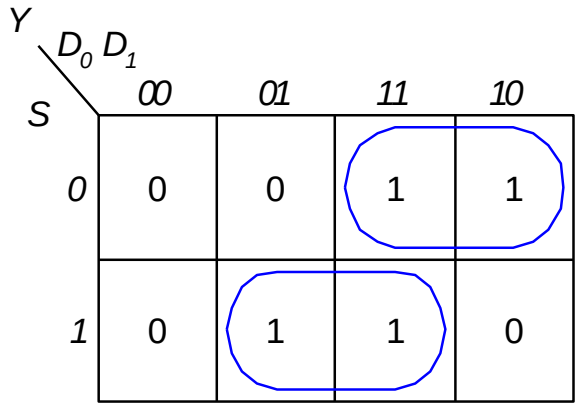


Multiplexer

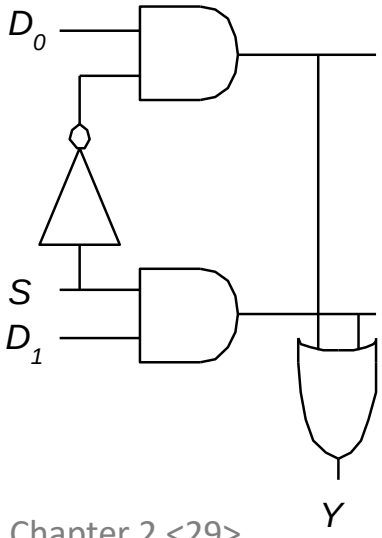


S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

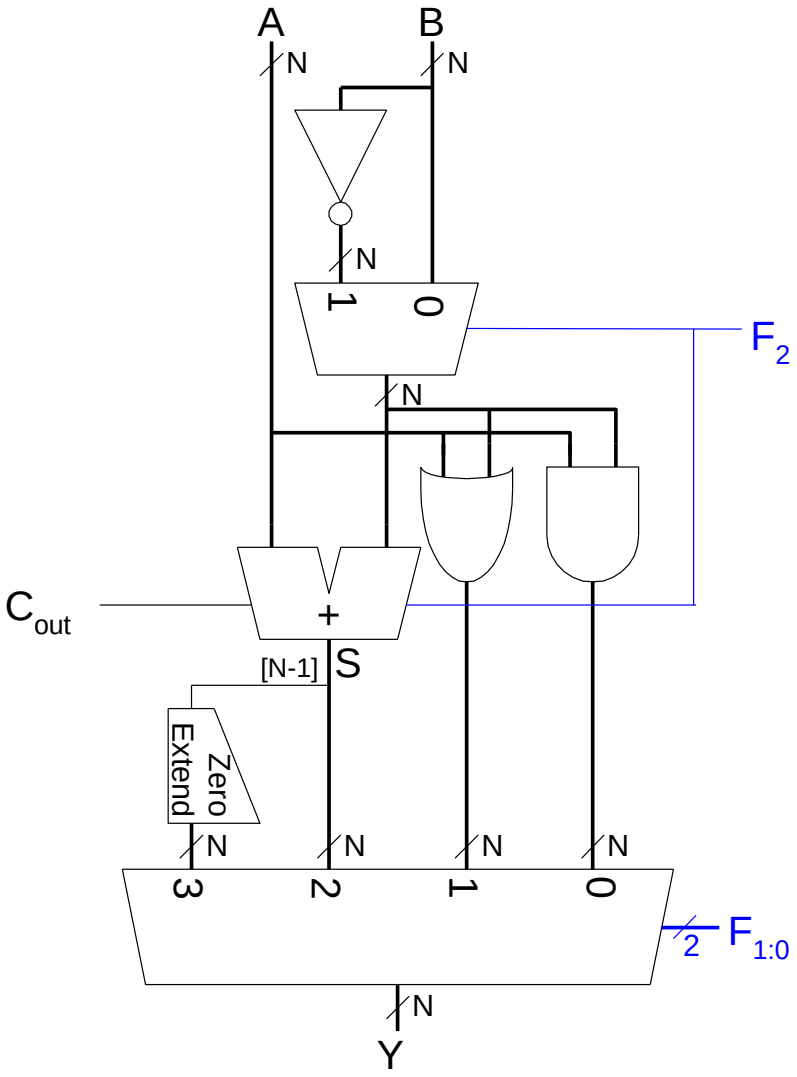
S	Y
0	D_0
1	D_1



$$Y = D_0\overline{S} + D_1S$$



ALU Design



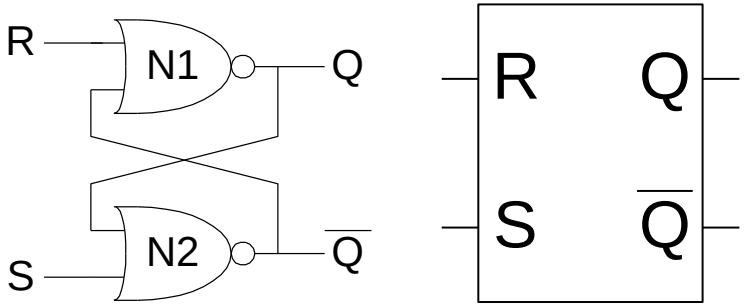
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Copyright © 2007 Elsevier

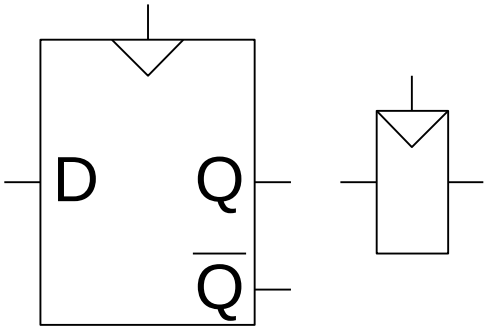
Sequential Circuits

- Give sequence to events
- Have memory (short-term)
- Use feedback from output to input to store information

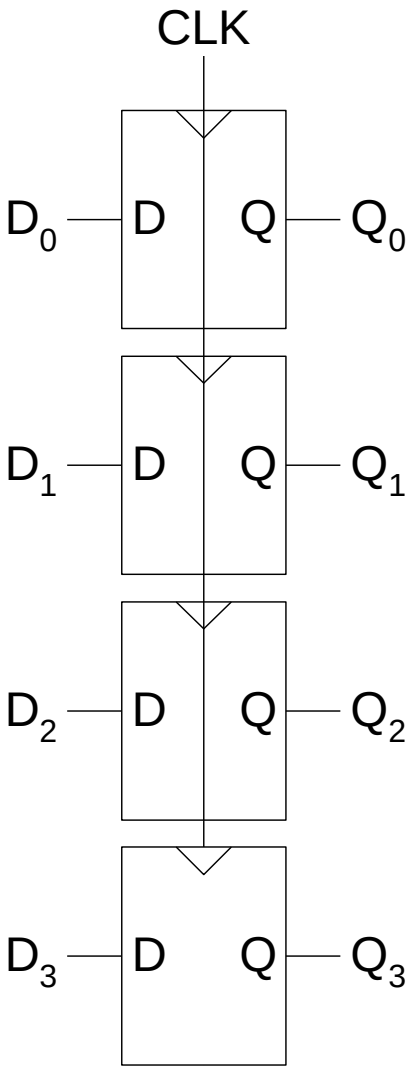
SR Latch Symbol

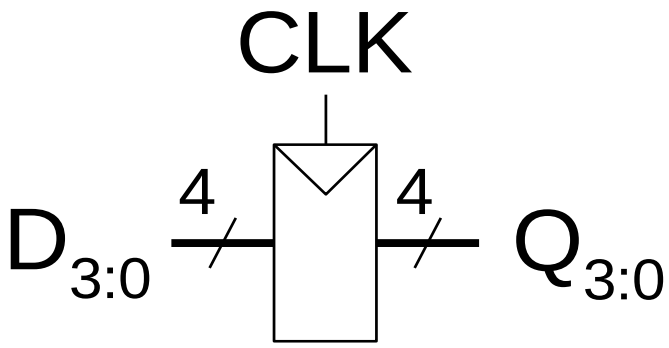


D Flip-Flop Symbols



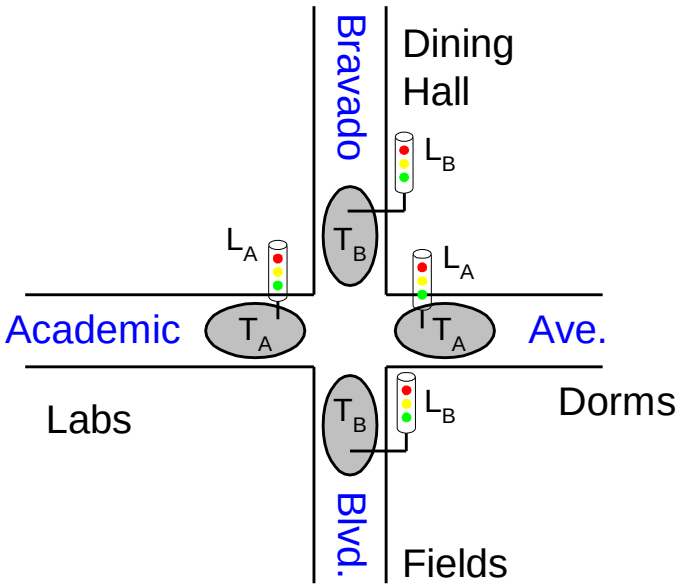
Registers

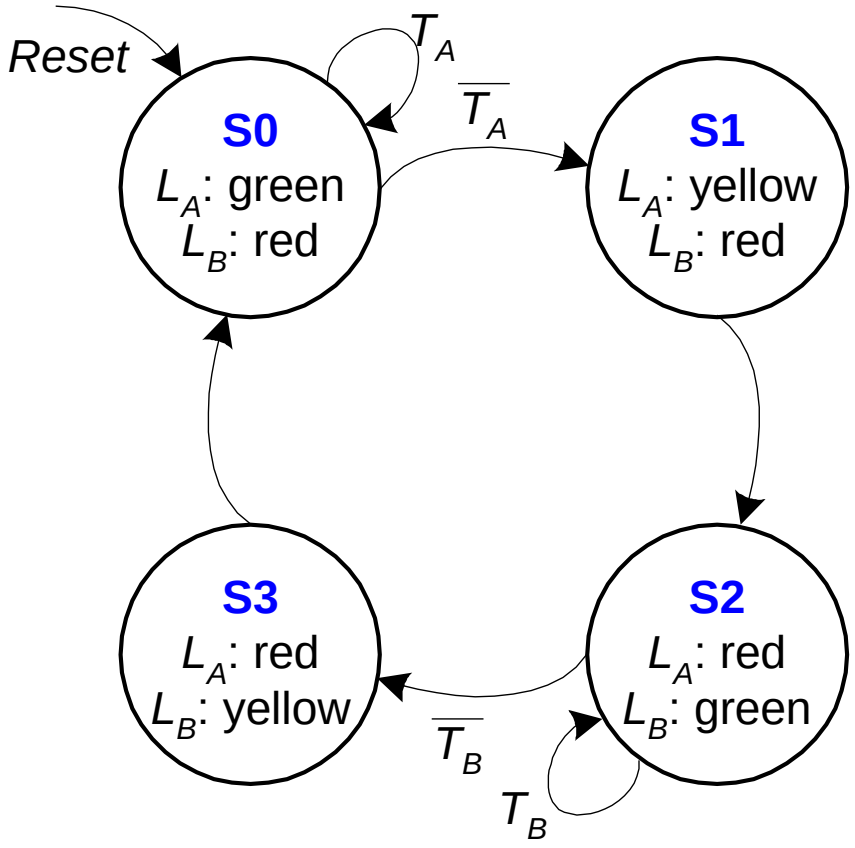




FSM State Transition

- **Moore FSM:** outputs labeled in each state
- **States:** Circles
- **Transitions:** Arcs

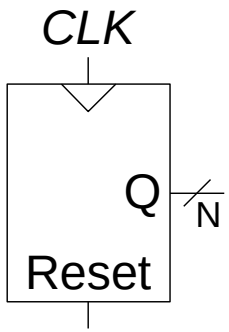




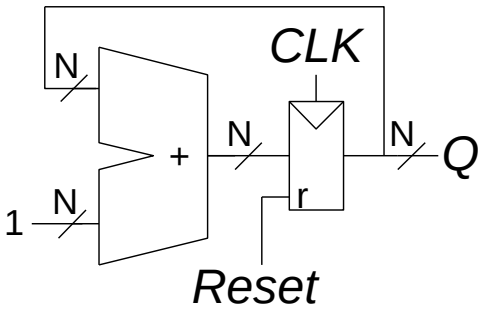
Counters

- Increments on each clock edge
- Used to cycle through numbers. For example,
  - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001…
- Example uses:
  - Digital clock displays
  - Program counter: keeps track of current instruction executing

Symbol



Implementation



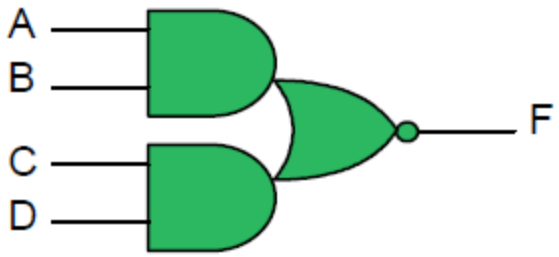
Verilog Overview

Verilog Overview

- To become familiar with the basic features of the Verilog Hardware Description Language.

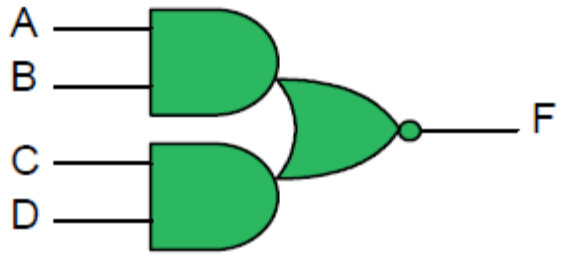
- Topics:
  - Modules.
  - Ports.
  - Continuous assignments.
  - Hierarchy.
  - Logic Values.
  - Test fixtures.
  - Initial blocks.

Modules and Ports




```
// An and-or-invert gate

module AOI (A, B, C, D, F);
  input A, B, C, D;
  output F;

// The function of the AOI module is described here...

endmodule
```

Continuous Assignment




```
// An and-or-invert gate

module AOI (A, B, C, D, F);
  input A, B, C, D;
  output F;

  assign F = ~((A & B) | (C & D));

endmodule
```

&	and
|	or
~	not

Rules and Regulations

All definitions and statements go inside a module

```
// An and-or-invert gate
module AOI (A, B, C, D, F);
  input A, B, C, D;
  output F;
  assign F = ~((A & B) | (C & D));
/*
These lines are ignored
by the compiler
*/
endmodule
```

A comment

Case sensitive names

; at end of definition/statement

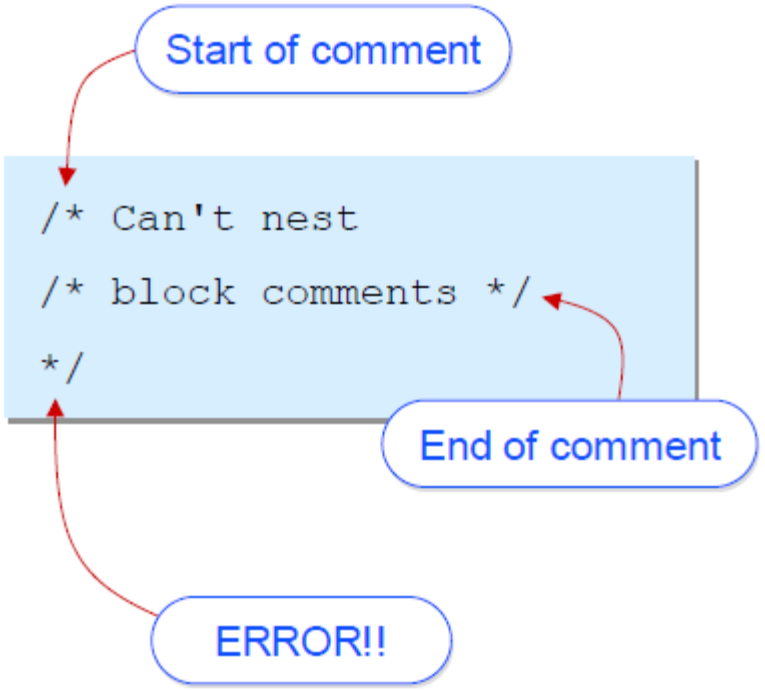
A block comment

Lower case keywords

Single-line vs. Block Comments

```
// Comment out
// a number of lines
// using single-line
// comments
```

```
/* Can nest
// single-line comments
// within block comments
*/
```



Names

- **Identifiers**

```
AB      Ab      aB      ab      // different!
G4X$6_45_123$
Y       Y_      _Y              //different!
```

- **Illegal!**

```
4plus4          $1
```

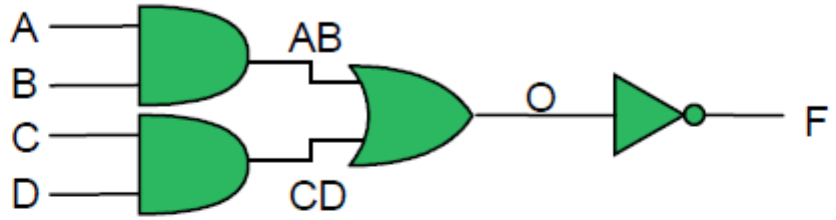
- **Escaped identifiers (terminate with white space)**

```
\4plus4         \$1     \a+b£$%^&*(
```

- **Keywords**

```
and   default   event   function   wire
```

Wires

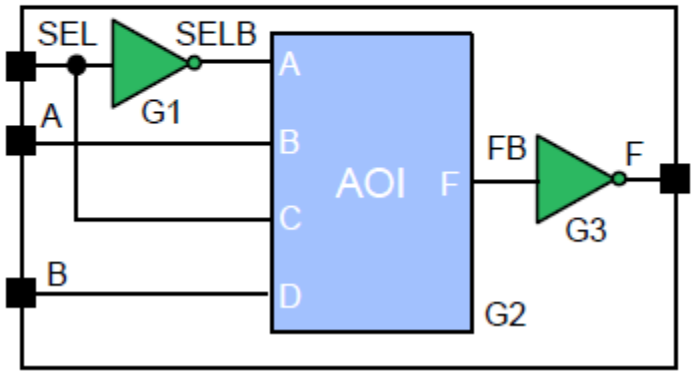



```
module AOI (A, B, C, D, F);
  input A, B, C, D;
  output F;

  wire F;                 // The default
  wire AB, CD, O;         // Necessary

  assign AB = A & B;
  assign CD = C & D;
  assign O = AB | CD;
  assign F = ~O;

endmodule
```

Target (LHS) of *assign* must be a *wire*

Hierarchy

```
module AOI (A, B, C, D, F);
   ...
endmodule

module INV (A, F);
   ...
endmodule

module MUX2 (SEL, A, B, F);
   input SEL, A, B;
   output F;
   INV G1 (SEL, SELB);
   AOI G2 (SELB, A, SEL, B, FB);
   INV G3 (FB, F);
endmodule
```



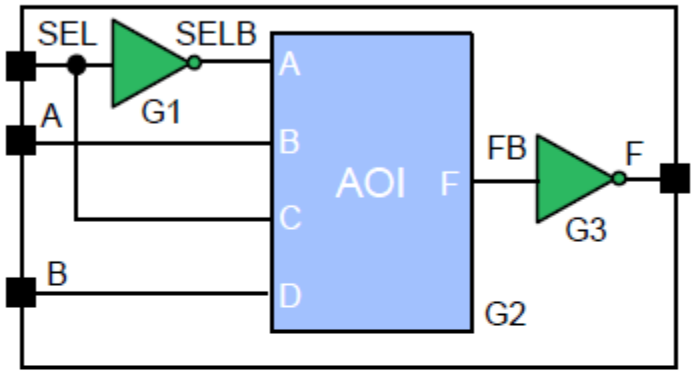
Instances of modules

Ordered mapping

Wires SELB and FB are implicit

Named Mapping

```
module AOI (A, B, C, D, F);
  ...
endmodule

module INV (A, F);
  ...
endmodule

module MUX2 (SEL, A, B, F);
  input SEL, A, B;
  output F;
  INV G1 (.A(SEL), .F(SELB));
  AOI G2 (.A(SELB), .B(A), .C(SEL), .D(B), .F(FB));
  INV G3 (.F(F), .A(FB));
endmodule
```



Named mapping

Order doesn't matter

Primitives

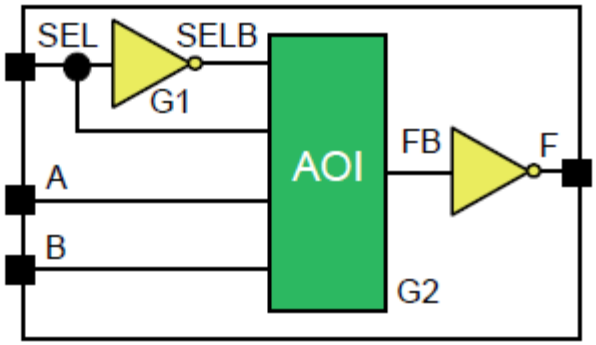
- **Gates**
  - **and, nand, or, nor, xor, xnor, buf, not**
- **Pulls, tristate buffers**
  - **pullup, pulldown, bufif0, bufif1, notif0, notif1**
- **Switches**
  - **cmos, nmos, ... tran, ...**

Built in

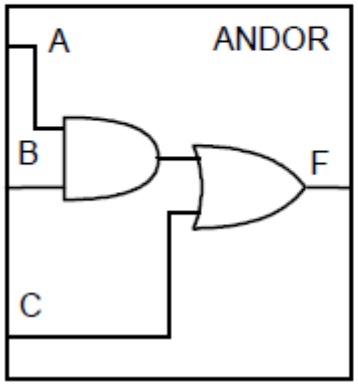
```
module MUX2 (SEL, A, B, F);
  input SEL, A, B;
  output F;
  not G1 (SELB, SEL);
  AOI G2 (SELB, A, SEL, B, FB);
  not    (F, FB);
endmodule
```

Instances of primitives

Instance name optional

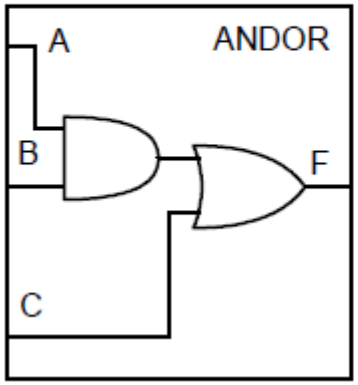


Quiz 1




```
module ANDOR (




endmodule
```

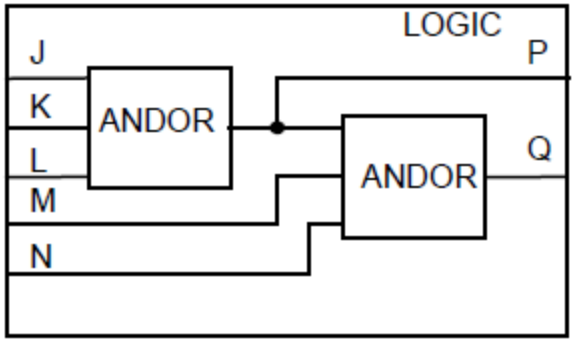
Quiz 1 – Solution




```
module ANDOR (A, B, C, F);
  input  A, B, C;
  output F;

  assign F = (A & B) | C;
endmodule
```

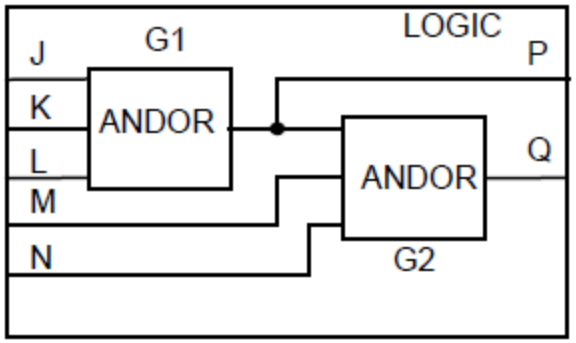
```
module ANDOR (A, B, C, F);
  input  A, B, C;
  output F;
  wire AB;

  assign AB = A & B;
  assign  F = AB | C;
endmodule
```

Quiz 2




```
module LOGIC (

endmodule
```

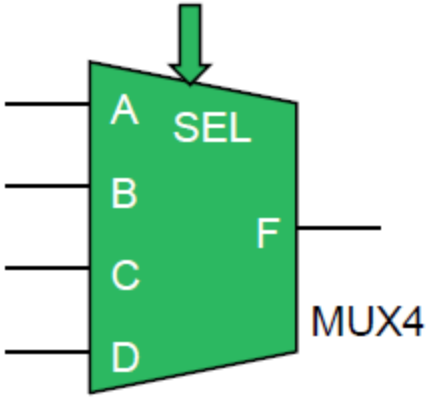
Quiz 2 – Solution




```
module LOGIC (J, K, L, M, N, P, Q);
  input  J, K, L, M, N;
  output P, Q;

  ANDOR G1 (J, K, L, P);
  ANDOR G2 (P, M, N, Q);
endmodule
```

```
module LOGIC (J, K, L, M, N, P, Q);
  input  J, K, L, M, N;
  output P, Q;

  ANDOR G1 (.A(J), .B(K), .C(L), .F(P));
  ANDOR G2 (.A(P), .B(M), .C(N), .F(Q));
endmodule
```

Vector Ports



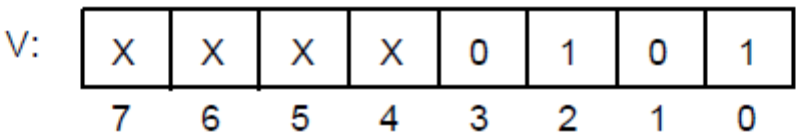

```
module MUX4 (SEL, A, B, C, D, F);
  input [1:0] SEL;
  input       A, B, C, D;
  output      F;

// ...

endmodule
```

Separate input declarations

Verilog Logic Values

- Verilog logic values are:
  - 1'b0 - logic 0, false, ground
  - 1'b1 - logic 1, true, power
  - 1'bX - unknown or uninitialised
  - 1'bZ - high impedance, floating
- A vector is a row of logic values

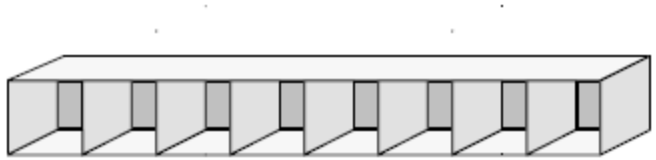
```
reg [7:0] V;
V = 8'bXXXX0101;
```



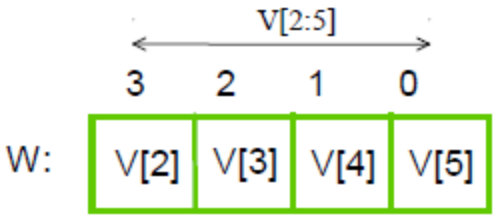

```
V[7] = 1'b0;
V[6] = B & V[0];
```

Part Selects

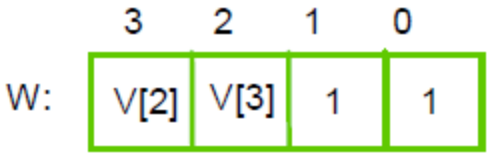
```
reg [0:7] V;
reg [3:0] W;
```



```
W = V[2:5];
```




```
W[1:0] = 2'b11;
```



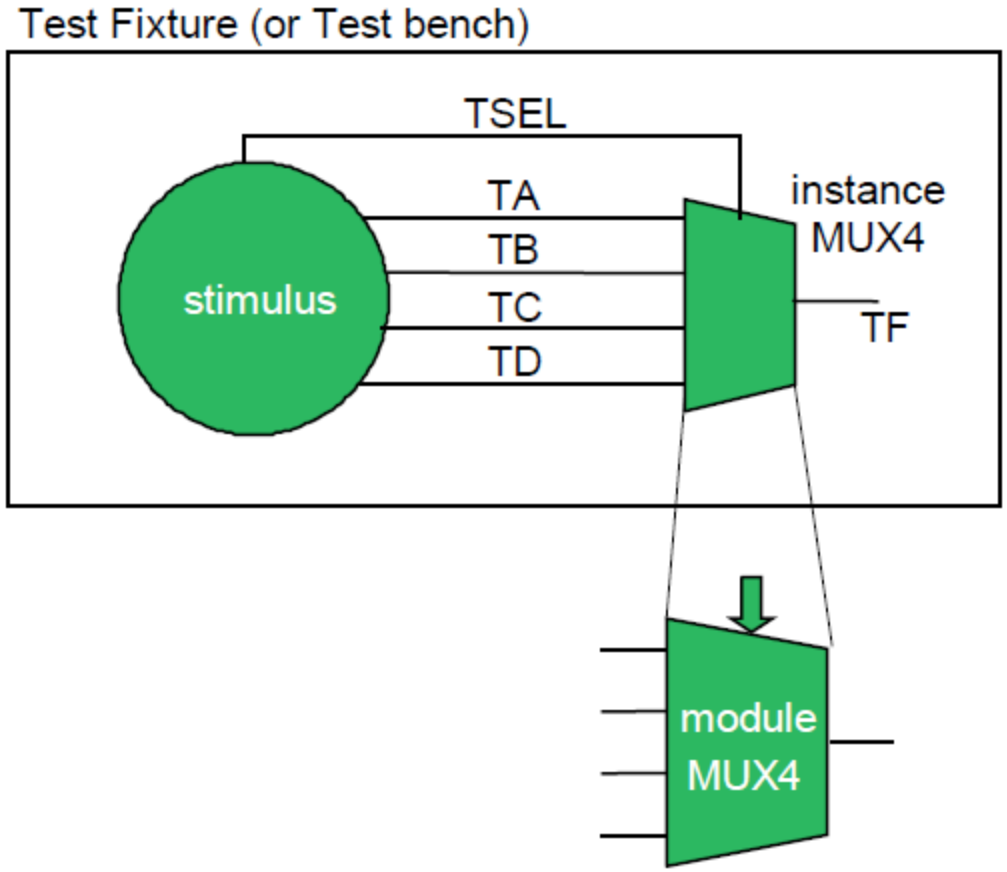

```
W[0:1] = 2'b11;
```

wrong direction - illegal

```
W[2:2] = 1'b1;
```

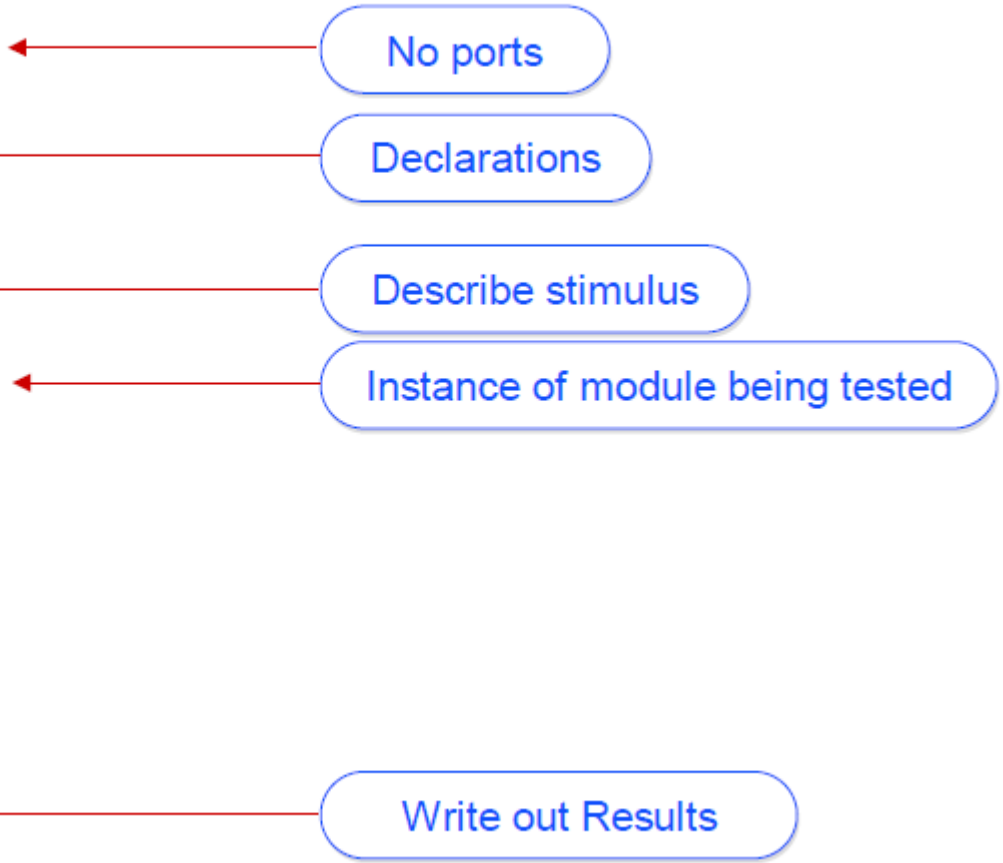
OK

Test Fixtures



Outline of Test Fixture for MUX4

```
module MUX4TEST;

  ...

  initial
    ...

  MUX4 M (
    .SEL(TSEL),
    .A(TA),
    .B(TB),
    .C(TC),
    .D(TD),
    .F(TF));

  initial
    ...

endmodule
```



Initial Blocks

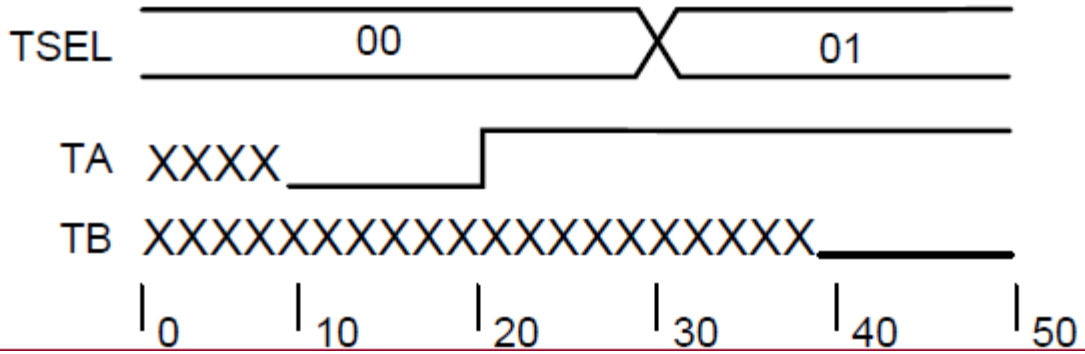
Shorthand for 1'b0

```
initial  // Stimulus
begin
       TSEL = 2'b00;
  #10  TA   = 0;
  #10  TA   = 1;
  #10  TSEL = 2'b01;
  #10  TB   = 0;
  ...
end
```

Executes once top to bottom

Procedural delay

Procedural assignments

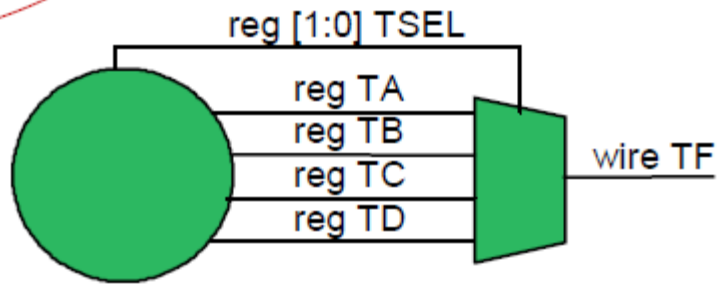


Registers

```
module MUX4TEST;

  reg TA, TB, TC, TD;
  reg [1:0] TSEL;

  wire TF;

  initial  // Stimulus
  begin
        TSEL = 2'b00;
    #10 TA   = 0;
    #10 TA   = 1;
    #10 TSEL = 2'b01;
    #10 TB   = 0;
    ...
  end

  MUX4 M (.SEL(TSEL), .A(TA), .B(TB), .C(TC), .D(TD), .F(TF));

  ...
endmodule
```

A *reg* is a kind of *register*

Target (LHS) of *procedural assignment* must be a *register*



$monitor

? **Can use $monitor to write a table of results**

```
module MUX4TEST;

  reg TA, TB, TC, TD;
  reg [1:0] TSEL;

  wire TF;

  initial  // Stimulus
    ...

  MUX4 M (.SEL(TSEL), .A(TA), .B(TB), .C(TC), .D(TD), .F(TF));

  initial  // Analysis
    $monitor($time,,TSEL,,TA,TB,TC,TD,,TF);

endmodule
```

```
 0 0 XXXX X
10 0 0XXX 0
20 0 1XXX 1
30 1 1XXX X
40 1 10XX 0
...
```

begin-end not needed

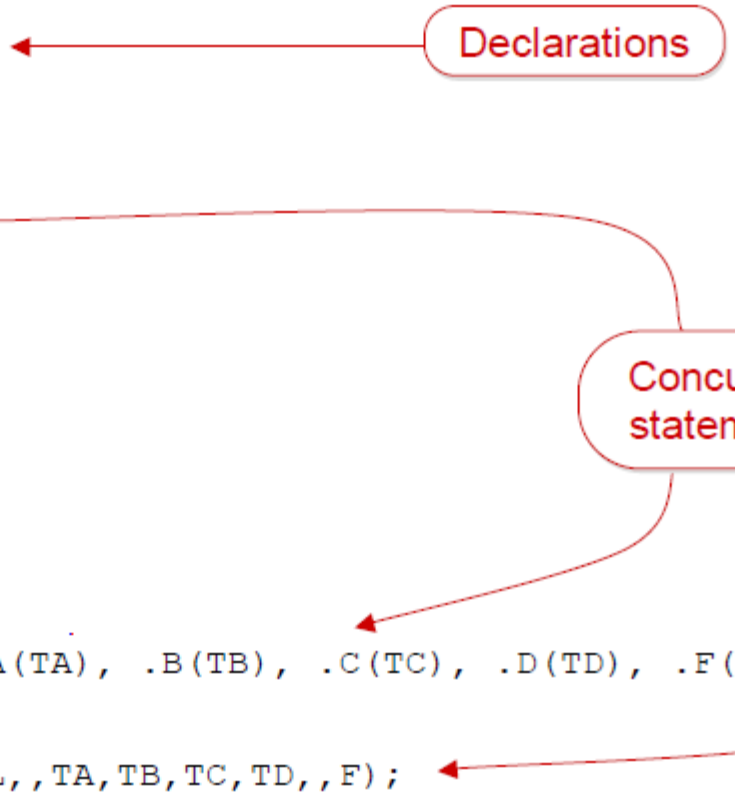
System function

writes a space

System task

The Complete Test Fixture

```
module MUX4TEST;

  reg TA, TB, TC, TD;
  reg [1:0] TSEL;
  wire TF

  initial
  begin
        SEL  = 2'b00;
    #10 TA   = 0;
    #10 TA   = 1;
    #10 TSEL = 2'b01;
    #10 TB   = 0;
    #10 TB   = 1;
    ...
  end

  MUX4 M (.SEL(TSEL), .A(TA), .B(TB), .C(TC), .D(TD), .F(TF));

  initial
    $monitor($time,,TSEL,,TA,TB,TC,TD,,F);

endmodule
```



Verilog Basics

Verilog Basics

- To become familiar with **more** basic features of the Verilog Hardware Description Language.

- Topics:
  - Numbers.
  - Formatted output.
  - Timescales.
  - System Functions ($stop and $finish).
  - Wires and Regs.
  - Operators and expressions.
  - `define and `include.
  - Conditional compilation.
  - Parameters.
  - Hierarchical names.

Numbers

```
reg [7:0] V;
```

Numbers are not case sensitive

```
V = 8'b111XX000;
```

Binary 111XX000

_ is ignored in numbers

```
V = 8'B111X_X000;
```

Binary 111XX000

```
V = 8'hFF;
```

Hex = 8'b1111_1111

Leading 0's are added

```
V = 8'o77;
```

Octal = 8'b00_111_111

```
V = 8'd10;
```

Decimal = 8'b0000_1010

"Disguised" Binary Numbers

```
reg [7:0] V;
```

Not binary!

```
V = 10;
```

10 = 'd10 = 32'd10 = 32'b1010 → 8'b00001010

```
V = -3;
```

-3 = 32'hFF_FF_FF_FD → 8'b1111_1101

```
V = "A";
```

ASCII = 8'd65 → 8'b01000001

Formatted Output

```
reg [3:0] TA, TB, TC, TD;
reg [1:0] TSEL;
wire [3:0] TF;
```

```
initial  // Write heading and table of results
begin
  $display("                    ",
           "Time TSEL TA    TB    TC    TD    TF");
  $monitor("%d %b   %b   %b   %b   %b   %b",
           $time, TSEL, TA, TB, TC, TD, TF);
end
```

```
Time TSEL TA    TB    TC    TD    TF
   0   00   0001  0010  0100  1000  0001
  10   00   1110  0010  0100  1000  1110
  20   01   1110  0010  0100  1000  0010
  30   01   1110  1101  0100  1000  1101
```

Formatting

- Special formatting strings:
  - %b binary
  - %o octal
  - %d decimal
  - %h hexadecimal
  - %s ASCII string
  - %v value and drive strength
  - %t time (described later)
  - %m module instance
  - \n newline
  - \t tab
  - \" "
  - \nnn nnn is octal ASCII value of character
  - \\ \

Formatting Text

- One format string with correct number of values

```
$display ("%b %b", Expr1, Expr2);
```

- Two format strings

Equivalent

```
$display ("%b", Expr1, " %b", Expr2);
```

- Too many values

```
$display ("%b", Expr1, Expr2);
```

Expr2 is written in decimal

- Too few values

```
$display ("%b %b %b", Expr1, Expr2);
```

ERROR!!

Always

? **Clock generator**

```
module ClockGen (Clock);
  output Clock;
  reg Clock;

  initial
    Clock = 0;

  always
    #5 Clock = ~Clock;

endmodule
```

Outputs may be regs

Binary count

```
initial
  Count = 0;

always #10
  Count = Count + 1;
```

Using Registers

- ***Necessary*** **when assigning in initial or always**
- **Only needed when assigning in initial or always**
- **Outputs can be regs**

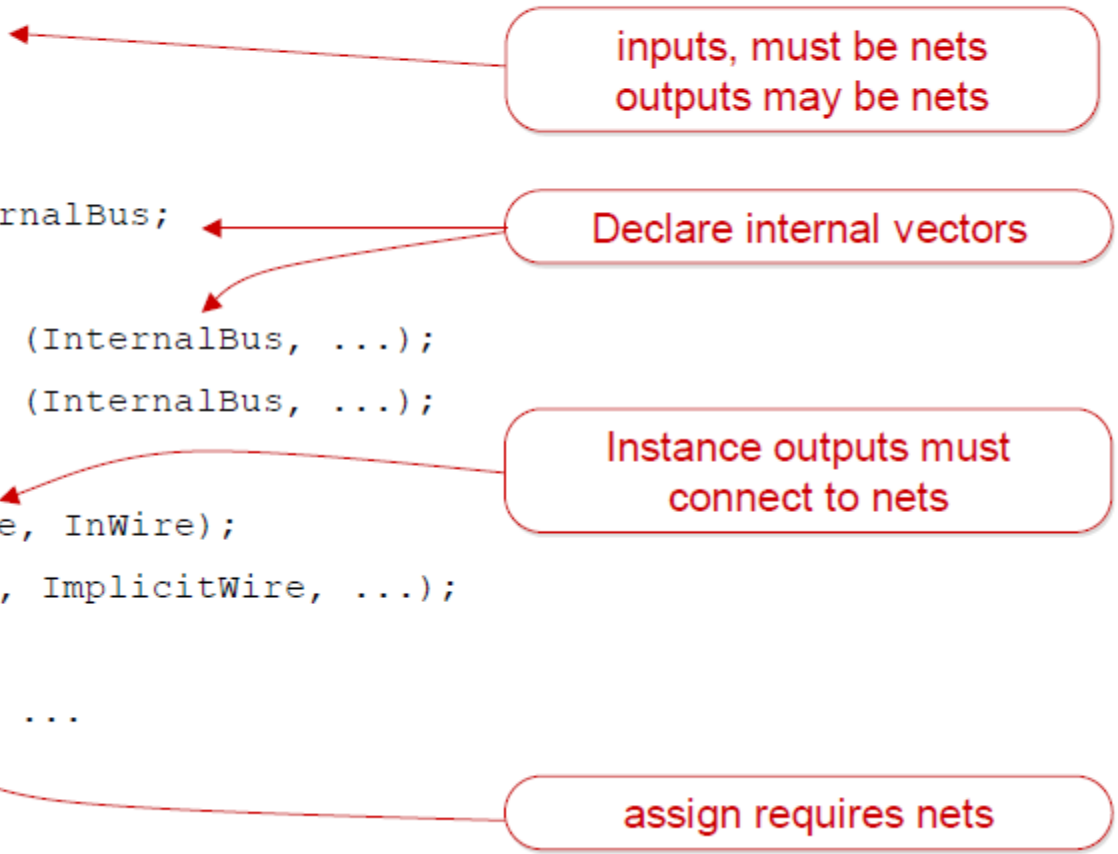
```
module UsesRegs (OutReg);
  output [7:0] OutReg;
  reg    [7:0] OutReg;

  reg R;

  initial
    R = ...

  always
    OutReg = 8'b...

endmodule
```

Declarations agree

Must be registers

Using Nets

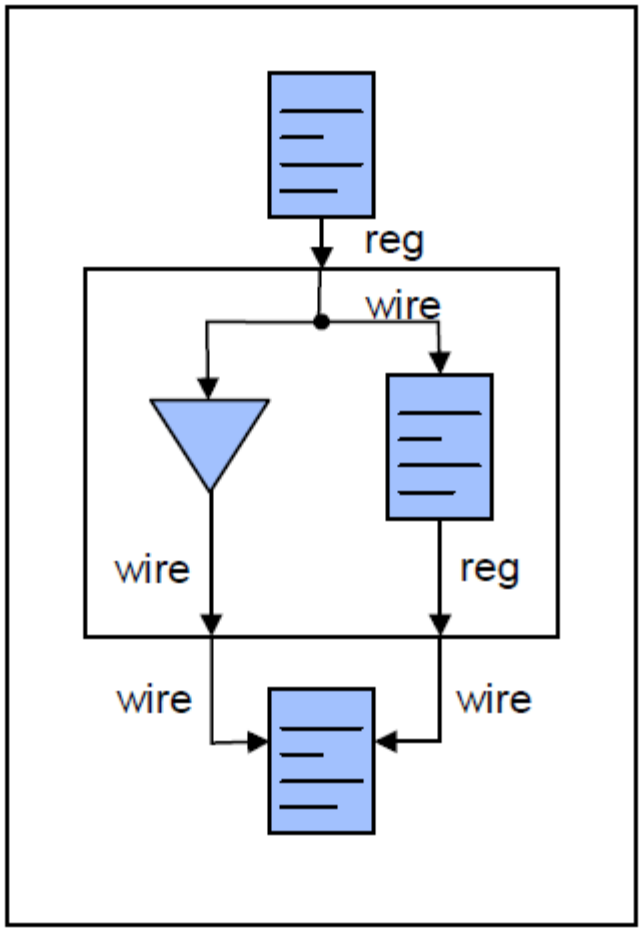
```
module UsesWires (InWire, OutWire);
  input  InWire;
  output OutWire;

  wire [15:0] InternalBus;

  AnotherModule U1 (InternalBus, ...);
  YetAnother    U2 (InternalBus, ...);

  not (ImplicitWire, InWire);
  and (AnotherWire, ImplicitWire, ...);

  assign OutWire = ...

endmodule
```



Module Boundaries

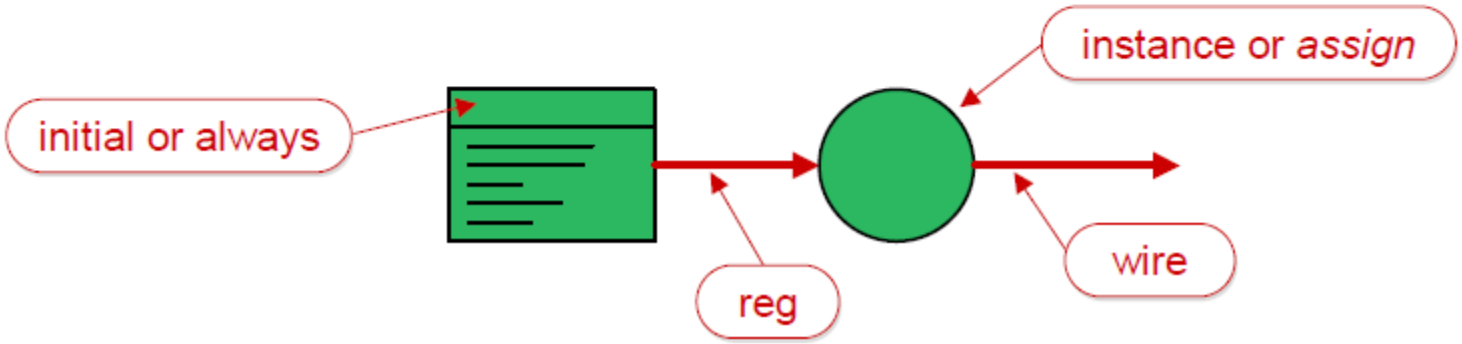
```
module Top;
   reg R;
   wire W1, W2;
   Silly S (.InWire(R), .OutWire(W1),
            .OutReg(W2));
   ...
endmodule
```

```
module Silly (InWire, OutWire,
              OutReg);
   input InWire;
   output OutWire, OutReg;
   reg OutReg;
   initial OutReg = InWire;
   assign OutWire = InWire;
endmodule
```



Wire vs. Reg – Summary

- You *must use* registers
  - when assigning in initial statements
  - when assigning in always statements
- You *must use* nets
  - when assigning in continuous assignments
  - for inputs and inouts
  - when connecting module or primitive instance outputs or inouts
- Wires are structural; regs are behavioral



Bitwise and Reduction Operators

&	Bitwise and / reduction and
|	Bitwise or / reduction or
^	Bitwise xor / reduction xor
~	Bitwise not

For example:

4'b0101 & 4'b1100	4'b0100
4'b0101 | 4'b1100	4'b1101
4'b0101 ^ 4'b1100	4'b1001
&4'b0101	1'b0
|4'b0101	1'b1
^4'b0101	1'b0
~4'b1100	4'b0011

Logical Operators

&&	Logical and
||	Logical or
!	Logical not

```
assign f = !((a && b) || (c && d));
```

```
if ( p == q && r == s )
  ...
```

Non-zero values are considered TRUE in logical expressions:

4'b0101 && 4'b1100	1'b1
!4'b1100	1'b0
!4'b0000	1'b1
4'b0XX0	1'b0
4'b01X0	1'b1

```
if (Expression)
  $display("Expression is non-zero");
else if (!Expression)
  $display("Expression is zero");
else
  $display("Expression is unknown");
```

Comparison Operators

Use to describe logic

Use in test fixtures

==	Logical Equality
!=	Logical Inequality
===	Case Equality
!==	Case Inequality

Logical Equality				
==	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

Case Equality				
===	0	1	X	Z
0	1	0	0	0
1	0	1	0	0
X	0	0	1	0
Z	0	0	0	1

```
if (A == 1'bx)     // Always false!
  F = 1'bx;        // Never executed!
```

Arithmetic Operators

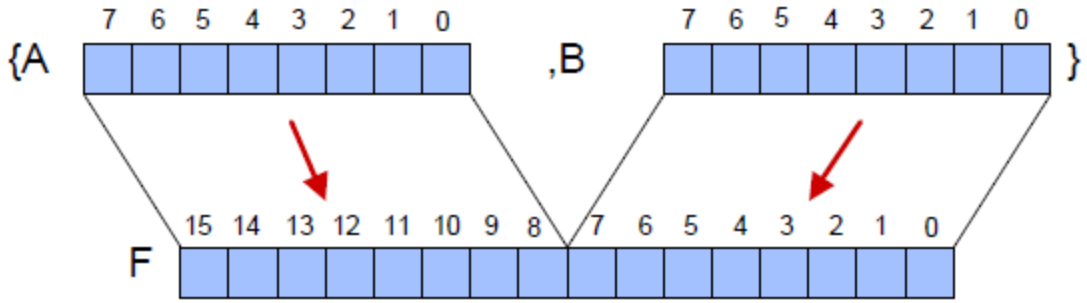
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus

Concatenation

? **Concatention operator "{ }" …**

```
reg  [7:0] A, B;
reg [15:0] F;
…
F = {A, B};
```

{A,0} is illegal



? **On left-hand side of assignment**

```
wire COUT, CIN;
wire [15:0] A, B, SUM;
assign {COUT, SUM} = A + B + CIN;
```

Replication

{N{A}}	Replication

```
{64{1'b1}}
```

```
{I+J{A}}
```

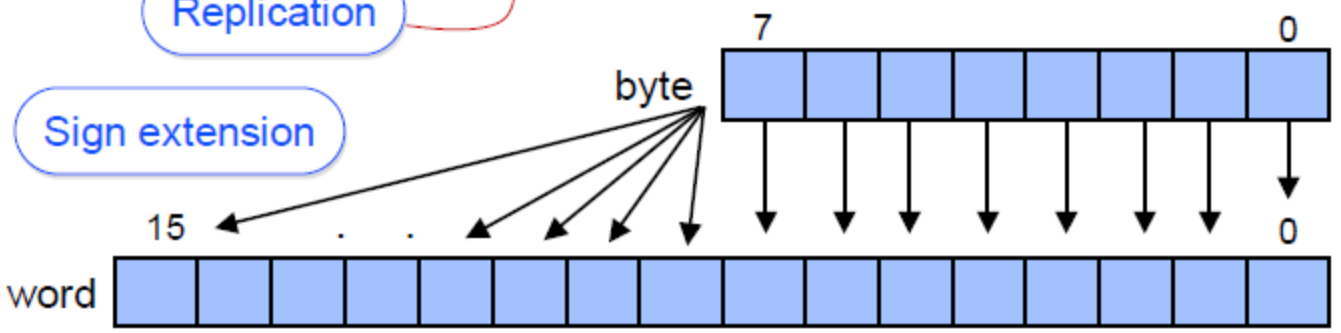
```
{32{A,B}}
```

```
reg  [7:0] byte;
reg [15:0] word;
```

Concatenation

```
word = { {8{byte[7]}}, byte };
```

Replication



Parameters

```
module M;

  parameter Width = 8;

  parameter Add   = 8'b00111100,
            Sub   = 8'b00010000,
            Load  = 8'b01010000,
            Store = 8'b11010000,
            Jump  = 8'b00101111,
            Halt  = 8'b11111111;

  parameter error_message = "Gone wrong again";
```

Parameters are declared in a module

Constant values

Any Verilog “type”

Using Parameters

- **Sizes of regs and wires**

```
reg  [Width-1:0] Bus;
wire [Width-1:0] BusWire;
```

- **"Magic Numbers"**

```
always @(Bus)
  if (Bus == Halt)
     $display("%s", error_message);
```

Not: Bus == 8'b11111111

- **Size of Numbers**

```
assign BusWire = Enable ? Bus : {Width{1'bz}};
```

Replication

Width'bz is illegal

Combinational Modeling

Combinational Modeling

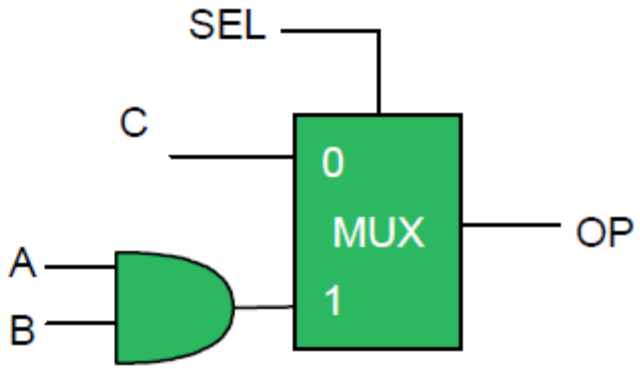
- To learn how to describe combinational logic.

- Topics:
  - RTL always statements.
  - If statements.
  - Incomplete assignment and latches.
  - Conditional Operators.
  - Tristate buffer.
  - Case statements.
  - For, while, repeat and forever loops.

RTL Always Statements

Combinational always

```
reg OP;
...
always @(SEL or A or B or C)
  if (SEL)
    OP = A and B;
  else
    OP = C;
```

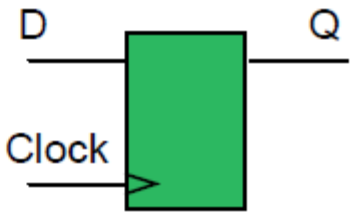


Event control

Clocked always

```
reg Q;
...
always @(posedge Clock)
  Q <= D;
```

"Non-blocking" assignment

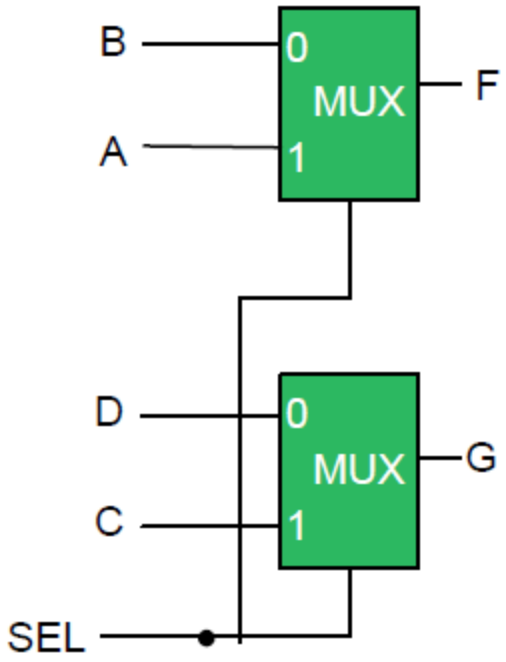


Begin-End

```
always @(SEL or A or B or C or D)
begin

  if (SEL)
  begin
    F = A;
    G = C;
  end
  else
  begin
    F = B;
    G = D;
  end

end
```

begin-end optional

begin-end required



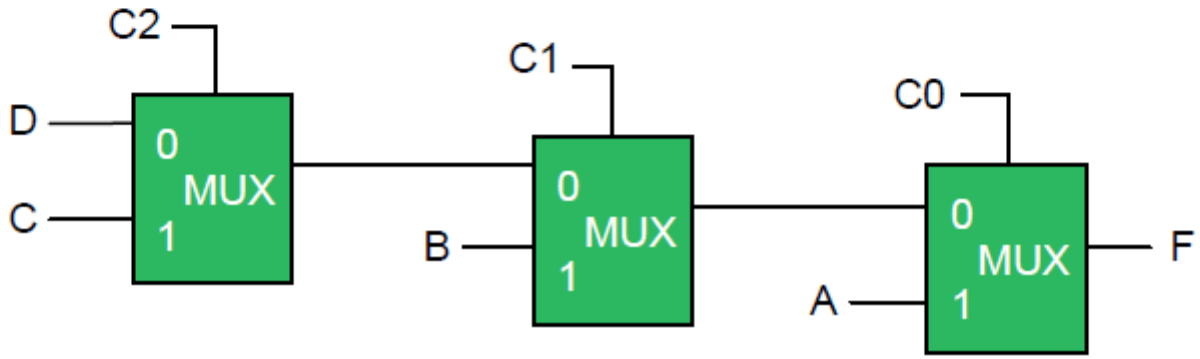
Else If

```
always @(C0 or C1 or C2 or A or B or C or D)
  if (C0)
    F = A;
  else if (C1)
    F = B;
  else if (C2)
    F = C;
  else
    F = D;
```

Complete “sensitivity list”

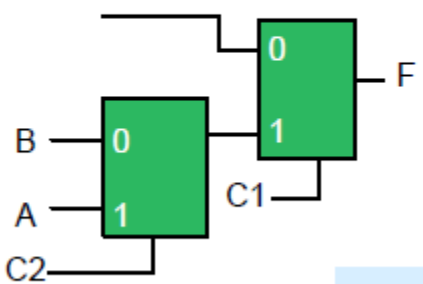
Nested if statements

Else if =>priority



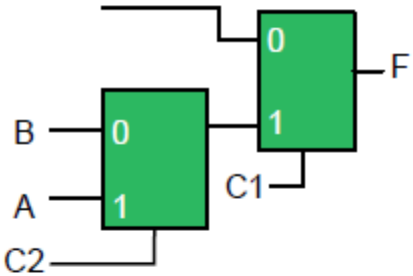
Nested If and Begin-End

```
if (C1)
  if (C2)
    F = A;
  else
    F = B;
```

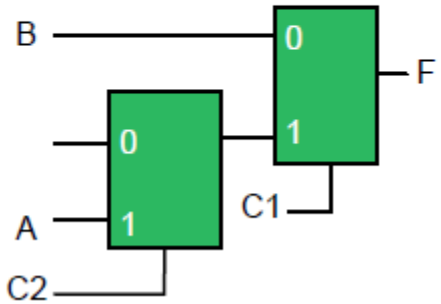


Same

```
if (C1)
  if (C2)
    F = A;
else
  F = B;
```



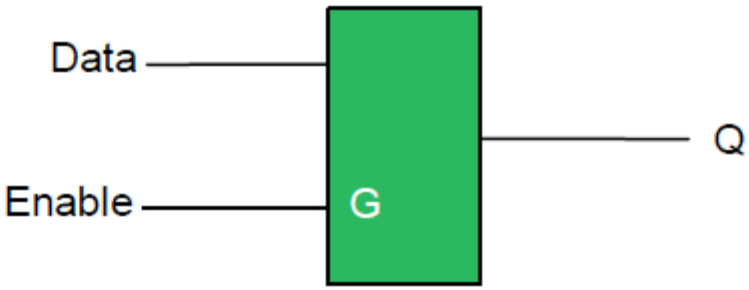

```
if (C1)
begin
  if (C2)
    F = A;
end
else
  F = B;
```



Verilog cannot read your mind!

Incomplete Assignment

```
always @(Enable or Data)
  if (Enable)
    Q = Data;
```



Beware unwanted latches!

Conditional Operator

A ? B : C	Conditional

```
value = A ? B : C;
```

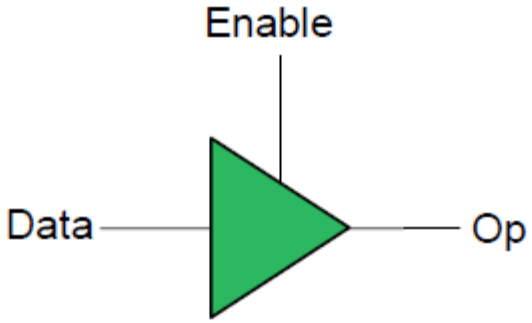
- **Equivalent to:**

```
if (A)
  value = B;
else
  value = C;
```

- **Mux using continuous assignment**

```
assign F = SEL ? A : B;
```

Tristates



Using *assign*

```
assign Op = Enable ? Data : 1'bZ;
```

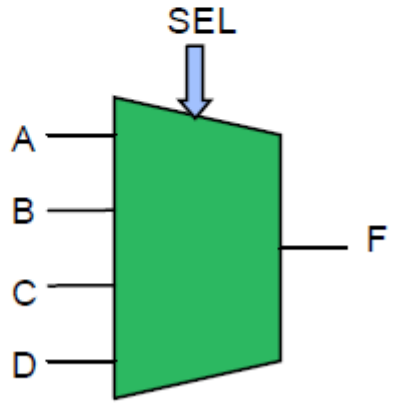
Using *always*

```
always @(Enable or Data)
  if (Enable)
    Op = Data;
  else
    Op = 1'bz;
```

Case Statement

```
always @(SEL or A or B or C or D)
  case (SEL)
    2'b00:   F = A;
    2'b01:   F = B;
    2'b10:   F = C;
    2'b11:   F = D;
    default: F = 1'bX;
  endcase
```

Simulation only



Case Statement (Cont.)

```
case (Code)
  3'b000:
    begin
      P = 1;
      Q = 1;
    end
  3'b001, 3'b010, 3'b100:
    begin
      Q = 1;
      R = 1;
    end
  3'b110, 3'b101, 3'b011:
    R = 1;
endcase
```

begin-end needed here

Alternatives

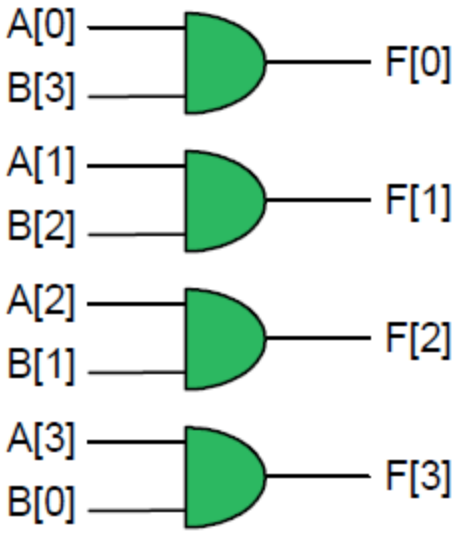
Some cases missing - OK

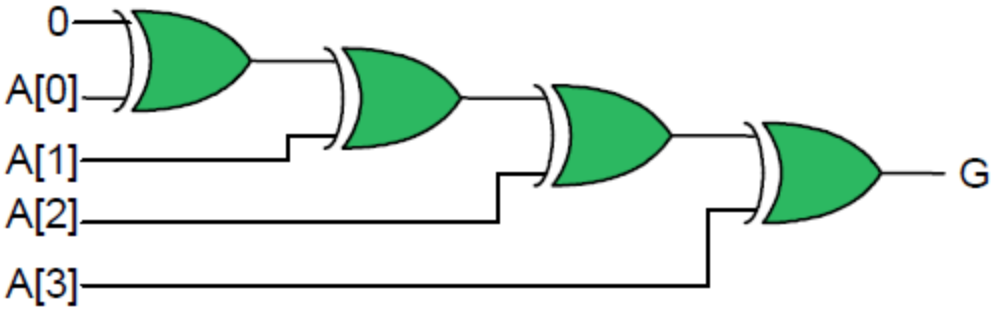
Latches may be inferred

For Loops

```
always @(A or B)
begin
  G = 0;
  for (I = 0; I < 4; I = I + 1)
  begin
    F[I] = A[I] & B[3-I];
    G = G ^ A[I];
  end
end
```

for => repeated H/W





Other Loop Statements

- The *for* loop can be used for synthesis

```
for (initialise; condition; assignment)
   ...
```

- Three other loops used for test fixtures and algorithms

```
while (condition)
  ...
```

Loops while condition is true

```
repeat (expression)
  ...
```

Loops *expression* times

```
forever
  ...
```

Infinite loop

Sequential Modeling

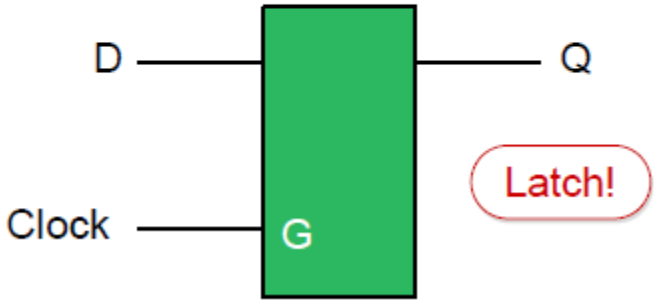
Sequential Modeling

- To learn how to describe Sequential logic.

- Topics:
  - Latch – Flip Flop.
  - Blocking – Non-Blocking assignment.
  - Shift Register.
  - Counter.
  - Finite State Machine (FSM).
  - Memory.

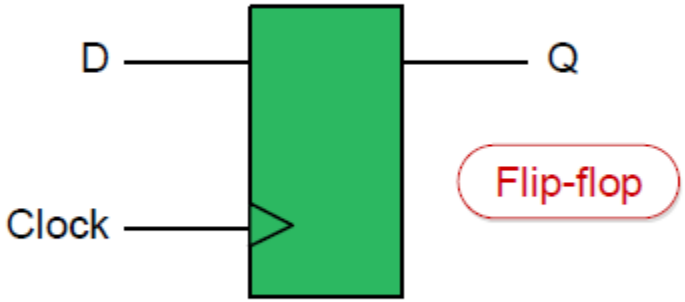
Edge Triggered Flip-Flop

```
always @(Clock)
  if (Clock)
    Q = D;
```




```
always @(posedge Clock)
  Q = D;
```

or negedge



Avoiding Simulation Races

- **Race to read and write b!**

```
always @(posedge clock)
  b = a;

always @(posedge clock)
  c = b;
```

- **To fix the simulation race...**

```
always @(posedge clock)
begin
  tmpa = a;
  #1 b = tmpa;
end

always @(posedge clock)
begin
  tmpb = b;
  #1 c = tmpb;
end
```

Not recommended!

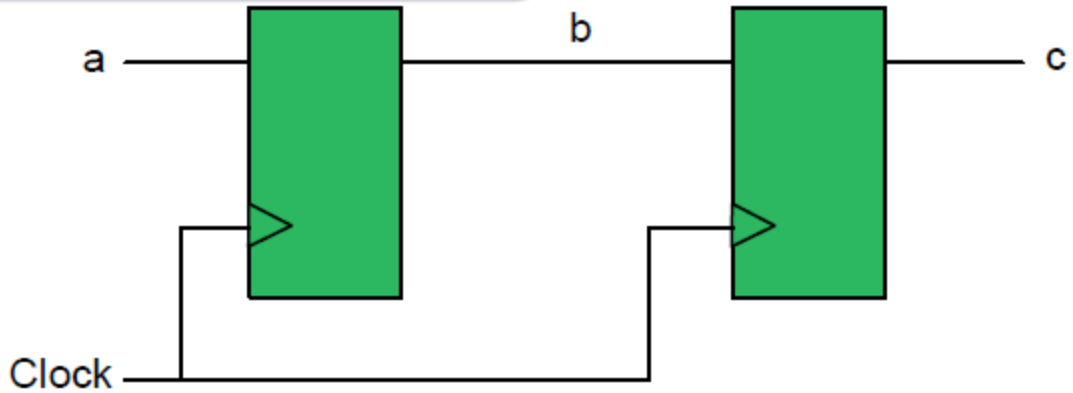
Non-Blocking Assignments

? **Preferred solution – use *non-blocking* assignments**

```
always @(posedge Clock)
  b <= a;

always @(posedge Clock)
  c <= b;
```

Recommended

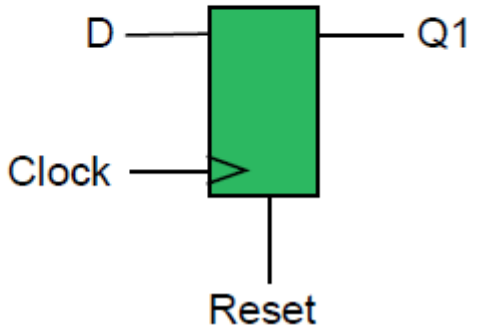
"Non-blocking" or "RTL" assignment



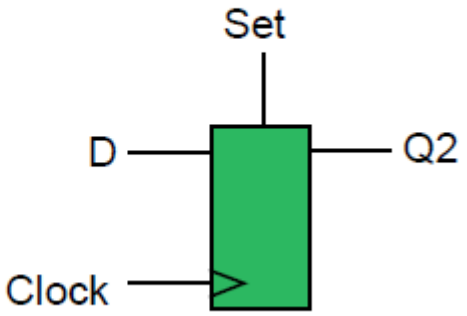
How would you write an asynchronous reset?

Asynchronous Set or Reset

```
always @(posedge Clock or posedge Reset)
  if (Reset)
    Q1 <= 0;
  else
    Q1 <= D;
```




```
always @(posedge Clock or posedge Set)
  if (Set)
    Q2 <= 1;
  else
    Q2 <= D;
```

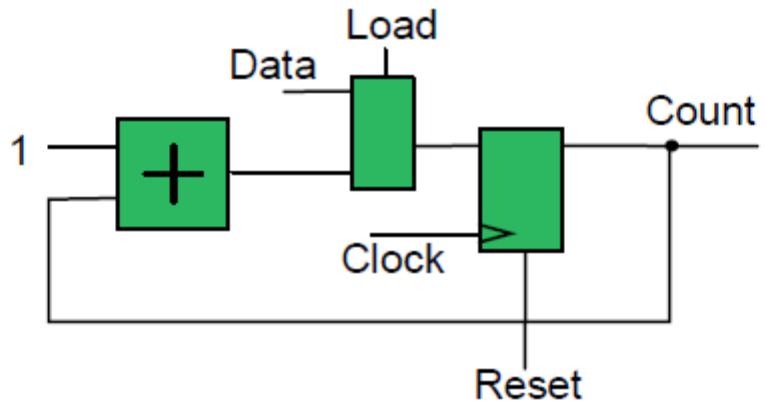


Synchronous vs. Asynchronous Actions

```
always @(posedge Clock or posedge Reset)
  if (Reset)
    Q <= 0;
  else if (Load)
    Q <= Data;
  else
    Q <= Q + 1;
```

Asynchronous reset

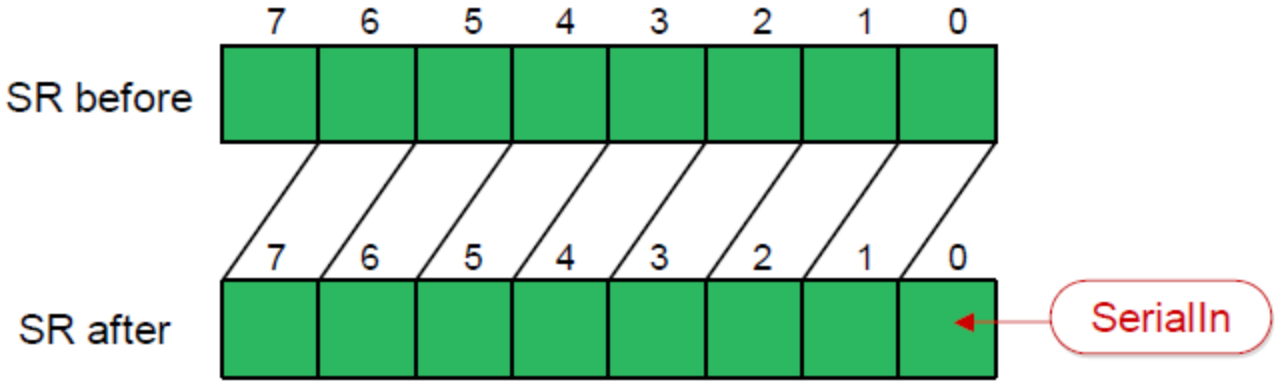
Synchronous load



Shift Registers

Shift register using concatenation

```
reg [7:0] SR;

always @(posedge Clock or posedge Reset)
  if (Reset)
    SR <= 8'b0;
  else
    SR <= {SR[6:0],SerialIn};
```

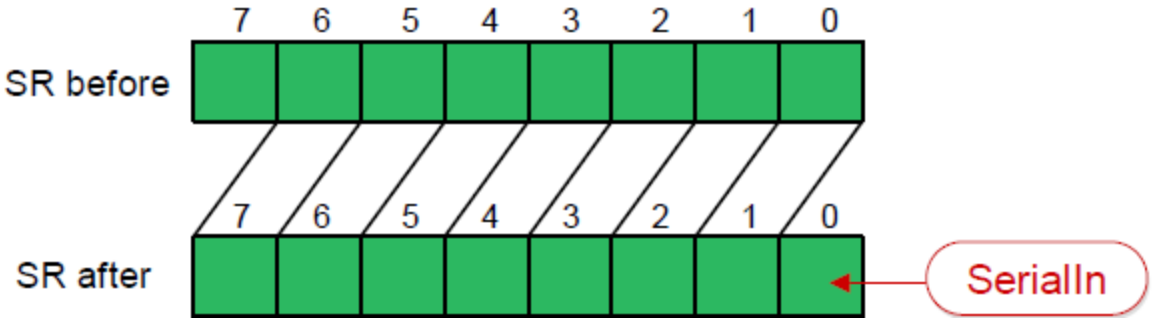


Shift Operators

<<	Left Shift
>>	Right Shift

Shift register using shift operator

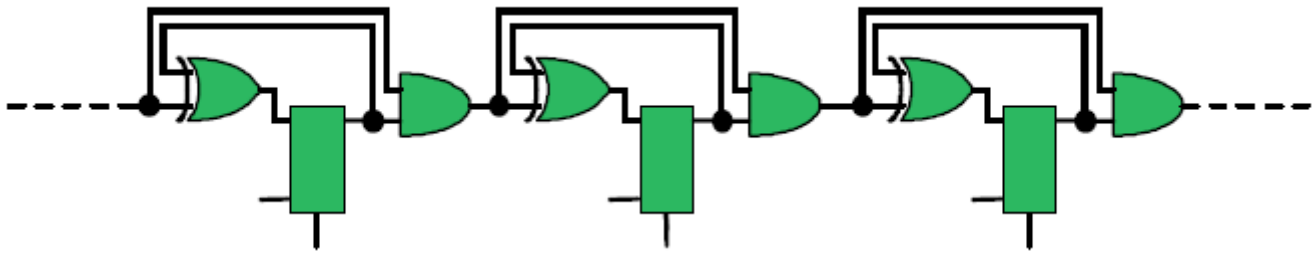
```
always @(posedge Clock or posedge
Reset)
  if (Reset)
    SR <= 8'b0;
  else
    begin
      SR <= SR << 1;
      SR[0] <= SerialIn;
    end
```



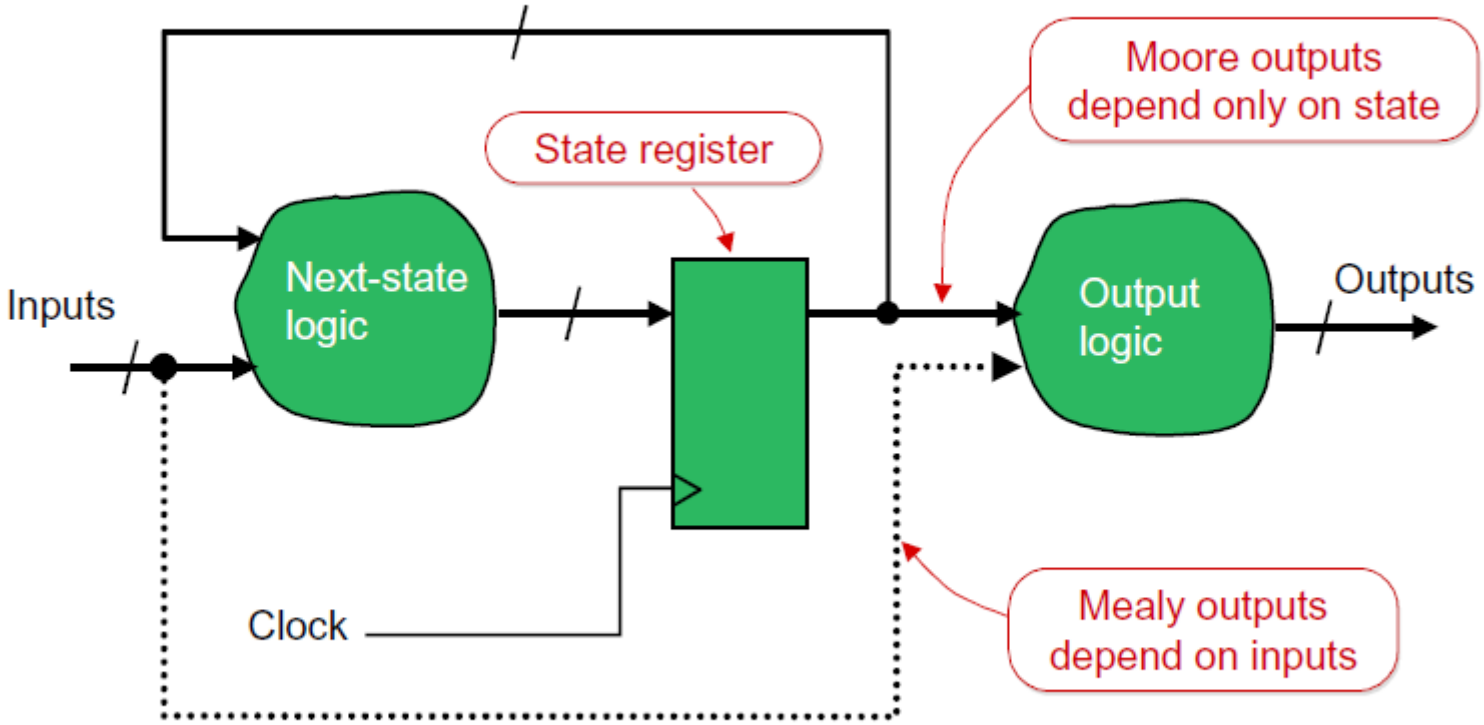
Counters

```
reg [3:0] count;

always @(posedge clk)
  count <= count + 1;
```

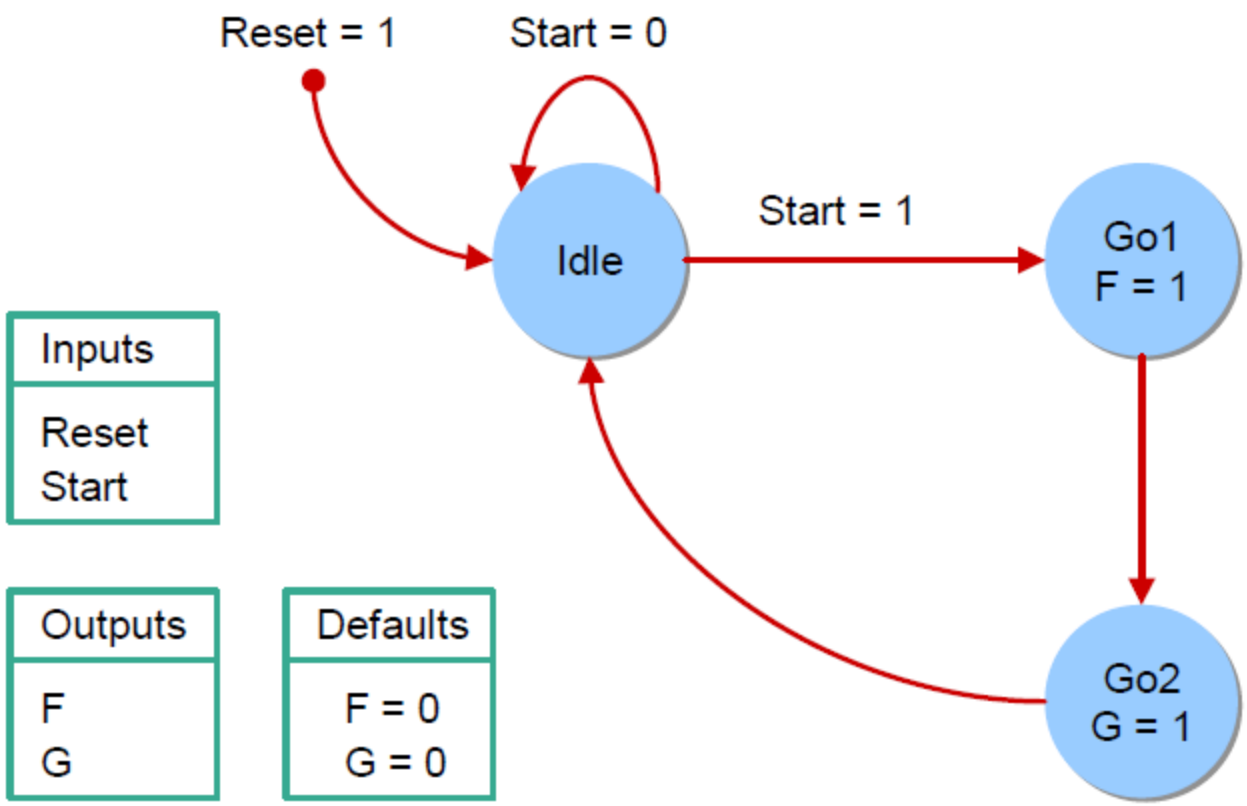
4'b1111 + 1 = 4'b0000



Finite State Machines



State Transition Diagrams



Explicit State Machine Description

```
parameter Idle = 2'b00,
          Go1  = 2'b01,
          Go2  = 2'b10;
reg [1:0] State;
...
always @(posedge Clock or posedge Reset)
  if (Reset)
  begin
    State <= Idle;
    F <= 0;
    G <= 0;
  end
  else
    case (State)
      Idle : if (Start)
             begin
               State <= Go1;
               F <= 1;
             end
      Go1  : begin
               State <= Go2;
               F <= 0;
               G <= 1;
             end
      Go2  : begin
               State <= Idle;
               G <= '0';
             end
    endcase
```

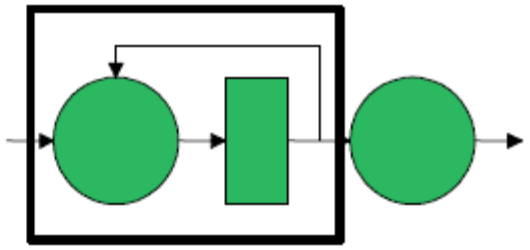
FSM should have a reset

How many flip-flops?

Separate Output Decoding

```
parameter Idle = 2'b00,
          Go1  = 2'b01,
          Go2  = 2'b10;
reg [1:0] State;

...

always @(posedge Clock or posedge Reset)
  if (Reset)
    State <= Idle;
  else
    case (State)
      Idle : if (Start)
               State <= Go1;
      Go1  : State <= Go2;
      Go2  : State <= Idle;
    endcase
```

```
always @(State)
begin
  F = 0;
  G = 0;
  if (State == Go1)
    F = 1;
  else if (State == Go2)
    G = 1;
end
```



Unreachable States

```
parameter Idle = 2'b00,
          Go1  = 2'b01,
          Go2  = 2'b10;
reg [1:0] State;
```

```
case (State)
  Idle : ...
  Go1  : ...
  Go2  : ...
endcase
```

State Name	Binary Encoding
Idle	2'b00
Go1	2'b01
Go2	2'b10
-	2'b11

- 3 states
- 2 flip-flops (binary or gray code)
- 4 hardware states
- 1 unreachable state

No control over 4th state

Logic may be minimized

Controlling Unreachable States

```
parameter Idle  = 2'b00,
          Go1   = 2'b01,
          Go2   = 2'b10,
          Dummy = 2'b11;
reg [1:0] State;
```

Either

```
parameter Idle = 2'b00,
          Go1  = 2'b01,
          Go2  = 2'b10;
reg [1:0] State;
```

```
case (State)
  Idle : ...
  Go1  : ...
  Go2  : ...
  default : State = Idle
endcase
```

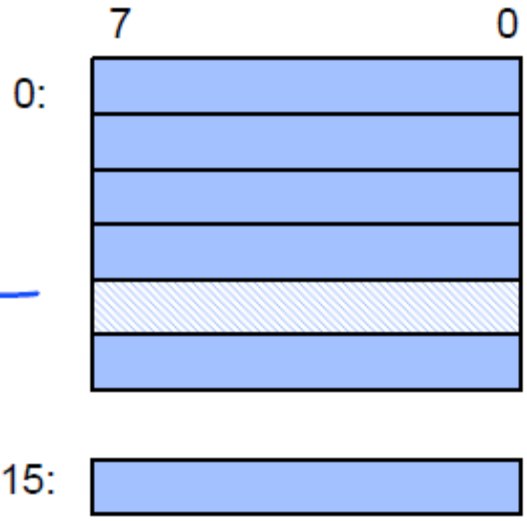
Explicitly define behavior of hardware in unreachable state

Memories

Width

Depth

```
reg [7:0] MEM [0:15];
```




```
reg [7:0] WORD;
reg [3:0] ADDR;


initial
begin
  ADDR = 4;
  WORD = MEM[ADDR];
  MEM[ADDR] = {WORD[3:0], WORD[7:4]};  // Nibble swap
end
```

Memories vs. Regs

- **An eight bit reg**

```
reg [7:0] R;
```

```
R = 8'b00001111;
```

OK

```
R[3:0] = 4'b1111;
```

- **An eight word by 1 bit memory array**

```
reg R[7:0];
```

```
R = 8'b00001111;
```

```
R[3:0] = 4'b1111;
```

Illegal references to memory

```
R[7] = 1'b1;
```

OK

RAMs

```
module smallram (clock, wr, rd, addr, data);
  input clock, wr, rd;
  input [3:0] addr;
  inout [7:0] data;

  reg [7:0] mem [0:15];

  always @(posedge clock)
    if (wr)
      mem[addr] = data;

  assign data = rd ? mem[addr]: 8'bZ;

endmodule
```

4 address bits = 16 addresses

16x8 memory

Dynamic word select

data is an inout...

What would be synthesized?

Loading Memories

? **$readmemb and $readmemh load memories from text files**

```
module rom (addr, data);
  input  [7:0] addr;
  output [7:0] data;
  reg [7:0] arom [0:'H1FF];
  assign data = arom[addr];
  initial
    $readmemb("rom.txt", arom);
endmodule
```

$readmemh - hex data

$readmemb - binary data

rom.txt

```
0000_0101          // Load at address 0
0110_1110          // Load at address 1
10011111 01100111 // Load at addresses 2 & 3
@100               // Skip addresses 4 to FF
1111_1100          // Load at address 100 (Hex)
```

Address (Hex)

Comments

Underscores are allowed