CS222: Computer Architecture

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

الاحترام - الادب - الاخلاق
الطالب - المعيد - الدكتور

Verilog HDL

- **FPGA.**
- **Verilog**
  - Introduction to Verilog HDL.
  - Verilog Basics.
  - Combinational modeling.
  - Sequential modeling.

Digital System Implementation Spectrum

Hardware
ASIC

Reconfigurable Architectures

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

Software

- µProcessor
- µController
- DSP

flexibility trade-off efficiency

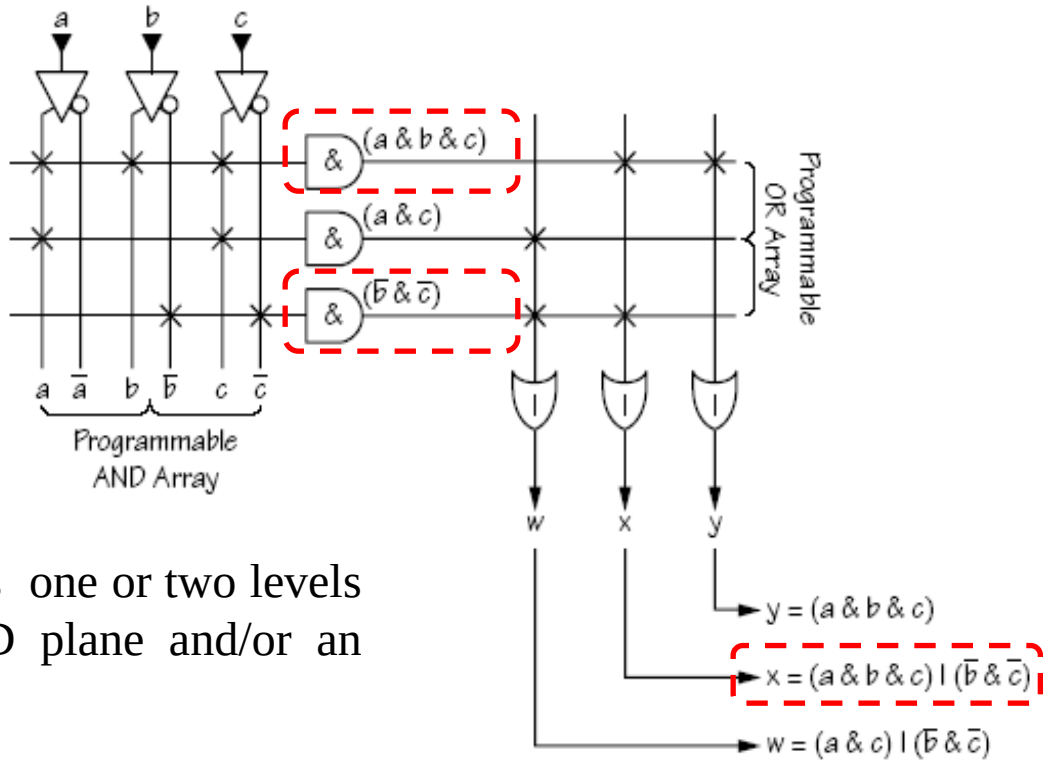
Reconfigurable Architectures

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

Programmable logic Devices

Logic Functions

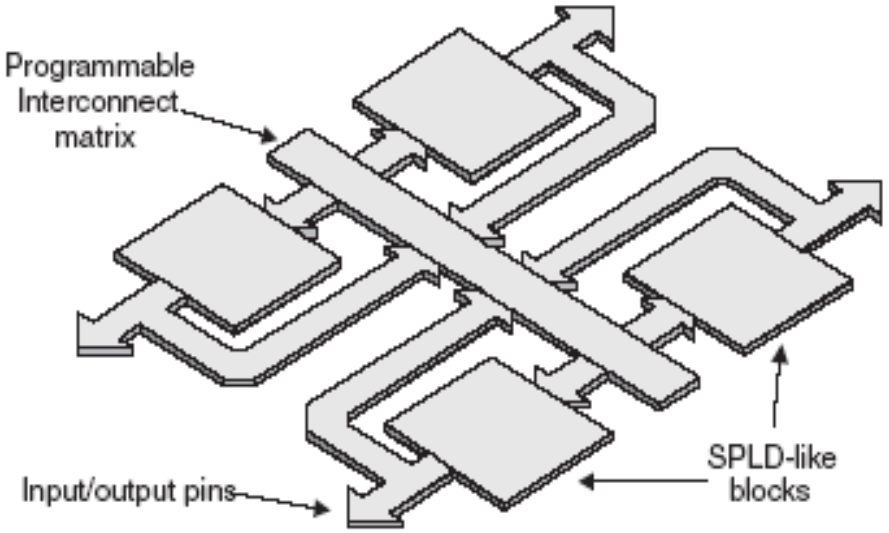
- $y = (a \,\&\, b \,\&\, c)$
- $x = (a \,\&\, b \,\&\, c) \mid (\overline{b} \,\&\, \overline{c})$
- $w = (a \,\&\, c) \mid (\overline{b} \,\&\, \overline{c})$

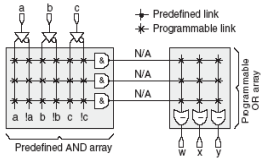


Relatively small FPD that contains one or two levels of programmable logic—an AND plane and/or an OR plane.

Complex PLD

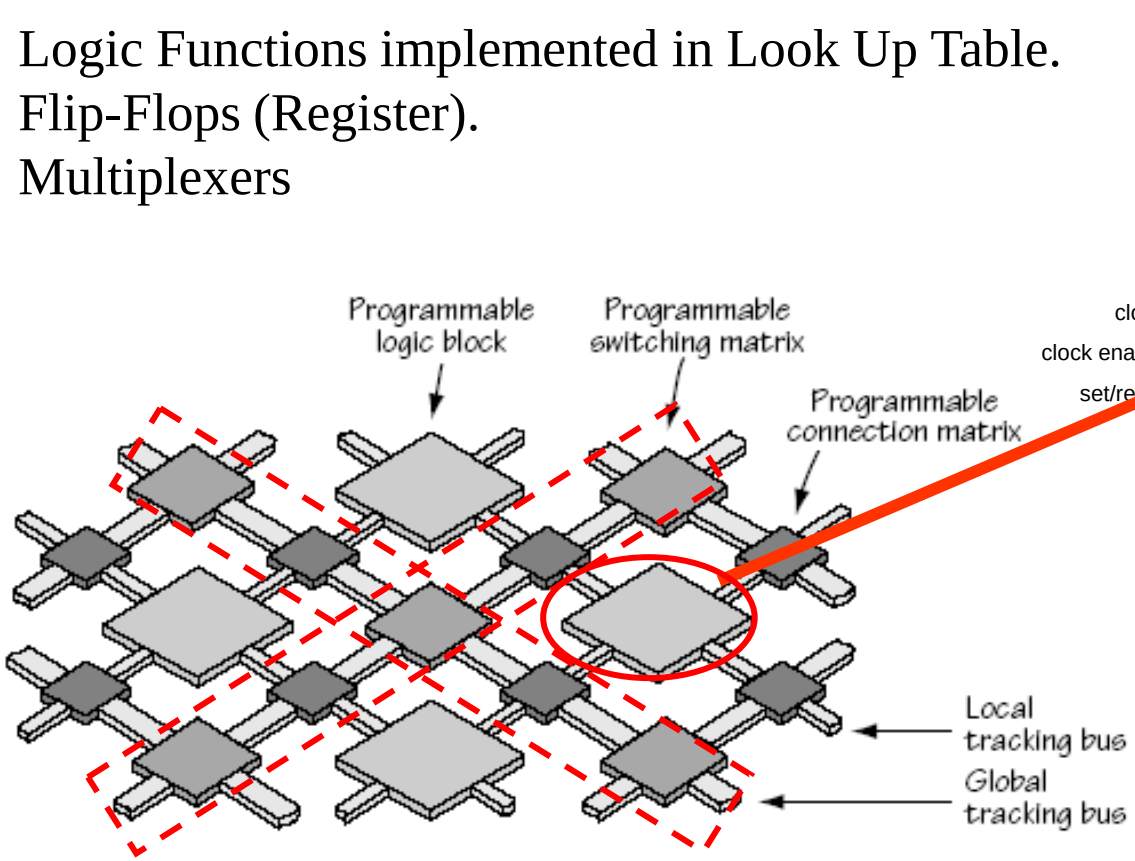
- Arrangement of multiple SPLD-like blocks on a single chip.
- Programmable PLD Blocks, Programmable Interconnects

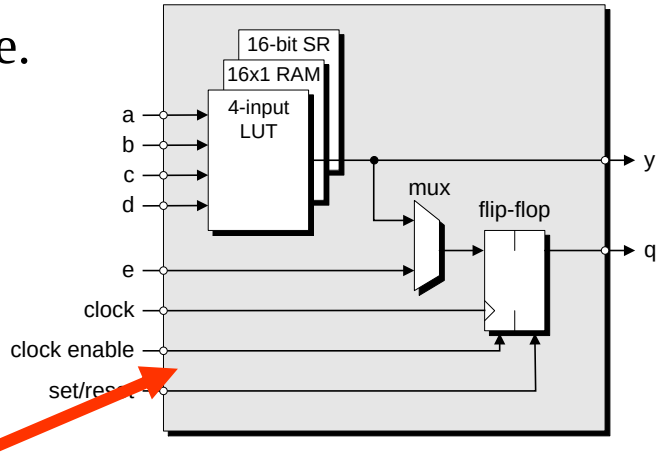




FPGA

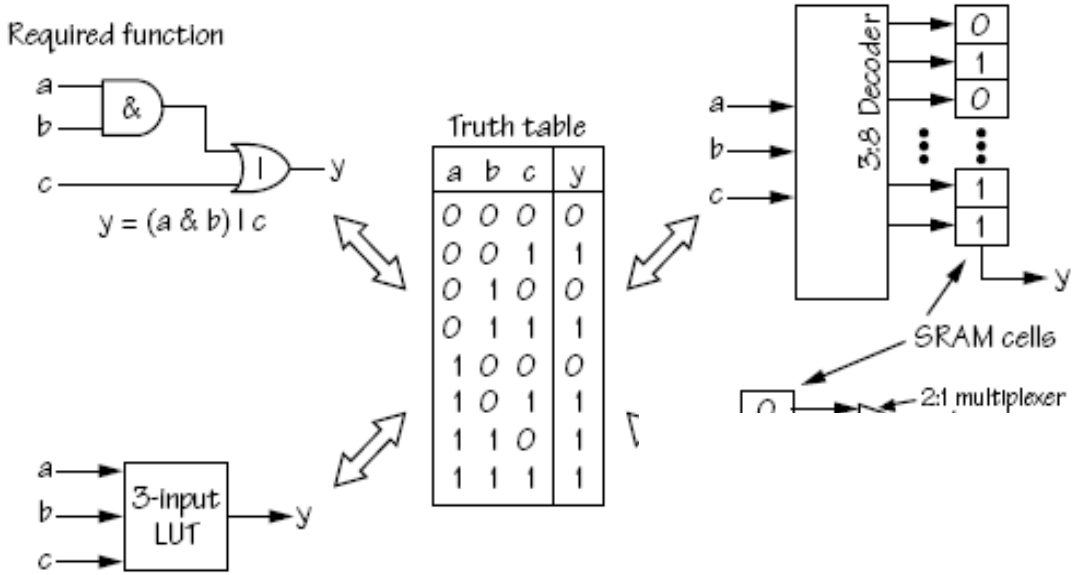
- Logic Functions implemented in Look Up Table.
- Flip-Flops (Register).
- Multiplexers





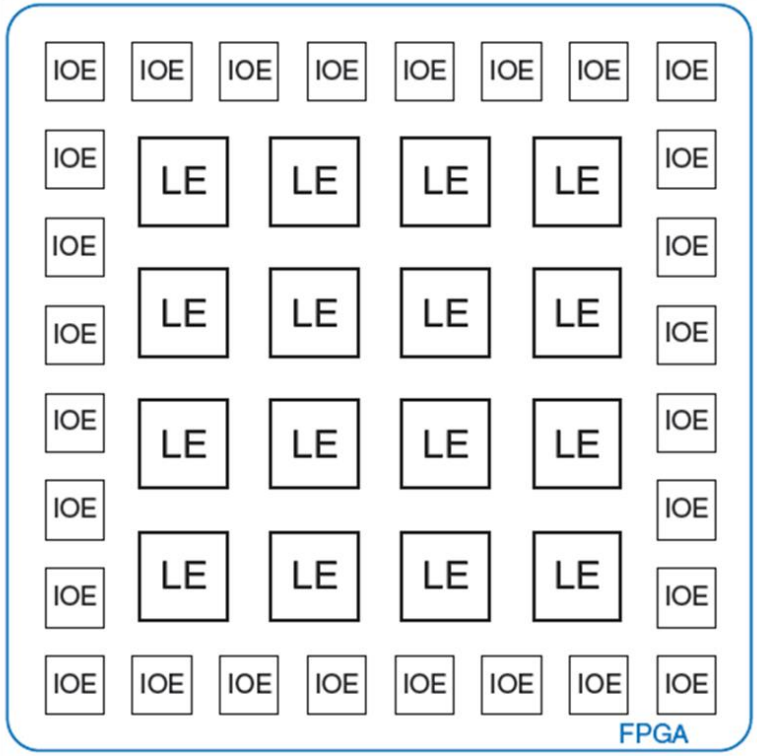
FPGA - LUT

- 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



General FPGA Layout

- Composed of:
  - **LEs** (Logic elements): perform logic
    - **LUTs** (lookup tables): perform combinational logic
    - **Flip-flops:** perform sequential logic
    - **Multiplexers:** connect LUTs and flip-flops)
  - **IOEs** (Input/output elements): interface with outside world
  - **Programmable interconnection:** connect LEs and IOEs



  - Some FPGAs include other building blocks such as multipliers and RAMs

FPGA – Design Flow

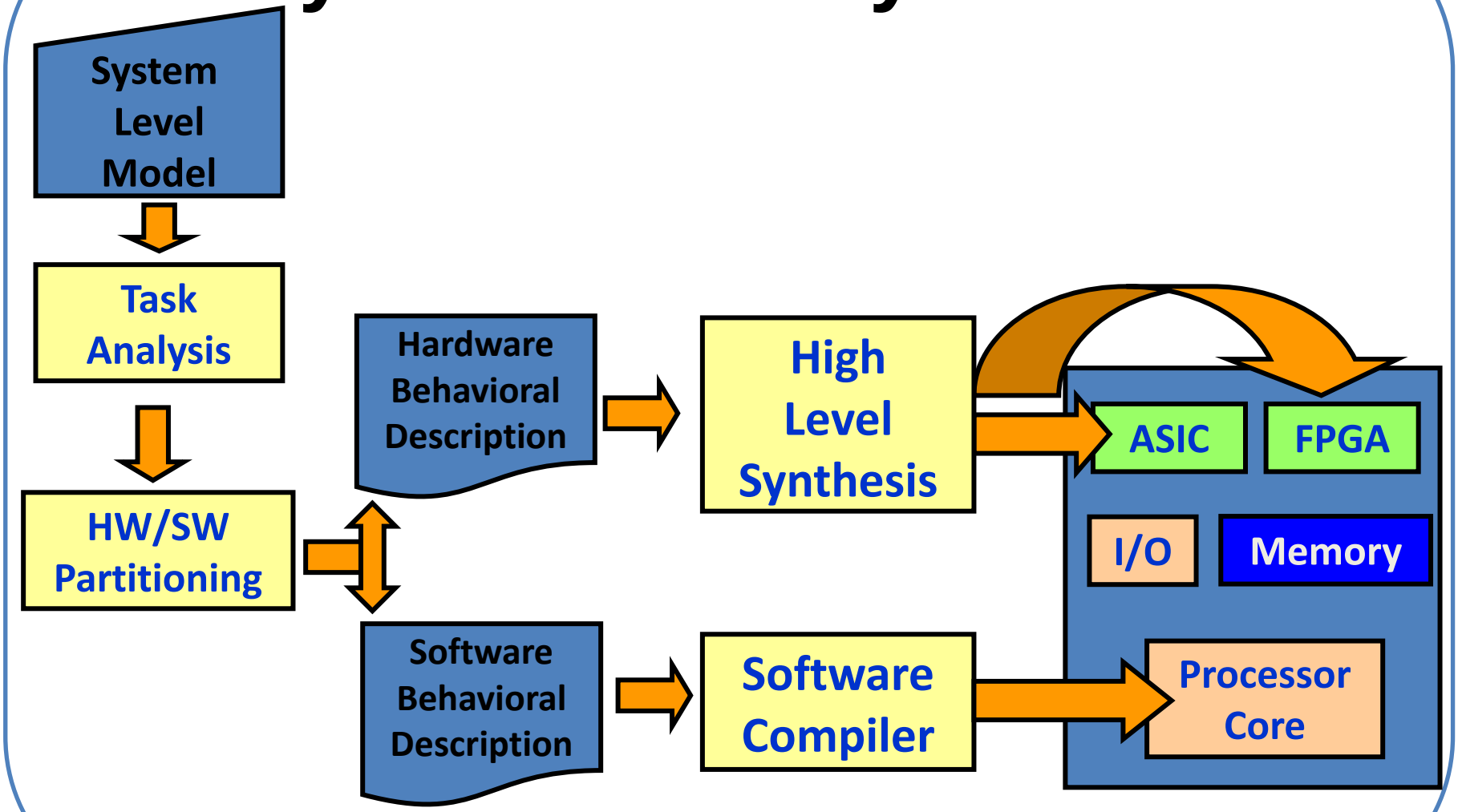
- Reading Specs and Define the full definition of the problem
- Detailed Specs and architecture of the project.
- Behavioral design
  - High level Description of Logic Design.
  - Schematic.
  - State machine.
  - Flow chart.
  - Block diagram.
- Function Verification (Pre-Synthesis Simulation).
- Synthesis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

Altera's Quartus II

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




Silicon Run 1 2nd Edition.rm - VLC media player
Media Playback Audio Video Subtitle Tools View Help
14:21
38:30
65%

[Silicon Run]

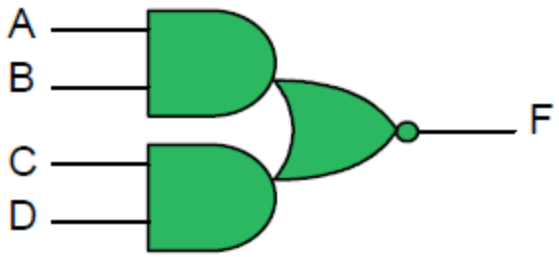
Verilog Overview

Verilog Overview

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

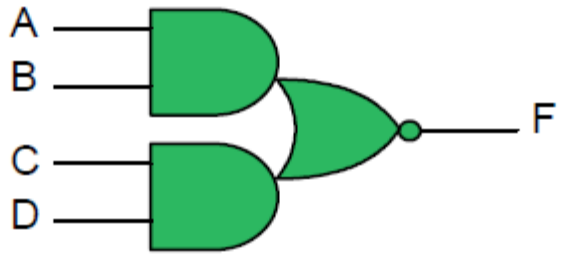
- Topics:
  - Modules.
  - Ports.
  - Continuous assignments.
  - Hierarchy.
  - Logic Values.
  - Test fixtures.
  - Initial blocks.

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

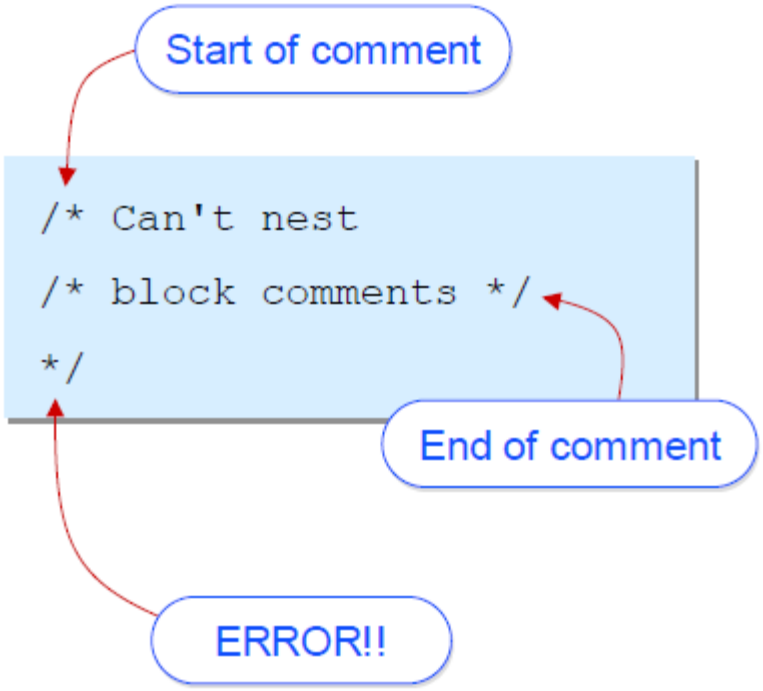
Lower case keywords

Single-line vs. Block Comments

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

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

```
/* Can't nest
/* block comments */
*/
```



Names

- **Identifiers**

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

- **Illegal!**

```
4plus4          $1
```

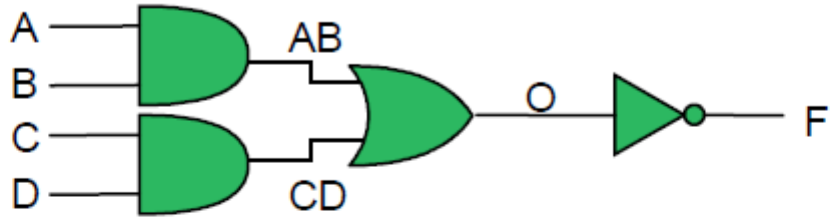
- **Escaped identifiers (terminate with white space)**

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

- **Keywords**

```
and   default   event   function   wire
```

Wires

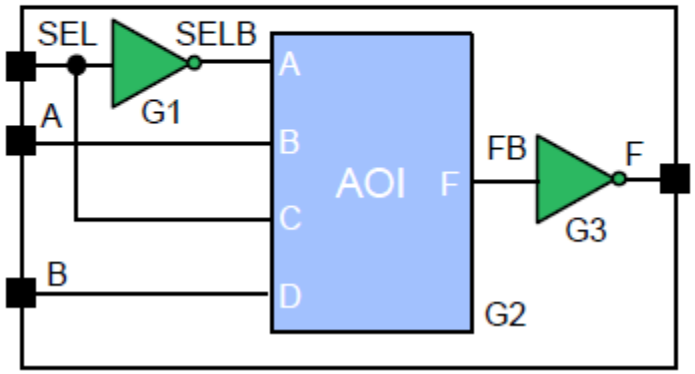



```
module AOI (A, B, C, D, F);
  input A, B, C, D;
  output F;

  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
```



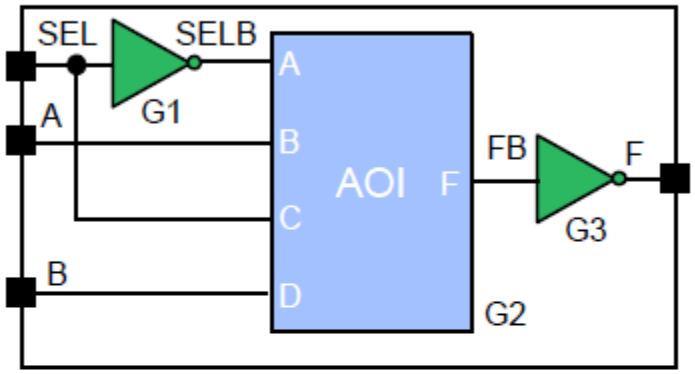
Instances of modules

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

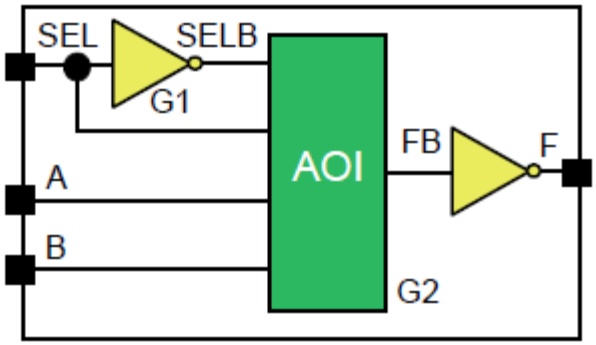
- **Gates**
  - **and, nand, or, nor, xor, xnor, buf, not**
- **Pulls, tristate buffers**
  - **pullup, pulldown, bufif0, bufif1, notif0, notif1**
- **Switches**
  - **cmos, nmos, ... tran, ...**

Built in

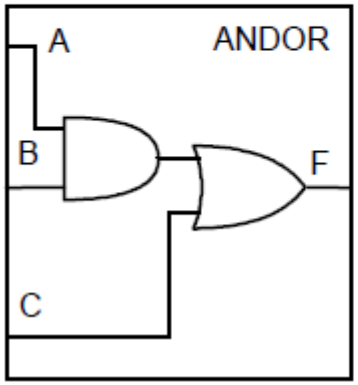
```
module MUX2 (SEL, A, B, F);
  input SEL, A, B;
  output F;
  not G1 (SELB, SEL);
  AOI G2 (SELB, A, SEL, B, FB);
  not    (F, FB);
endmodule
```

Instances of primitives

Instance name optional

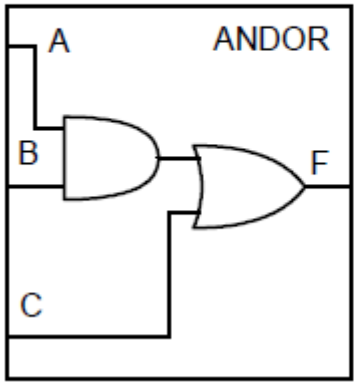


Quiz 1




```
module ANDOR (




endmodule
```

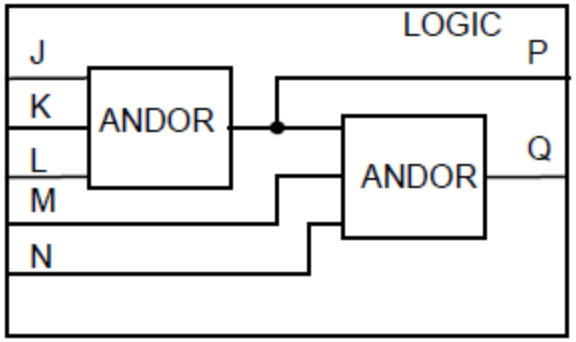
Quiz 1 – Solution




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

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

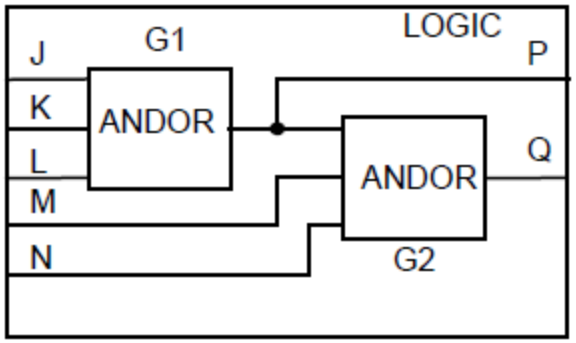
```
module ANDOR (A, B, C, F);
  input  A, B, C;
  output F;
  wire AB;

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

Quiz 2




```
module LOGIC (

endmodule
```

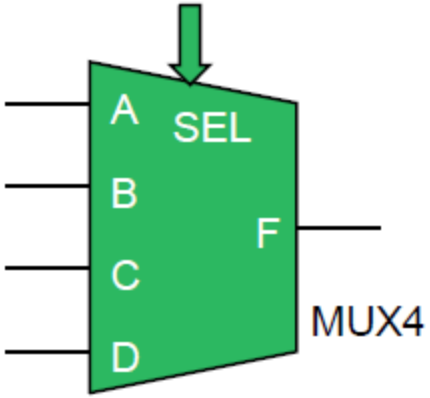
Quiz 2 – Solution




```
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



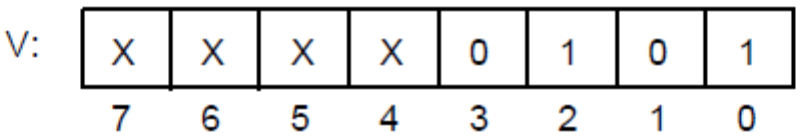

```
module MUX4 (SEL, A, B, C, D, F);
  input [1:0] SEL;
  input       A, B, C, D;
  output      F;

// ...

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

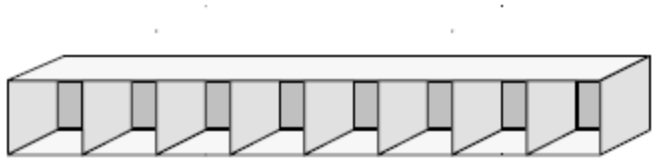
```
reg [7:0] V;
V = 8'bXXXX0101;
```



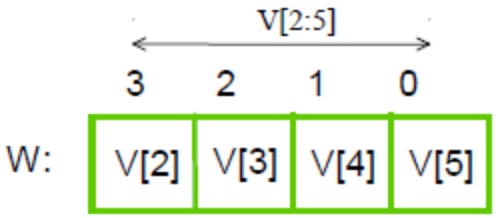

```
V[7] = 1'b0;
V[6] = B & V[0];
```

Part Selects

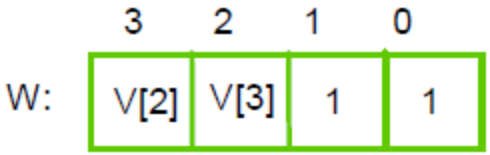
```
reg [0:7] V;
reg [3:0] W;
```



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




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



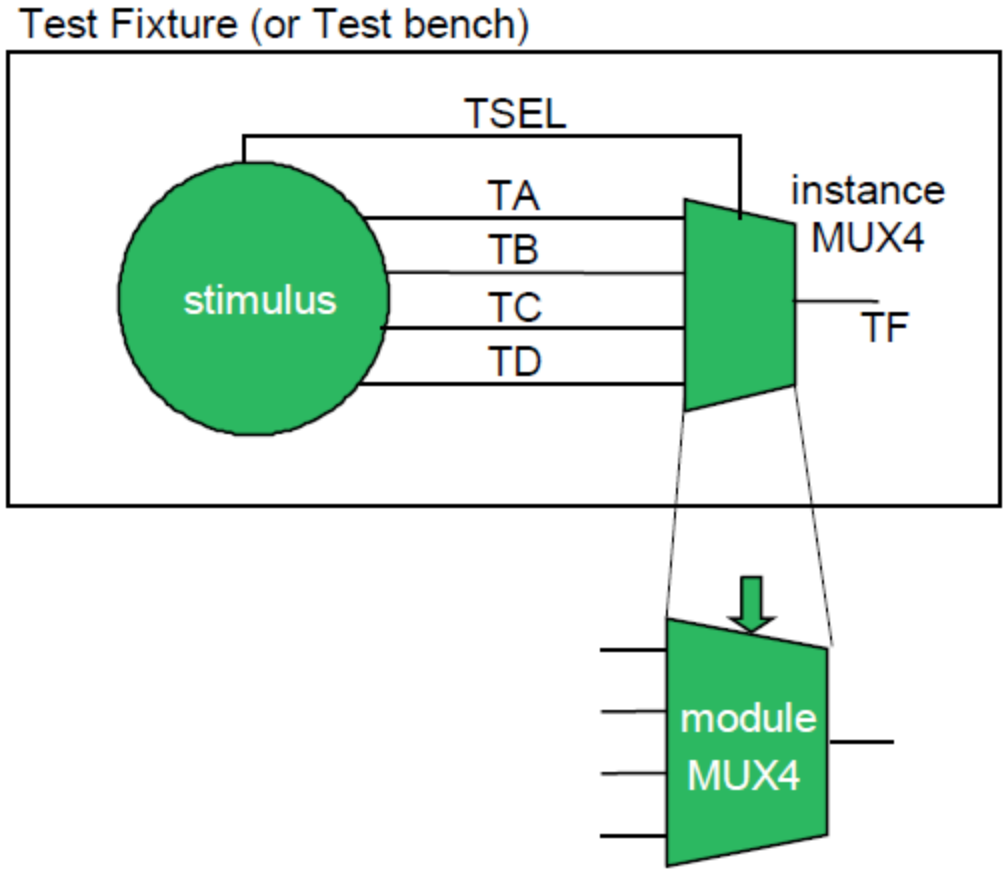

```
W[0:1] = 2'b11;
```

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
```

No ports

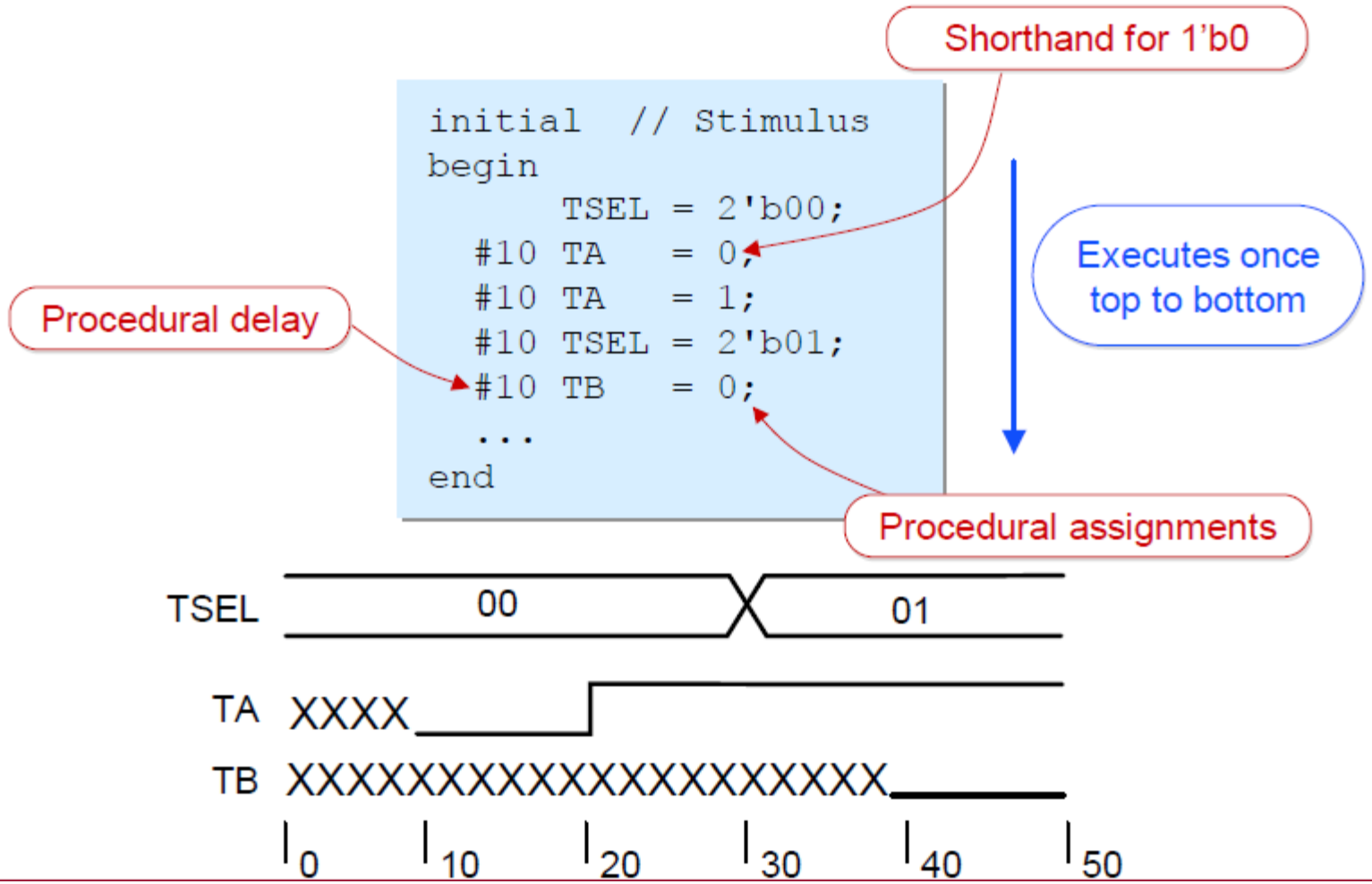
Declarations

Describe stimulus

Instance of module being tested

Write out Results

Initial Blocks

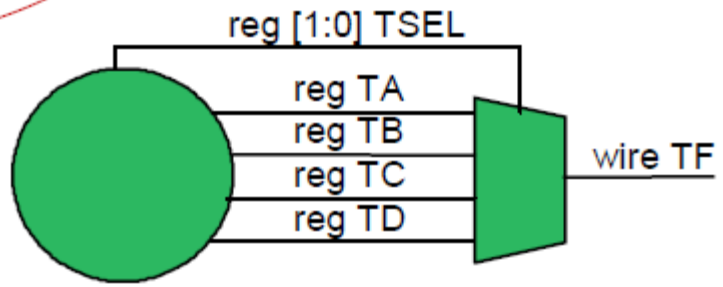


Registers

```
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

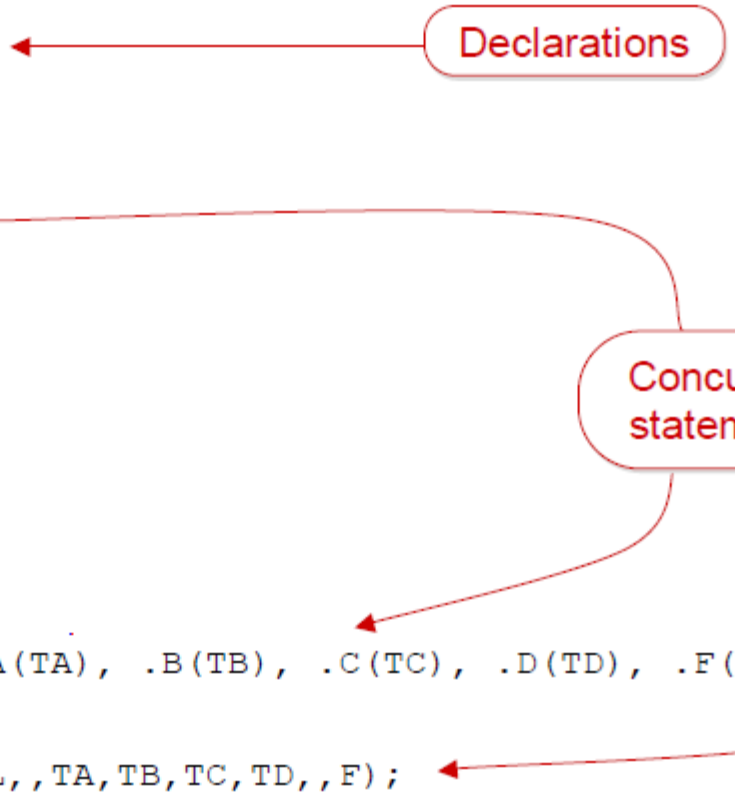
System function

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**

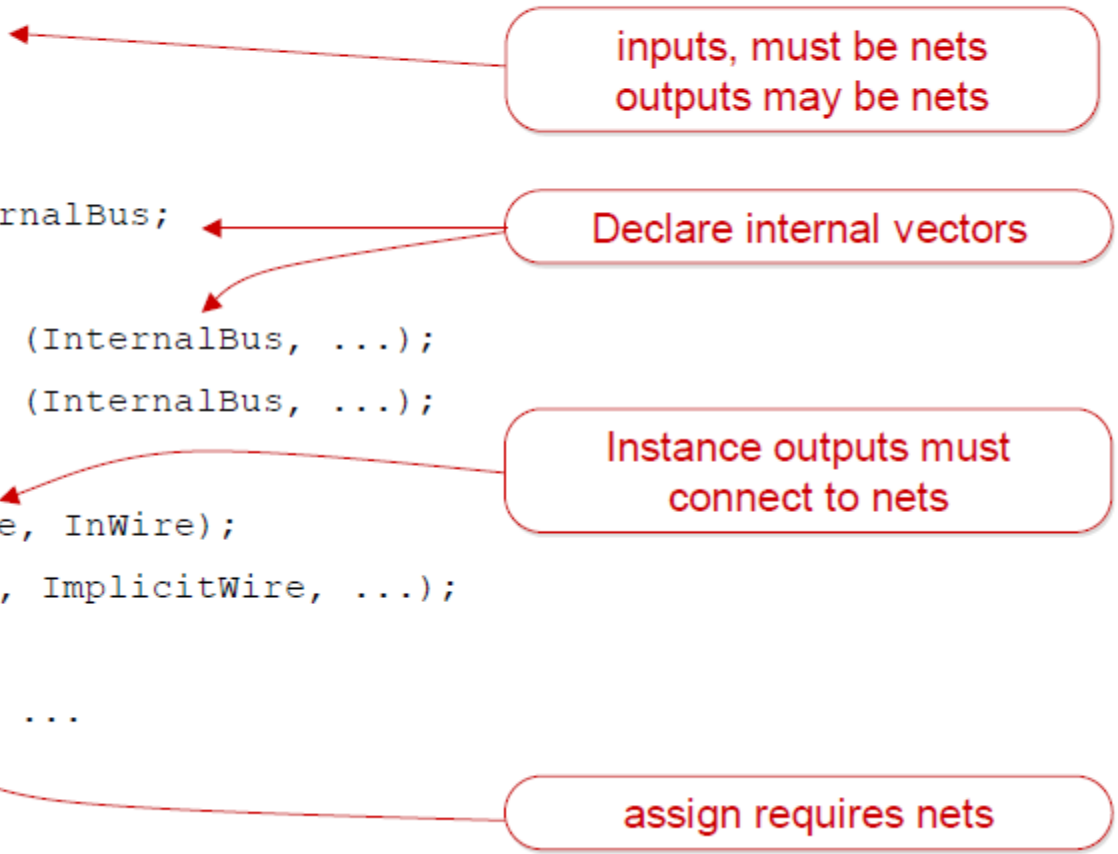
```
module UsesRegs (OutReg);
  output [7:0] OutReg;
  reg    [7:0] OutReg;

  reg R;

  initial
    R = ...

  always
    OutReg = 8'b...

endmodule
```

Declarations agree

Must be registers

Using Nets

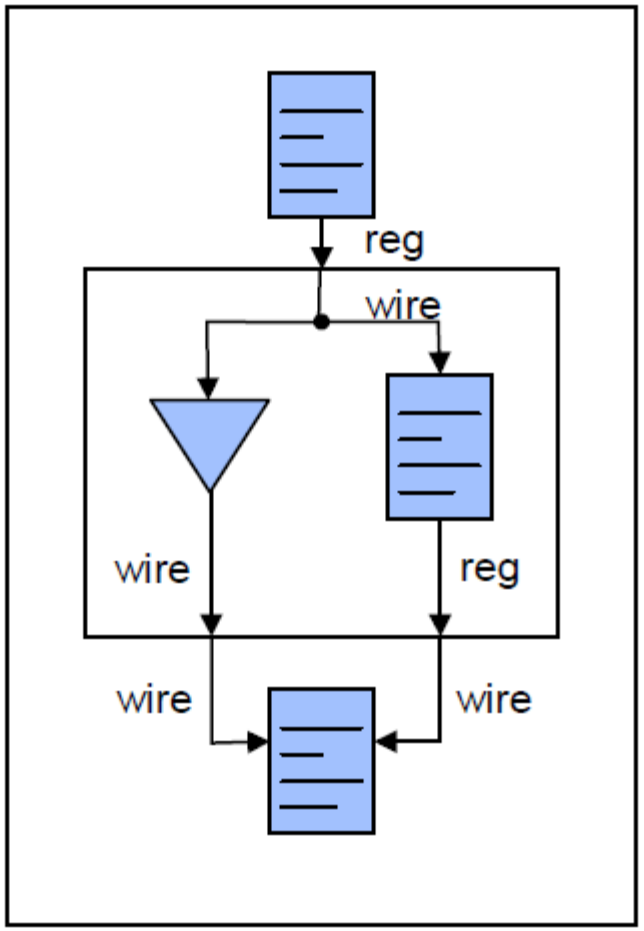
```
module UsesWires (InWire, OutWire);
  input  InWire;
  output OutWire;

  wire [15:0] InternalBus;

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

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

  assign OutWire = ...

endmodule
```



Module Boundaries

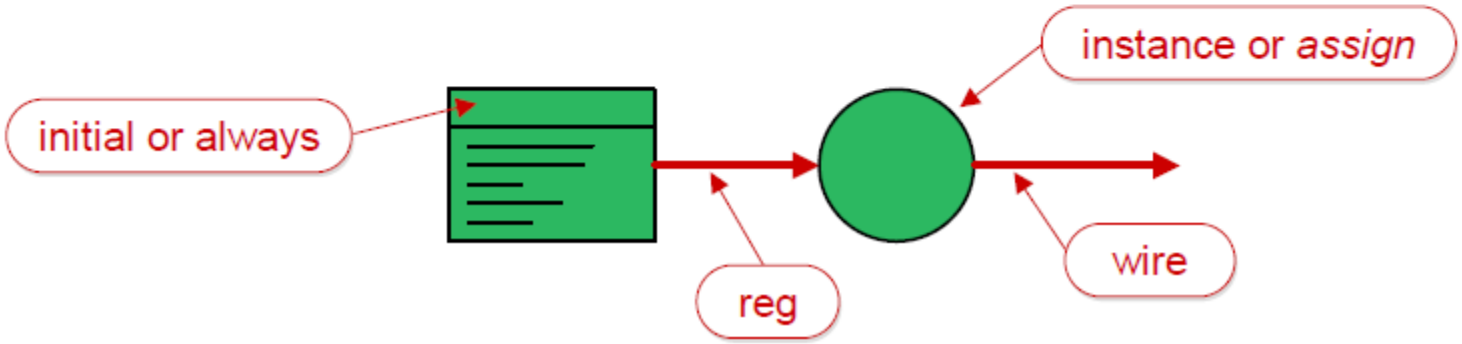
```
module Top;
   reg R;
   wire W1, W2;
   Silly S (.InWire(R), .OutWire(W1),
            .OutReg(W2));
   ...
endmodule
```

```
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

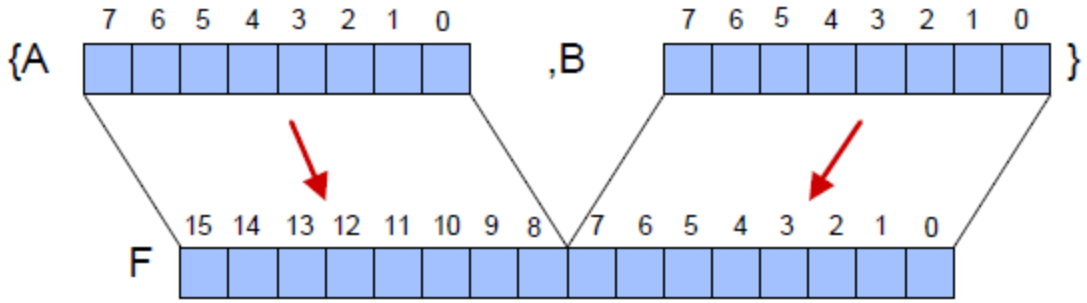
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus

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}}
```

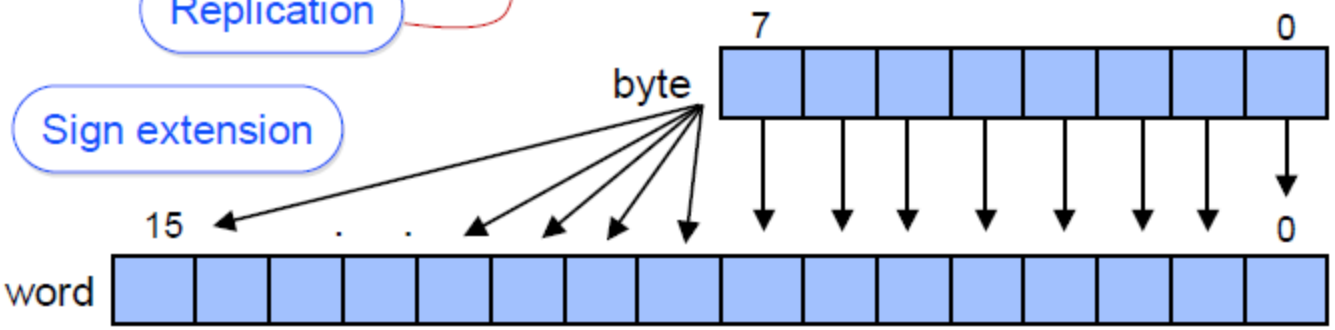
```
reg  [7:0] byte;
reg [15:0] word;
```

Concatenation

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

Replication

Sign extension



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