The Verilog Language

Bill Swartz

Dept. of EE
Univ. of Texas at Dallas

Design Process

- Idea
- Implement
- Test

- Manufacture

- Subject to iteration (Refinement)
- Hopefully converges

Implementation

- Prototype
- Abstraction to Physical

Prototype

- First implementation of physical
- Useful to be flexible
- See where real world differs from theoretical
- Modify as needed

Prototypes

- Breadboards
- Printed circuit boards
- Virtual Prototypes
  - Programmable Bread boards
  - Software programmable
  - Field Programmable Gate Array (FPGA)
  - Example: Baysys2 using Xilinx Spartan 3E FPGA

Virtual Prototype Implementation

- Software programmable hardware
- Hardware description languages used to describe prototype and its behavior

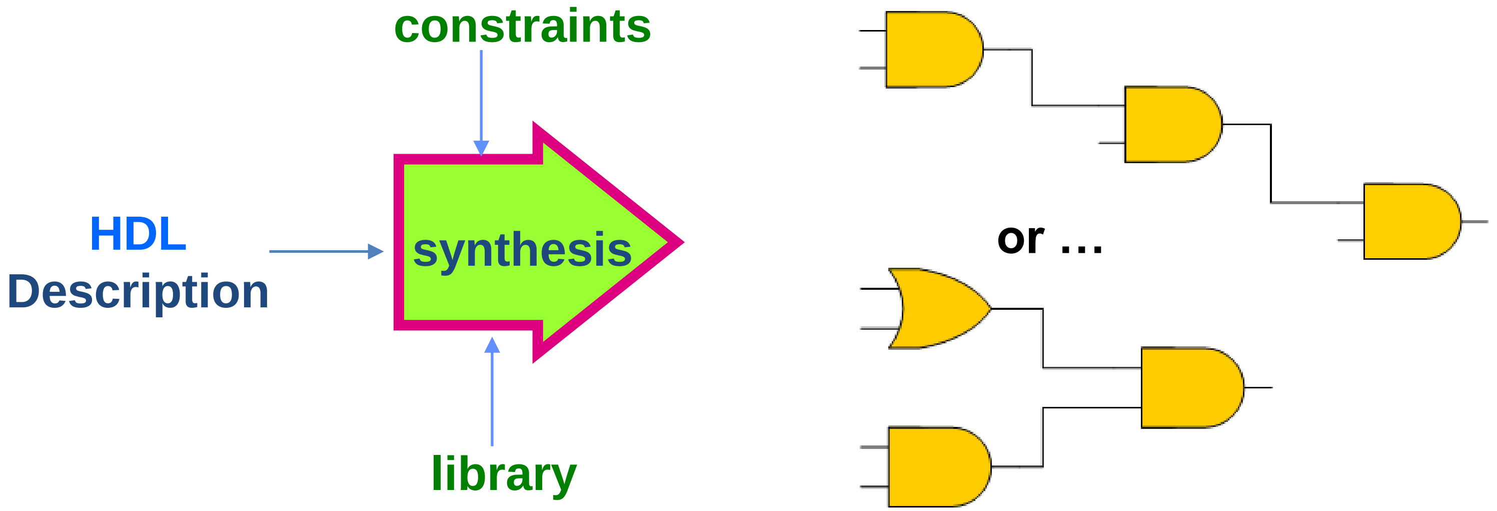
Hardware Modeling Using HDL

- **HDL: Hardware Description Language -** A high level programming language used to model hardware.

- Hardware Description Languages
  - have special hardware related constructs.
  - can be used to build models for **simulation**, **synthesis** and **test**
  - have been extended to the system design level
  - **VHDL: V**HSIC **H**ardware **D**escription **L**anguage
    - VHSIC – Very High Speed Integrated Circuit Program
    - Mostly used in academia
  - **Verilog** HDL
    - Mostly used in commercial electronics industry

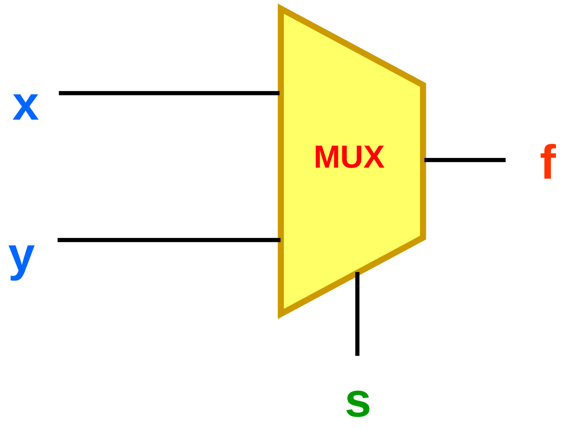
Concept of Synthesis

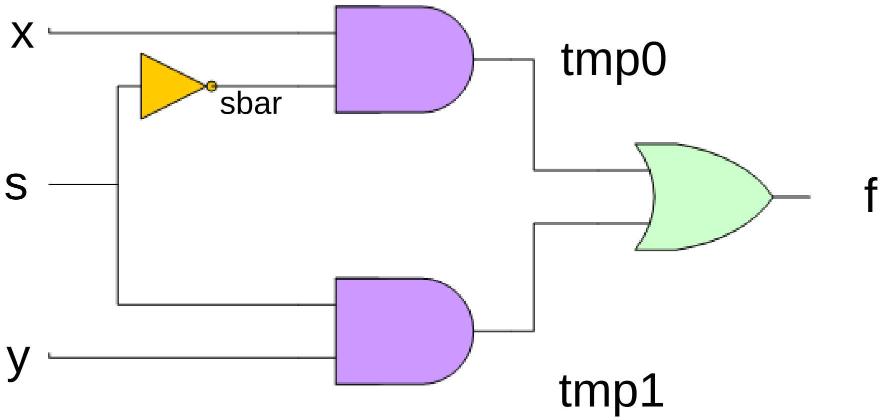
- Logic synthesis
  - A program that **“designs” logic from abstract descriptions** of the logic
    - takes constraints (e.g. size, speed)
    - uses a library (e.g. 3-input gates)
  - The aim of synthesis is to produce hardware which will do what the concurrent statements specify.
    - This includes processes as well as other concurrent statements.
- How?
  - You write an “abstract” HDL description of the logic
  - The synthesis tool provides alternative implementations



Goal

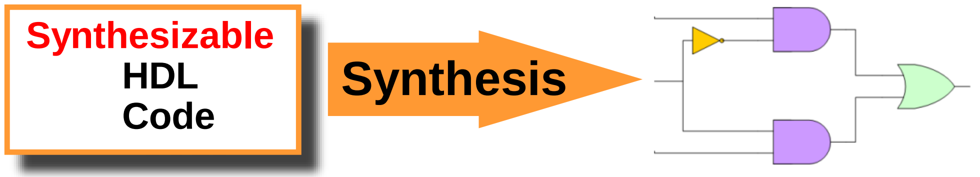
- We know the function we want, and can specify in C-like form.
  - … but we don't always know the exact gates (nor logic elements)…
  - … we want the tool to figure this out…

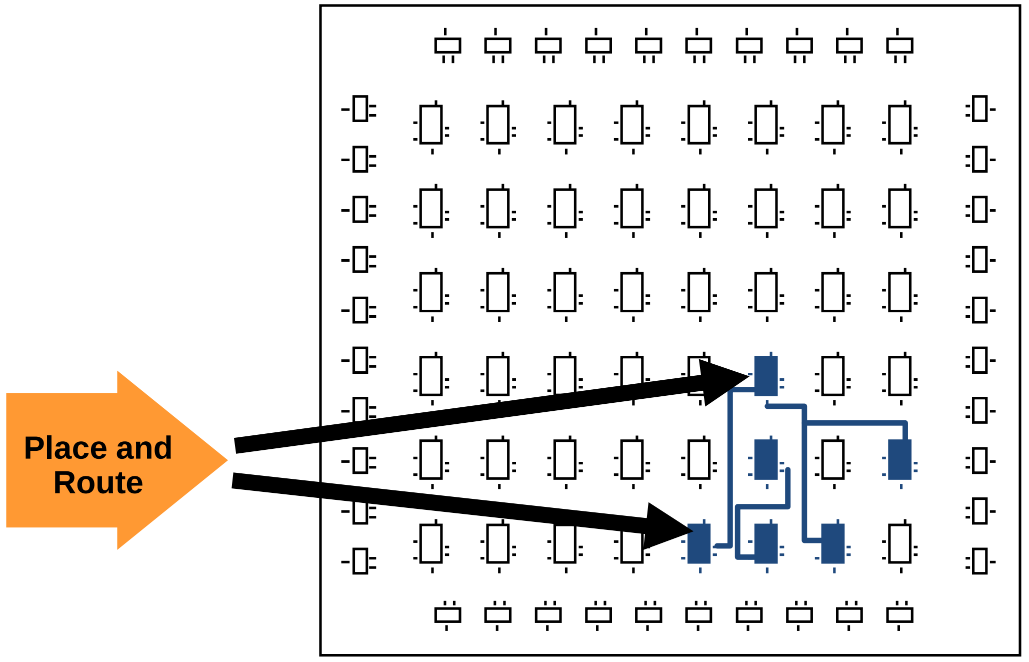




Importance of Synthesis

- In order to map an HDL code on a **FPGA**, **the code should be synthesizable!**
  - An HDL code that functions correctly in simulation, does not necessarily mean it is synthesizable.
  - Once you have gone through the synthesis tool for your HDL code **without errors**, your code is synthesizable.





Verilog

- **Created by Phil Moorby and Prabhu Goel of Gateway Design in 1984**
- **Gateway Design Automation was purchased by Cadence Design Systems in 1990.**
- **Originally a simulation language; later updated to support synthesis**
- **Verilog95**
  - **Cadence transferred Verilog into the public domain under the Open Verilog International (OVI)**
  - **Verilog was later submitted to IEEE and became IEEE Standard 1364-1995**
- **Verilog 2001**
  - **Extended to cover deficiencies in Verilog-95**
    - **Support for (2's complement) signed nets and variables.**
    - **Added generate/endgenerate construct (similar to VHDL's generate/endgenerate) File I/O has been improved by several new system tasks.**
    - **Syntax improvements (e.g. always @*, named parameter override, C-style function/task/module header declaration).**
    - **Verilog-2001 is the dominant flavor of Verilog supported by the majority of commercial EDA software packages.**
- **Verilog 2005**
  - ***Minor corrections, spec clarifications, and a few new language features (such as the uwire keyword).***

Useful Link

http://www.asic-world.com/verilog/veritut.html

Verilog Abstraction Levels

- Verilog supports designing at many different levels of abstraction. Three of them are very important:
- Gate Level
- Register-Transfer Level
- Behavioral level

Gate Level

- Described by logical entities and their timing properties.
- All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`).
- The usable operations are predefined logic primitives (AND, OR, NOT etc gates).
- Using gate level modeling might not be a good idea for any level of logic design.
- Gate level code is generated by tools like synthesis tools and this netlist is used for gate level simulation and for backend.
- It is the implementation of the design to be manufactured.

Register-Transfer Level

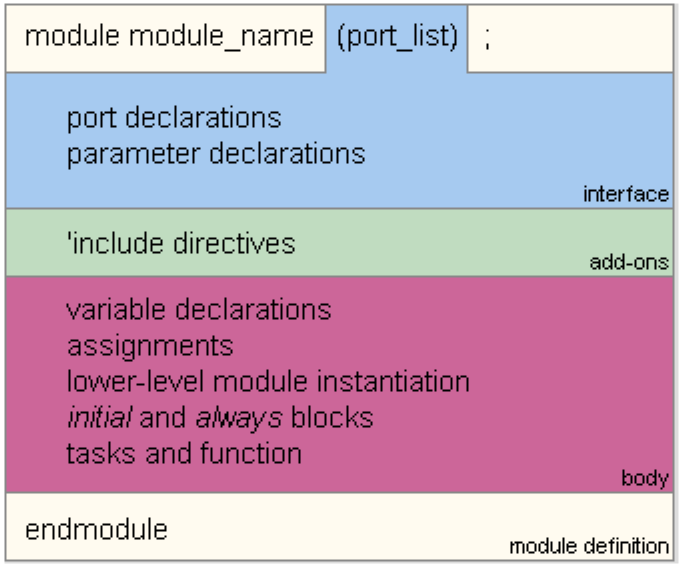
- Specify the characteristics of a circuit by operations and the transfer of data between the registers.
- An explicit clock is used. RTL design contains exact timing bounds: operations are scheduled to occur at certain times. Modern RTL code definition is "Any code that is synthesizable is called RTL code".

Behavioral Level

- This level describes a system by concurrent algorithms.
- Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other.
- Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realization of the design.

Definition of Module

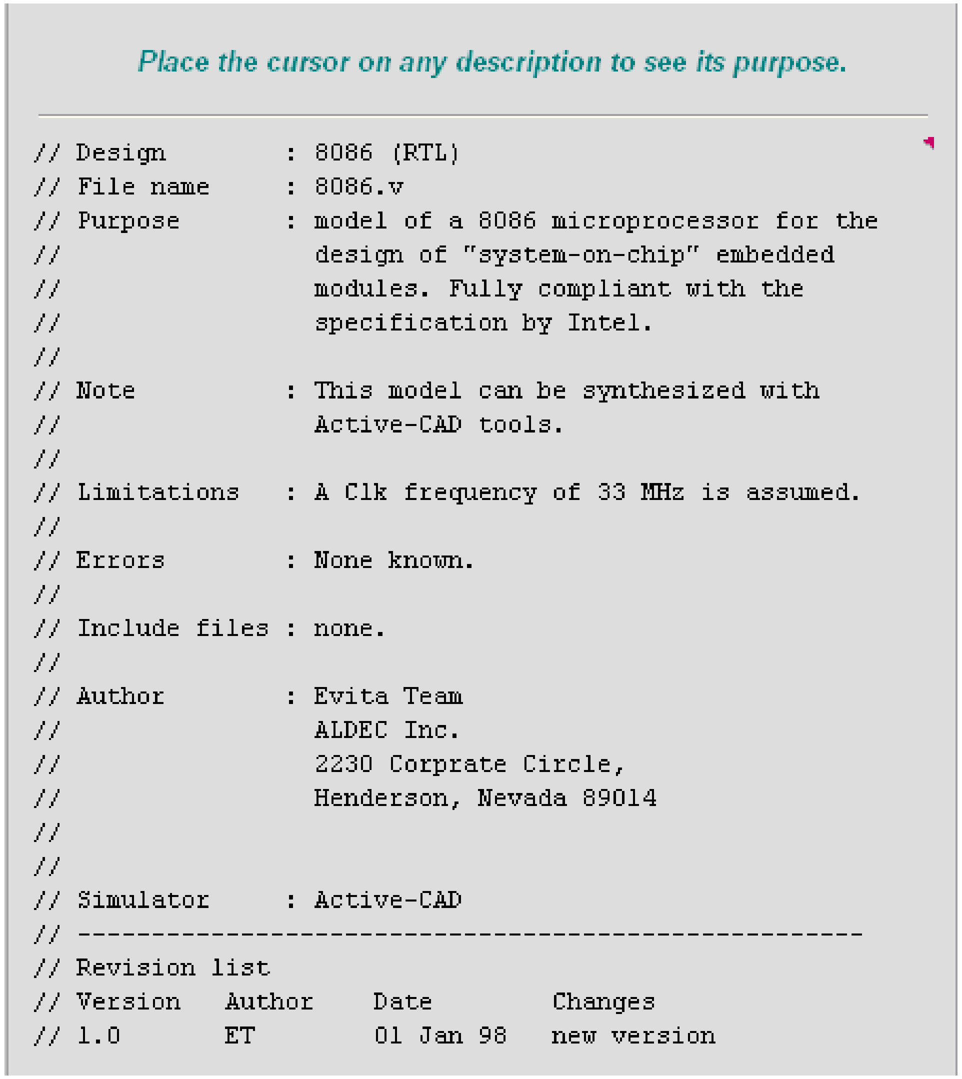
- Interface: port and parameter declaration
- Body: Internal part of module
- Add-ons (optional)



Some points to remember

- The name of Module
- Comments in Verilog
  - One line comment (// …………)
  - Block Comment (/*…………….*/)
- Description of Module (optional but suggested)

Description of Module



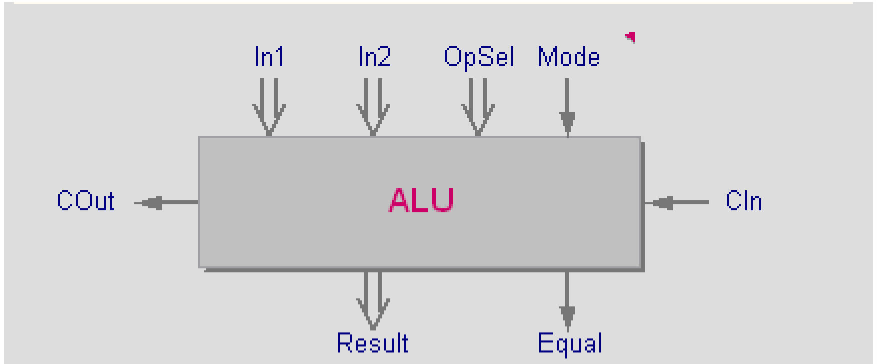
Place the cursor on any description to see its purpose.

```
// Design        : 8086 (RTL)
// File name     : 8086.v
// Purpose       : model of a 8086 microprocessor for the
//                 design of "system-on-chip" embedded
//                 modules. Fully compliant with the
//                 specification by Intel.
//
// Note          : This model can be synthesized with
//                 Active-CAD tools.
//
// Limitations   : A Clk frequency of 33 MHz is assumed.
//
// Errors        : None known.
//
// Include files : none.
//
// Author        : Evita Team
//                 ALDEC Inc.
//                 2230 Corprate Circle,
//                 Henderson, Nevada 89014
//
//
// Simulator     : Active-CAD
// ----------------------------------------------------
// Revision list
// Version    Author    Date        Changes
// 1.0        ET        01 Jan 98   new version
```

The Module Interface

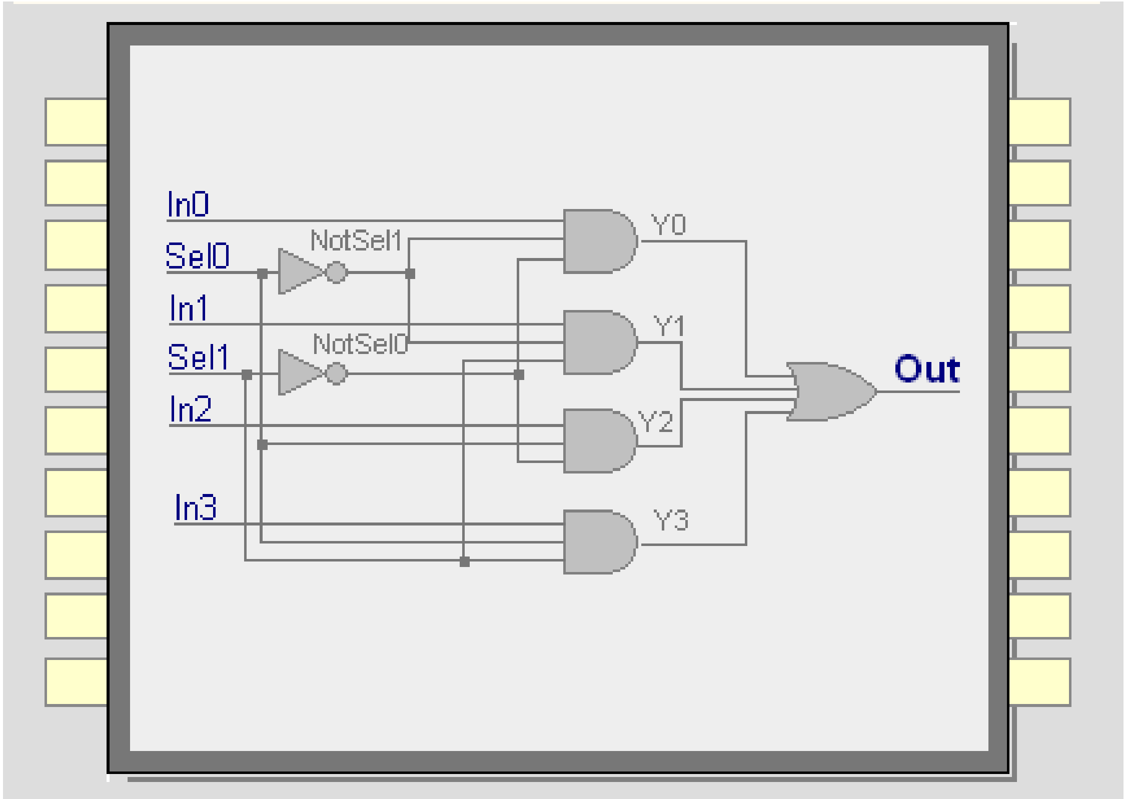
- Port List

- Port Declaration



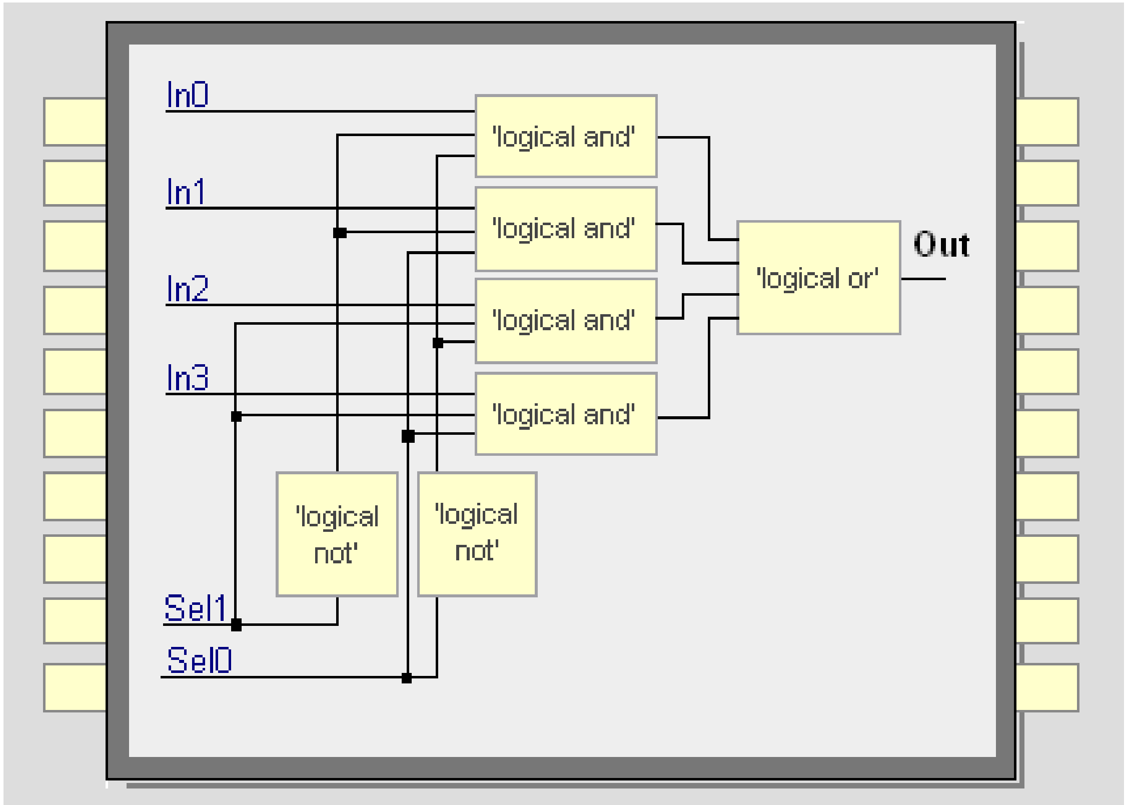
```
module ALU (Result, COut, Equal, In1, In2,
            OpSel, CIn, Mode);

output  [3:0] Result;   // operation result
output        COut;     // carry out
output        Equal;    // when 1, In1 = In2
input   [3:0] In1;      // first operand
input   [3:0] In2;      // second operand
input   [3:0] OpSel;    // operation select
input         CIn;      // carry in
input         Mode;     // mode arithm/logic;
                        // arithm when 0
 . . .

endmodule
```

One language, Many Coding Style



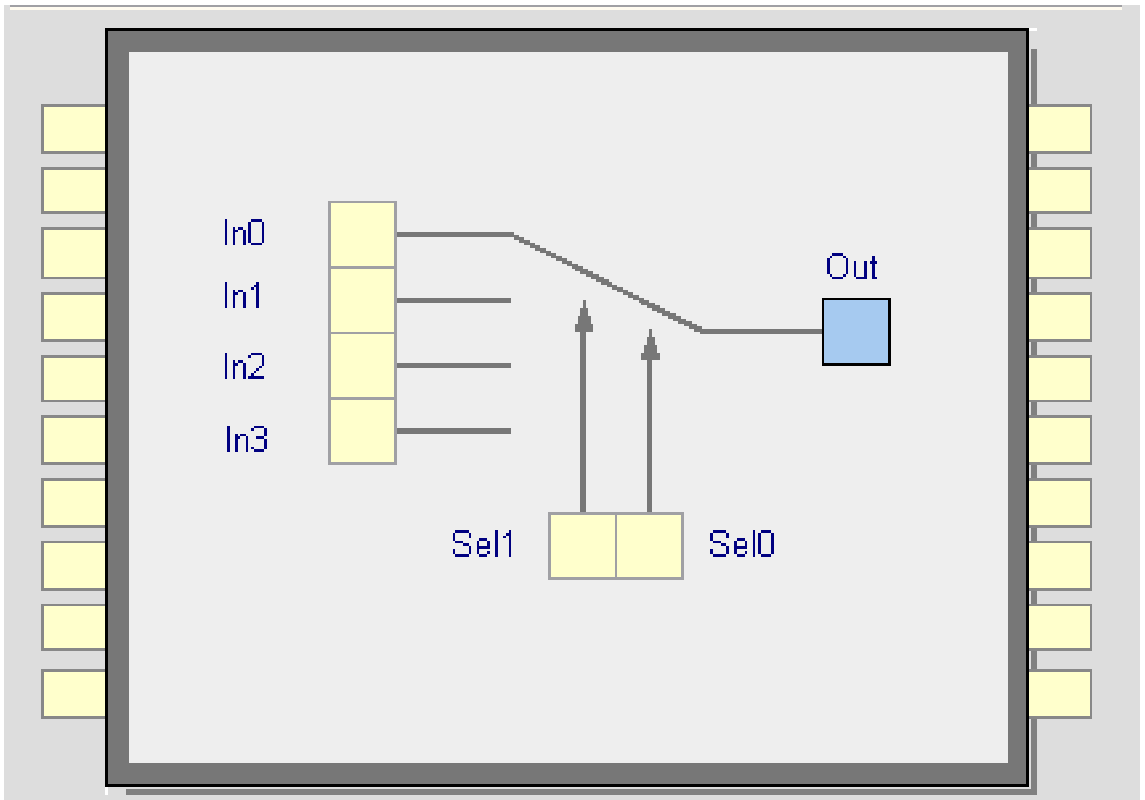
Structural Style. The circuit is specified in terms of instantiations of lower level components (in this case logic gates, which are Verilog primitives) connected with internal signals. The translation of such a specification into a physical circuit is straightforward.

One language, Many Coding Style (contd.)



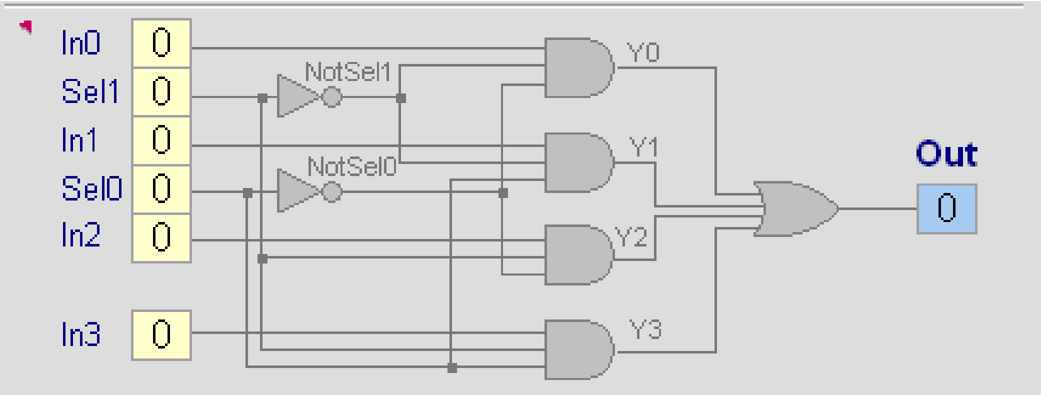
Dataflow Style. This style is similar to logical equations, although in general it is not limited to logical values only. The specification is comprised of expressions made up of input signals and assigned to outputs. In most cases, such an approach can be quite easily translated into a structure and then implemented.

One language, Many Coding Style (contd.)



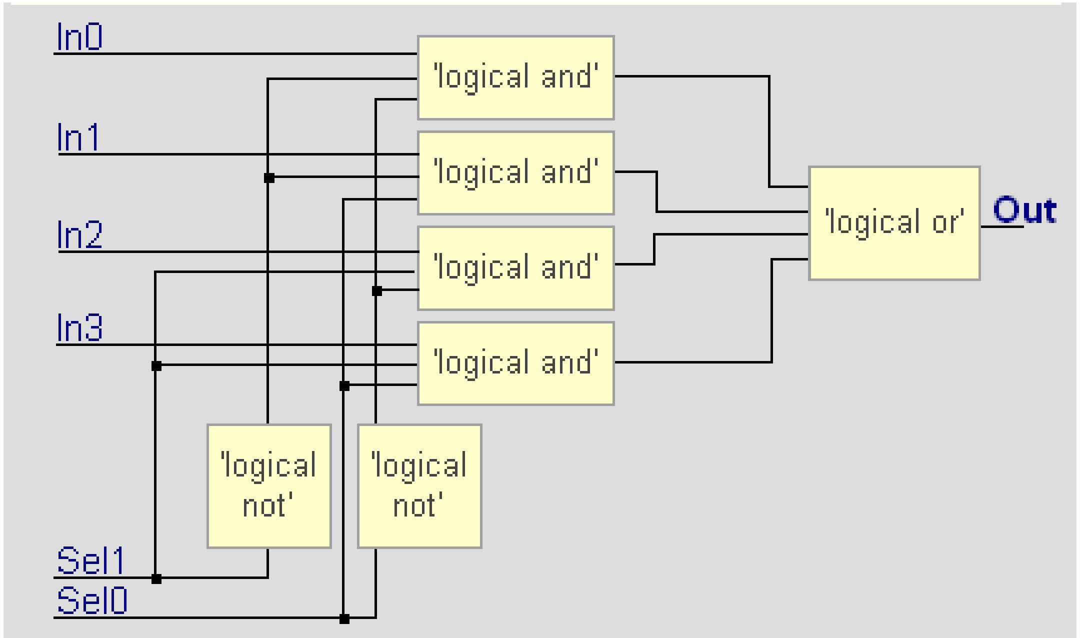
Behavioral Style. It specifies the circuit in terms of its expected behavior. It is the closest to a natural language description of the circuit functionality, but also the most difficult to synthesize.

Structural style: Verilog Code



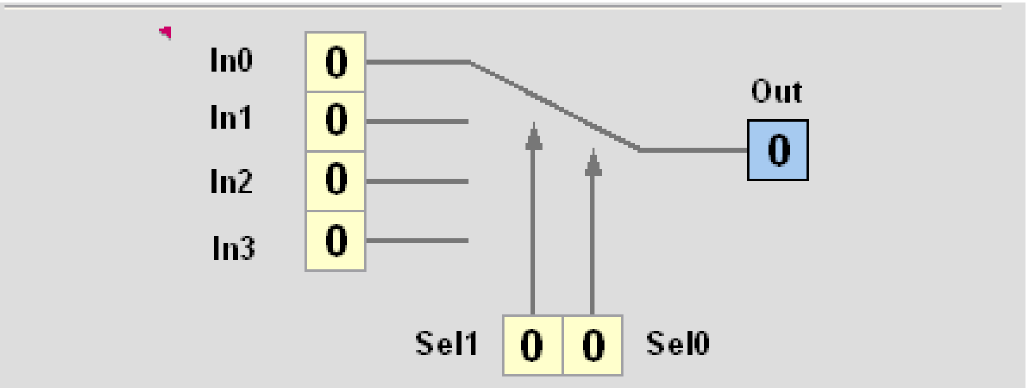
```
module mux_4_to_1 (Out,In0,In1,In2,In3,Sel1,Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;

~~wire NotSel0, NotSel1;~~
~~wire Y0, Y1, Y2, Y3;~~

not (NotSel0, Sel0);
not (NotSel1, Sel1);
and (Y0, In0, NotSel1, NotSel0);
and (Y1, In1, NotSel1, Sel0);
and (Y2, In2, Sel1, NotSel0);
and (Y3, In3, Sel1, Sel0);
or (Out, Y0, Y1, Y2, Y3);

endmodule
```

Dataflow style: Verilog Code



```
module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);

output Out;
input In0, In1, In2, In3, Sel0, Sel1;

assign Out = (~Sel1 & ~Sel0 & In0) | (~Sel1 & Sel0 & In1)
             | (Sel1 & ~Sel0 & In2)| (Sel1 & Sel0 & In3);

endmodule
```

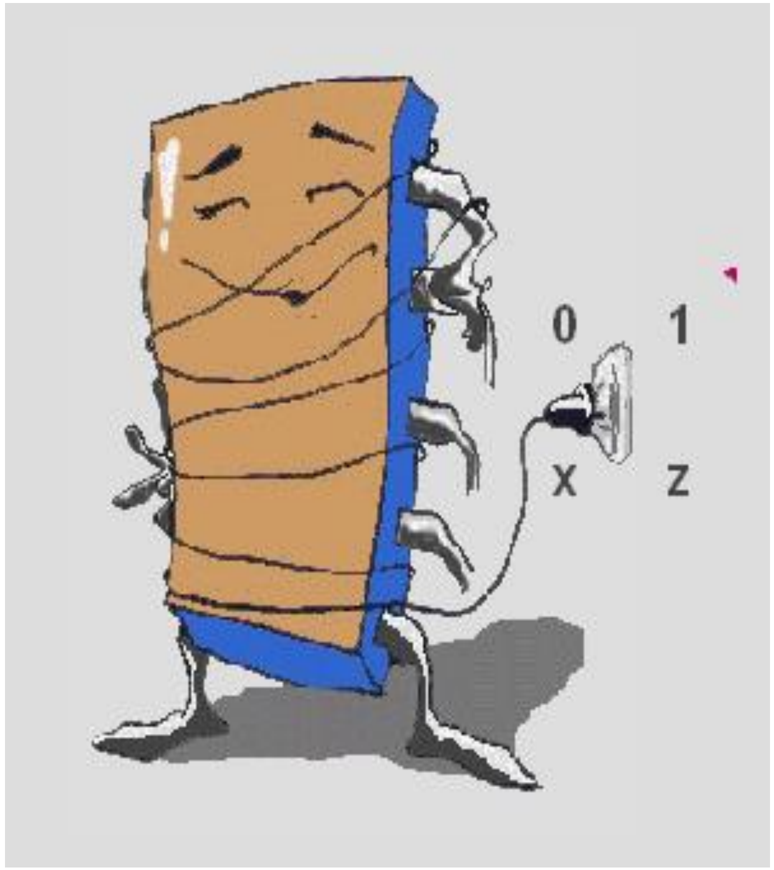
Behavioral style: Verilog Code



```
module mux_4_to_1 (Out,In0,In1,In2,In3,Sel1,Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;
reg Out;

always @(Sel1 or Sel0 or In0 or In1 or In2 or In3)
  begin
    case ({Sel1, Sel0})
    2'b00 : Out = In0;
    2'b01 : Out = In1;
    2'b10 : Out = In2;
    2'b11 : Out = In3;
    default : Out = 1'bx;
    endcase
  end

endmodule
```

Data Values and Representation

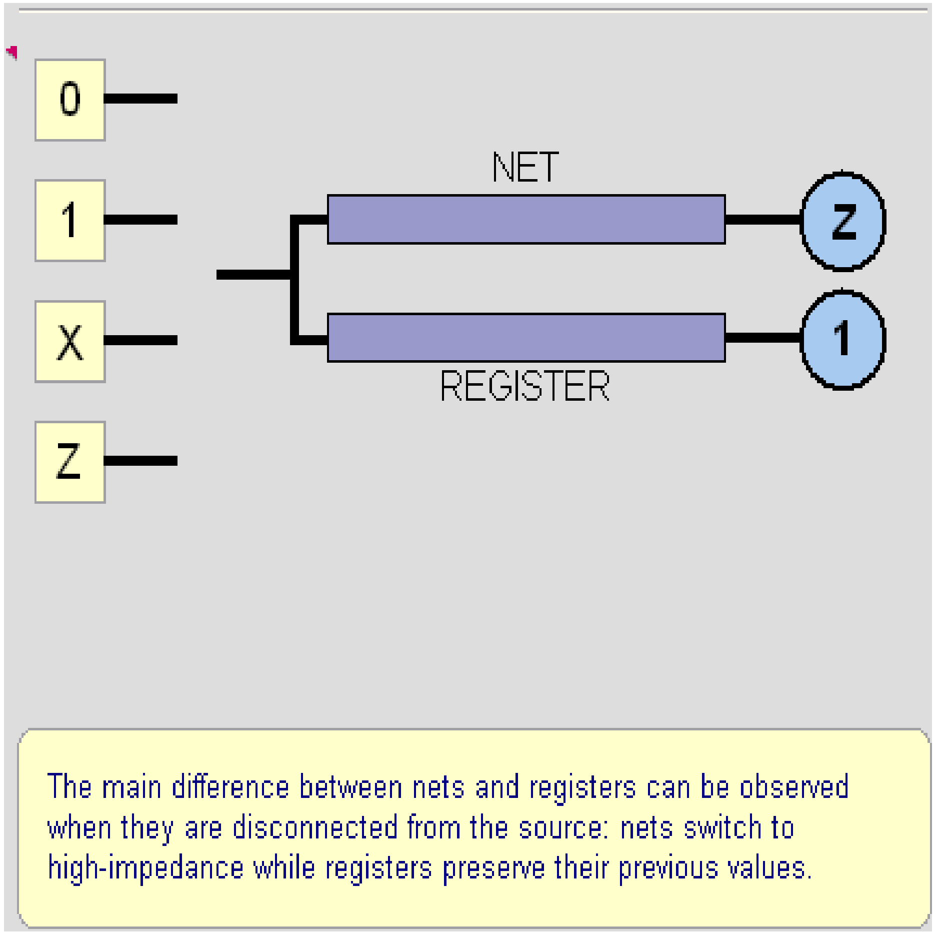
- Four Data value
- Data representation Type
  - Binary 6’b100101
  - Hex 6’h25



Class of Signals

- Nets: physical connection between hardware elements

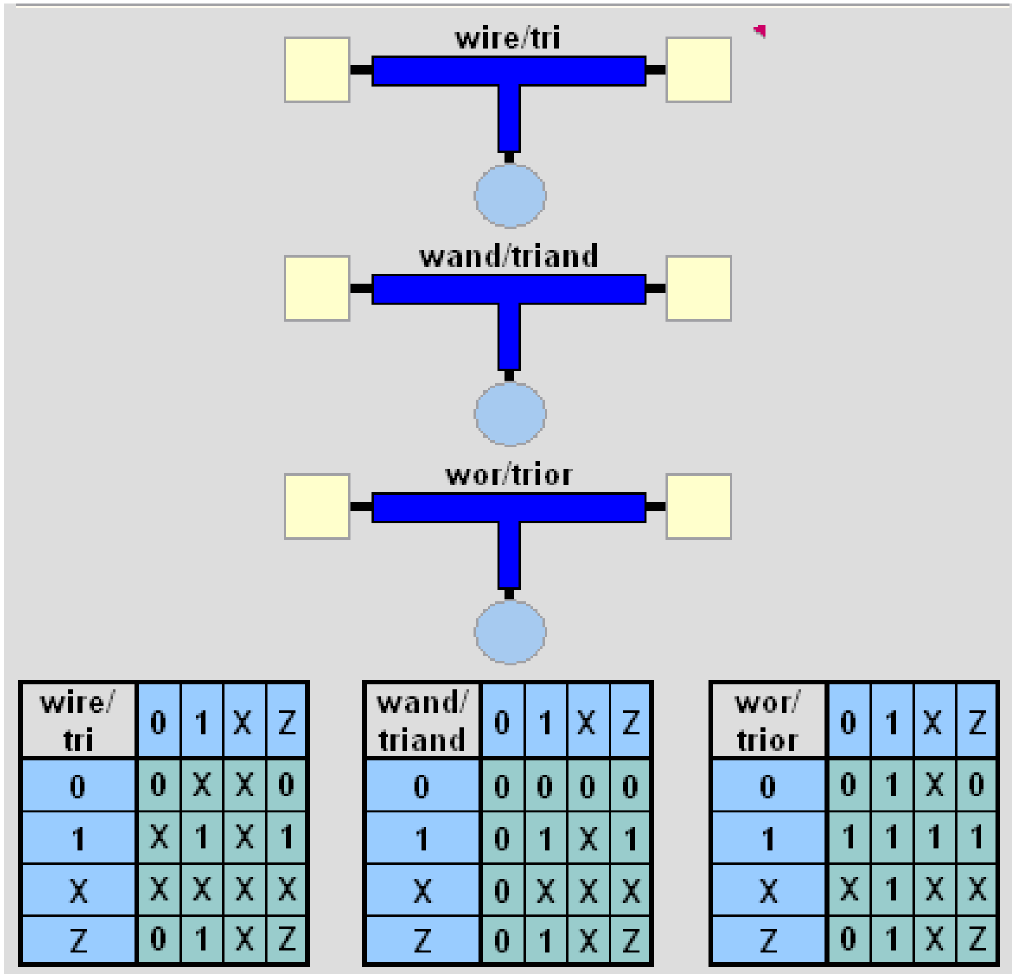
- Registers: Store value even if disconnected



The main difference between nets and registers can be observed when they are disconnected from the source: nets switch to high-impedance while registers preserve their previous values.

Nets

- wire/tri
- wand/triand
- wor/trior
- supply0,supply1
- tri0,tri1,trireg

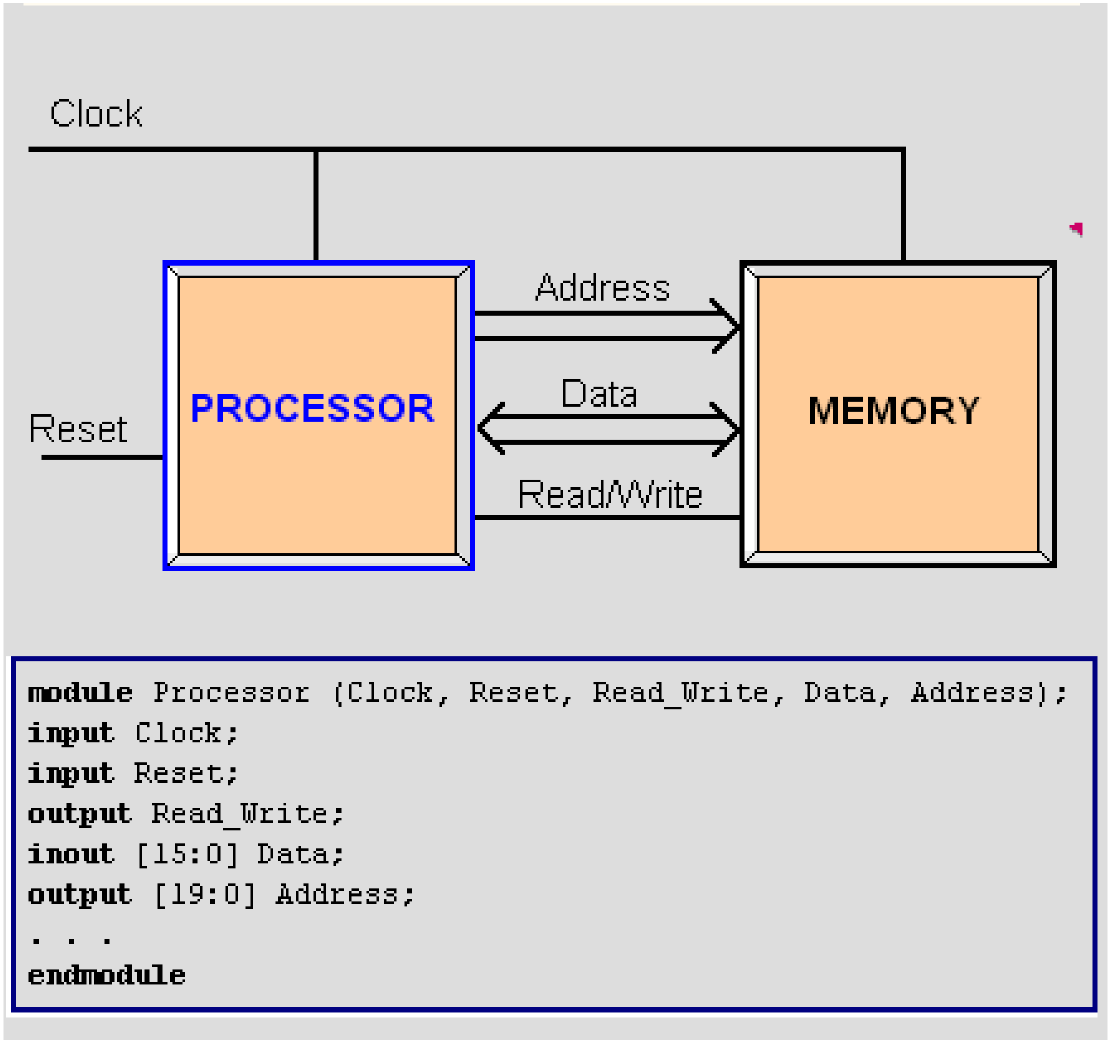


wire/ tri	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

wand/ triand	0	1	X	Z
0	0	0	0	0
1	0	1	X	1
X	0	X	X	X
Z	0	1	X	Z

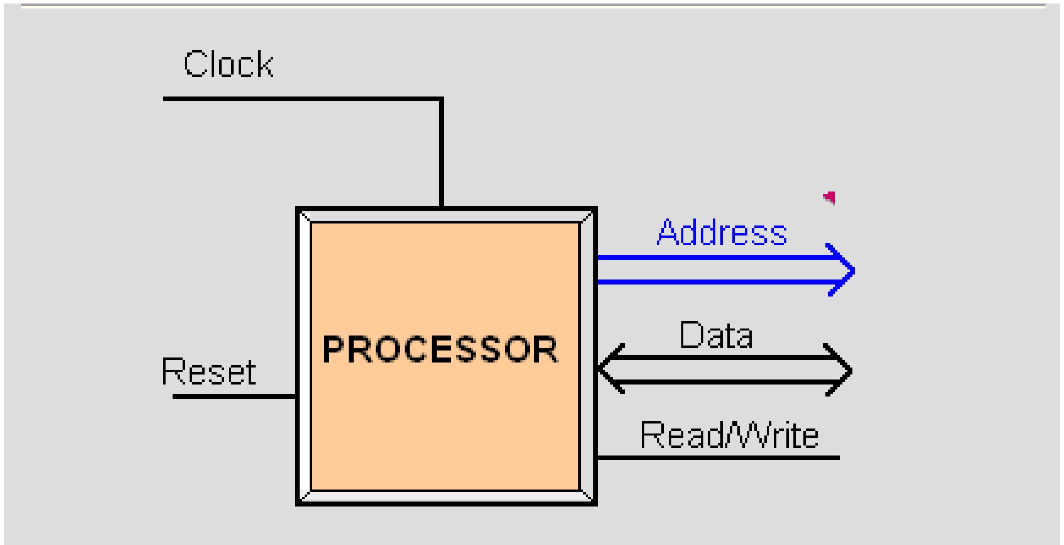
wor/ trior	0	1	X	Z
0	0	1	X	0
1	1	1	1	1
X	X	1	X	X
Z	0	1	X	Z

Specifications of Ports



```
module Processor (Clock, Reset, Read_Write, Data, Address);
input Clock;
input Reset;
output Read_Write;
inout [15:0] Data;
output [19:0] Address;
. . .
endmodule
```

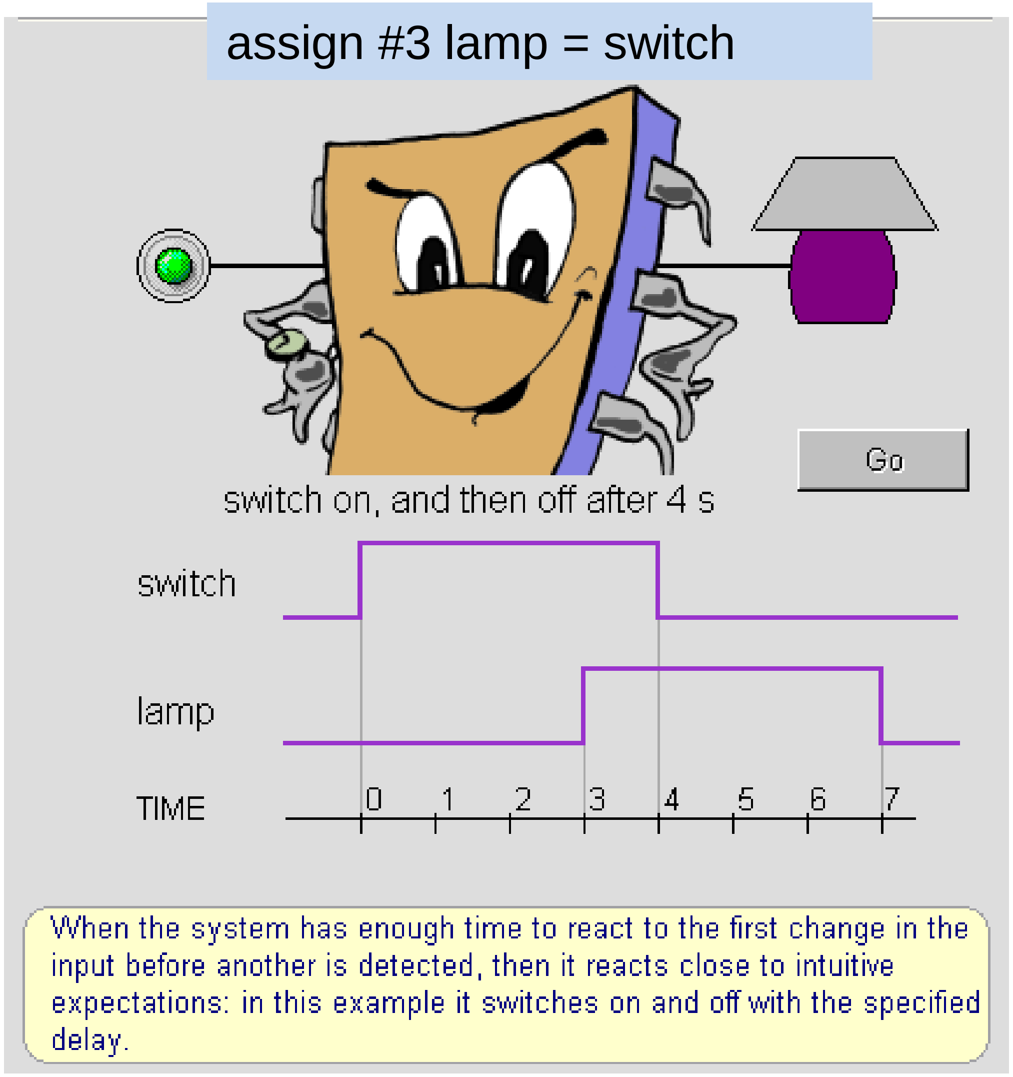
Registered Output



```
module Processor (Clock, Reset, Read_Write, Data, Address);
input Clock;
input Reset;
output Read_Write;
inout [15:0] Data;
output [19:0] Address;
. . .
reg [19:0] Address;
endmodule
```

Output ports can be type *register* by adding a new declaration for the port, stating that it is a register type.

Delay Statement



assign #3 lamp = switch

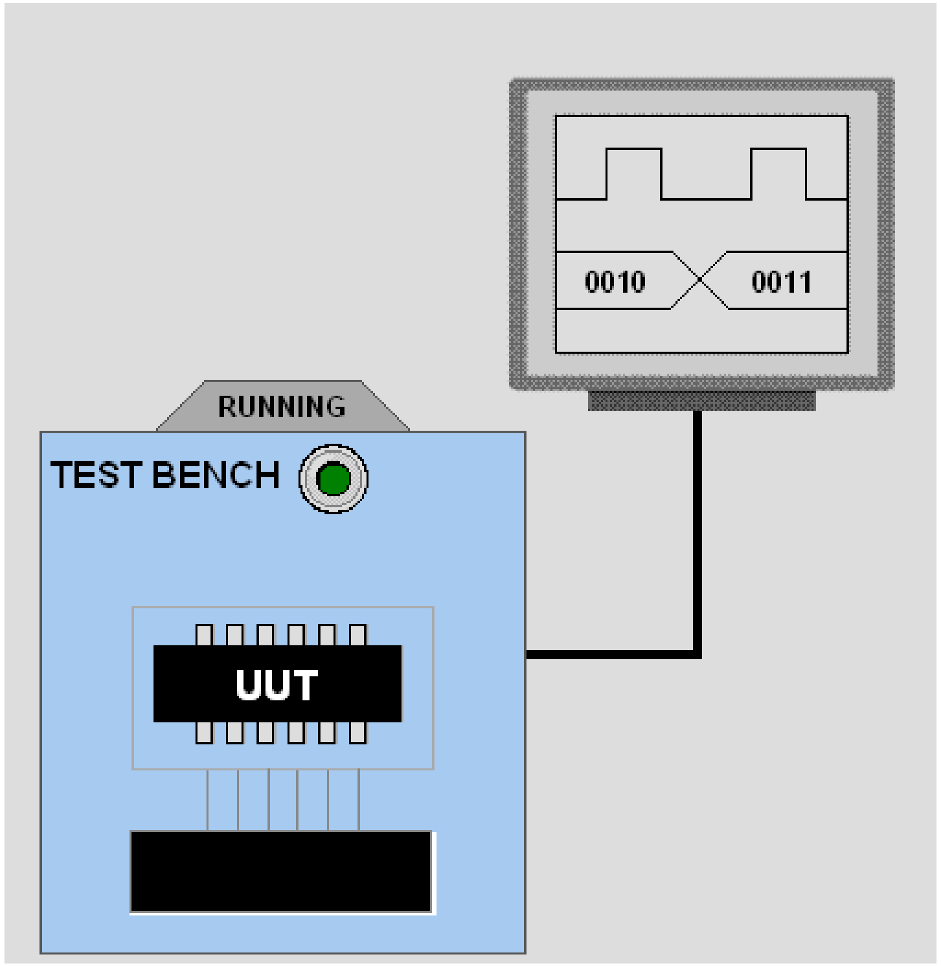
When the system has enough time to react to the first change in the input before another is detected, then it reacts close to intuitive expectations: in this example it switches on and off with the specified delay.

Parameter

```
reg [ 7 : 0 ] DataBus;
  . . .

for (cntr = 0; cntr < 8; cntr = cntr + 1)
    . . .

for (cntr = 7; cntr >= 0; cntr = cntr - 1)
    . . .
```

```
parameter BusSize = 8;
reg [Busize-1 : 0 ] DataBus;
  . . .

for (cntr=0; cntr<BusSize-1; cntr=cntr+1)
    . . .

for (cntr=BusSize; cntr>=0; cntr=cntr-1)
    . . .
```

Test Bench



```
module main;
reg  a, b, c;
wire sum, carry;

fulladder add(a,b,c,sum,carry);
initial
begin
   a = 0; b = 0; c = 0;
   #5;
   a = 0; b = 1; c = 0;
   #5;
   a = 1; b = 0; c = 1;
   #5;
   a = 1; b = 1; c = 1;
   #5
end
endmodule
```

A simple Verilog Example

- Verilog Code

```
// A simple example

module and2 (a, b ,c);

input a, b;

output c;

assign c = a & b;

endmodule
```

← comment line

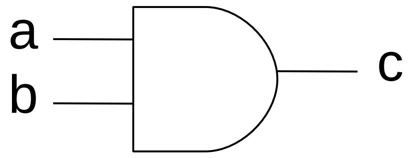
← module name
port list

← port declarations

← body

← end module

- Circuit

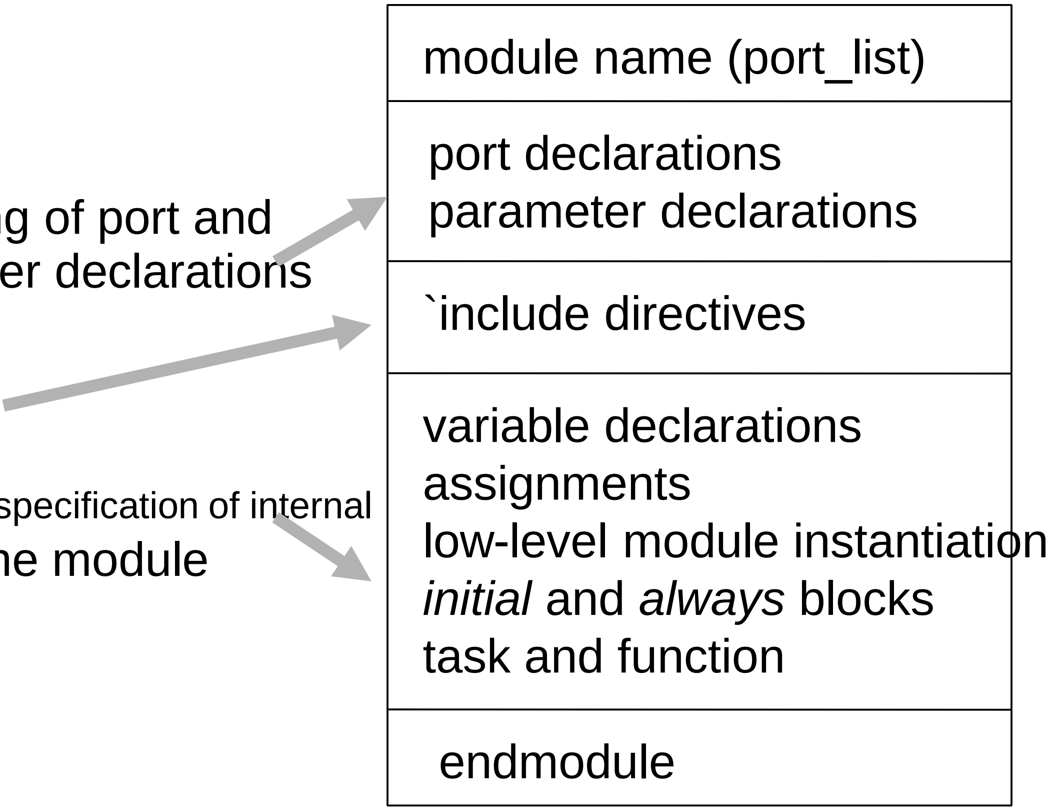


Many materials in this lecture are taken from ALDEC Verilog tutorial (www.aldec.com)

Module definition

- Modules are the basic building blocks in Verilog. A module definition starts with the keyword **module** ends with the keyword **endmodule**

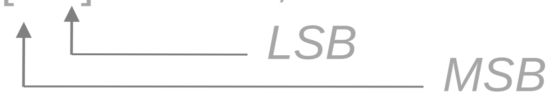
- Elements in a module
  - Interface: consisting of port and parameter declarations
  - optional add-ons
  - body: specification of internal part of the module

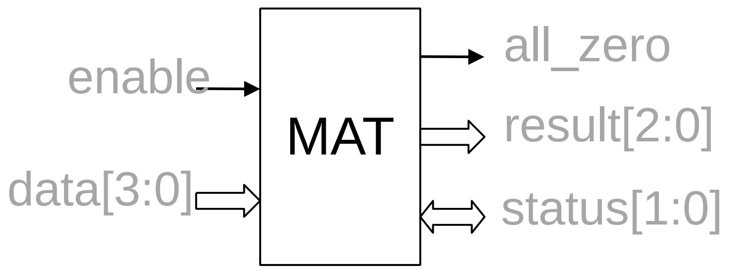


Port declaration

- Verilog Code
- Circuit

```
module MAT (enable, data, all_zero, result, status);
input           enable;     // scalar input
input    [3:0]  data;       // vector input
output          all_zero;   // scalar output
output  [3:0]   result;     // vector output
inout   [1:0]   status;     // bi-directional port
……
endmodule
```





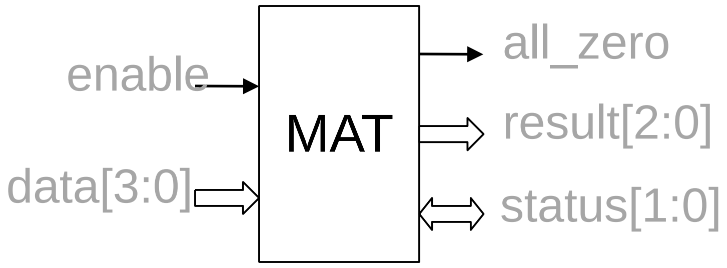
- To make code easy to read, use self-explanatory port names
- For the purpose of conciseness, use short port names
- In vector port declaration, MSB can be smaller index.
  e.g. output [0:3] result (result[0] is the MSB)

Port declaration

- Verilog Code
- Circuit

```
module MAT (enable, data, all_zero, result, status);
input           enable;     // scalar input
input    [3:0]  data;       // vector input
output          all_zero;   // scalar output
output   [3:0]  result;     // vector output
inout    [1:0]  status;     // bi-directional port
……
endmodule
```

In `[1:0]`: 1 is the MSB, 0 is the LSB.



- To make code easy to read, use self-explanatory port names
- For the purpose of conciseness, use short port names
- In vector port declaration, MSB can be smaller index.
  e.g. output [0:3] result (result[0] is the MSB)

Vectors

```
input    [3:0]   data ;
```

is a short hand way of writing

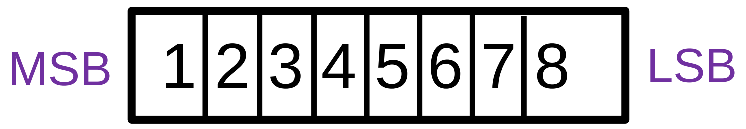
```
input data[3], input data[2], input data[1], input data[0] ;
```

- Vectors are a shortcut for writing a set of related signals or wires
- Order makes a difference [3:0] is not the same as [0:3]
- Vectors are sometimes called arrays or buses
- Prefer using vectors rather than buses because bus can also mean a bidirectional wire with many tri-stated devices
- May be connected as a bundle of related wires

Endianness

Suppose we had the hex number 0x12345678

This is a 32 bit number as shown as stored as



- This is how we would look at it in the Western Arabic number system
- The most significant bit is to the left
- Here each box is a nibble or 4 binary bits

Expanded into binary it would look like this:

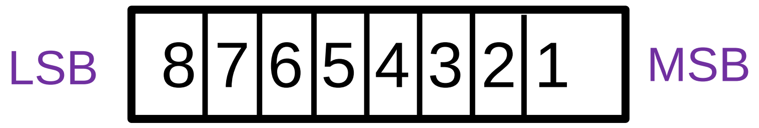
MSB 0001 0010 0011 0100 0101 0110 0111 1000

- We call this “big” endian
- In Verilog, we write
    - reg [31:0] my_register ; // 0 - based
    - reg [32:1] my_register ; // 1 - based

Endianness

Again suppose we had the hex number 0x12345678

What if we store this 32 bit number as shown



- The direction order is an arbitrary decision
- Here the most significant bit is to the right
- Again each box is a nibble or 4 binary bits
- Seems strange to put LSB first

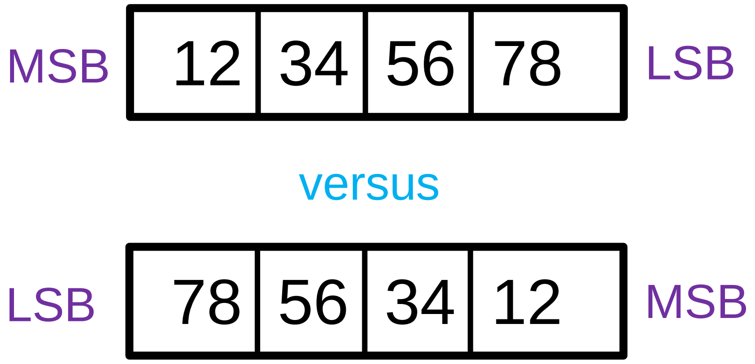
Expanded into binary it would look like this:

1000 0111 0110 0101 0110 0011 0010 0001   MSB

- We call this “little” endian
- In Verilog, we write
  - reg [0:31] my_register ; // 0 - based
  - reg [1:32] my_register ; // 1 - based

Endianness

- Choosing big or little endian does not matter if you are consistent
- In computer architecture, endianness occurs by convention at the byte level not the nibble or bit level
- This is a source of confusion



- In Verilog, endianness can occur at the bit level as shown previously

Port declaration

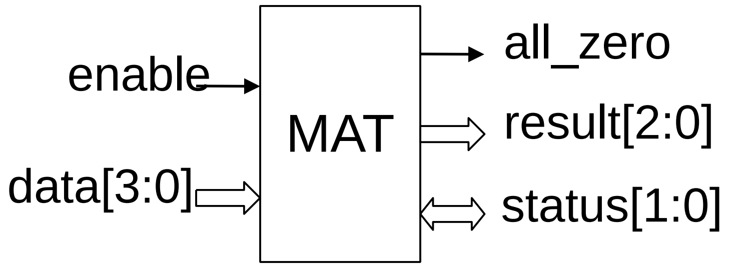
- Verilog Code

```
module MAT (enable, data, all_zero, result, status);
input           enable;      // scalar input
input    [3:0]  data;        // vector input
output          all_zero;    // scalar output
output  [3:0]   result;      // vector output
inout   [1:0]   status;      // bi-directional port
……
endmodule
```

LSB

MSB

- Circuit



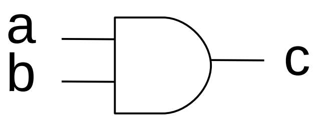
- To make code easy to read, use self-explanatory port names
- For the purpose of conciseness, use short port names
- In vector port declaration, MSB can be smaller index.
  e.g. output [0:3] result (result[0] is the MSB)

Available signal values

- Four signal values

—	1	True
—	0	False
—	X	Unknown
—	Z	High impedance

- Logic operations on four-value signals



Truth table

AND	1	0	X	Z
1	1	0	X	X
0	0	0	0	0
X	X	0	X	X
Z	X	0	X	X

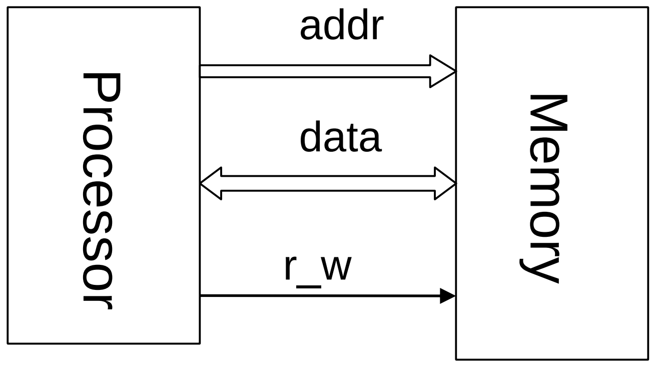
Signal Classification

- Each signal in Verilog belongs to either a net or a register
- A net represents a physical wire. Its signal value is determined by its driver. If it is not driven by any driver, its value is high impedance (Z).
- A register is like a variable in programming languages. It keeps its value until a new value is assigned to it.
- Unlike registers, nets do not have storage capacity.

Net declaration

- A net declaration starts with keyword **wire**

……
~~wire r_w~~; // scalar signal
wire [7:0] data; // vector signal
wire [9:0] addr; // vector signal
……



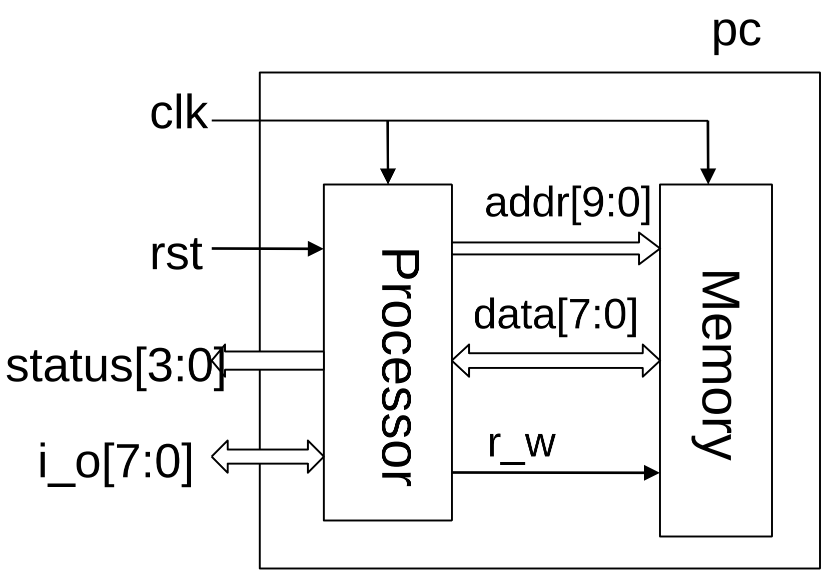
  - Selecting a single bit or a portion of vector signals
    - data[2]   single bit
    - data [5:3]   3 bits

- Other keywords that can be used to declare nets are: **tri, wand, triand, wor, trior, supply0, supply1, tri0, tri1, trireg**

Nets *v.s.* Ports

- Nets are internal signals that cannot be accessed by outside environment
- Ports are external signals to interface with outside environment
  - input ports can be read but cannot be written
  - output ports can be written but cannot be read
  - inout ports can be read and written

```
module pc (clk, rst, status, i_o);
input clk, rst;
output [3:0] status;
inout [7:0] i_o;
~~wire          r_w;~~
wire  [7:0]   data;
wire  [9:0]   addr;
……
endmodule
```



Register declaration

- A register declaration starts with keyword **reg**

```
……
reg             done;      // scalar signal
reg  [7:0]       count;     // vector signal
……
```

- Registers can be used to describe the behavior of sequential circuits

- Registers can also be used to implemented registered output ports

```
module pc (clk, rst, status, i_o);
input clk, rst;
output [3:0] status;
reg     [3:0] status;
inout [7:0] i_o;
……
```

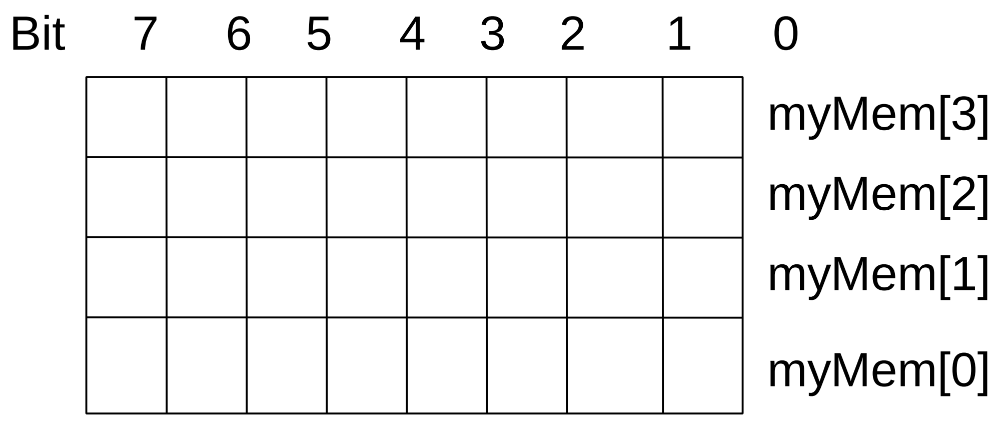
Better yet:

```
output reg [3:0] status ;
```

Defining memory

- A memory component can be defined using reg variables
- Example:

```
......
reg [7:0]  myMem [3:0];     // It defines a memory with 4 locations and each
                             //  location contains an 8-bit data
```



Memory Operation

```
reg [31:0] register_file [0:7];
wire [31:0] rf_bus;
wire r2b4;
   assign rf_bus = register_file [2];
   assign r2b4 = rf_bus[4];
```

Can’t use register_file[2][4] for assigning value to variable r2b4

No 3 dimensional arrays

3D RAM has not been invented yet

Using parameters

- The use of parameters make code easy to read and modify
- Example:

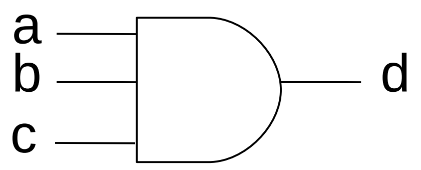
```
……
parameter  bussize = 8;
reg  [bussize-1 : 0]  databus1;
reg  [bussize-1 : 0]  databus2;
……
```

Predefined gate primitives

- Verilog offers predefined gate primitives
  - Multiple-input gates: **and, nand, or, nor, xor, xnor**

    e.g.

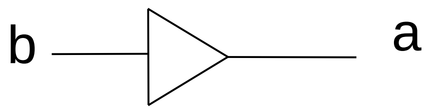
    and (d, a, b, c)



  - Multiple-output gates: **buf, not**

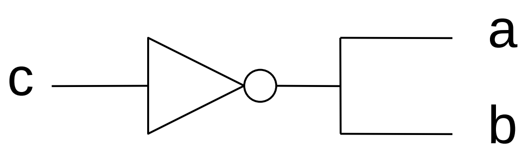
    e.g.

    buf (a, b)



    e.g.

    not (a, b, c)



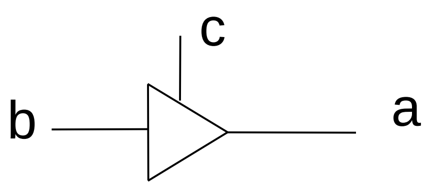
Two outputs???
Never used!

Predefined gate primitives

- tri-state gates: **bufif1, bufif0, notif1, notif0**

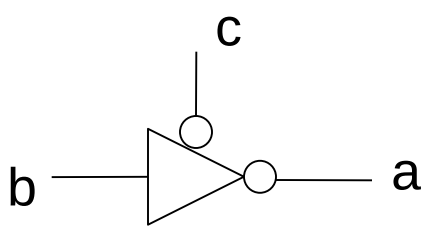
  e.g.

  bufif1 (a, b, c)



  e.g.

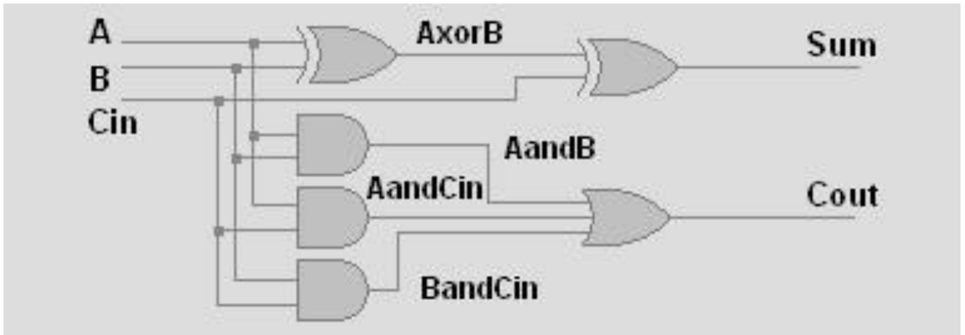
  notif0 (a, b, c)



- Verilog also offers two other gates (pull gates)

Example of structural Verilog code

- Example of using predefined gate primitives

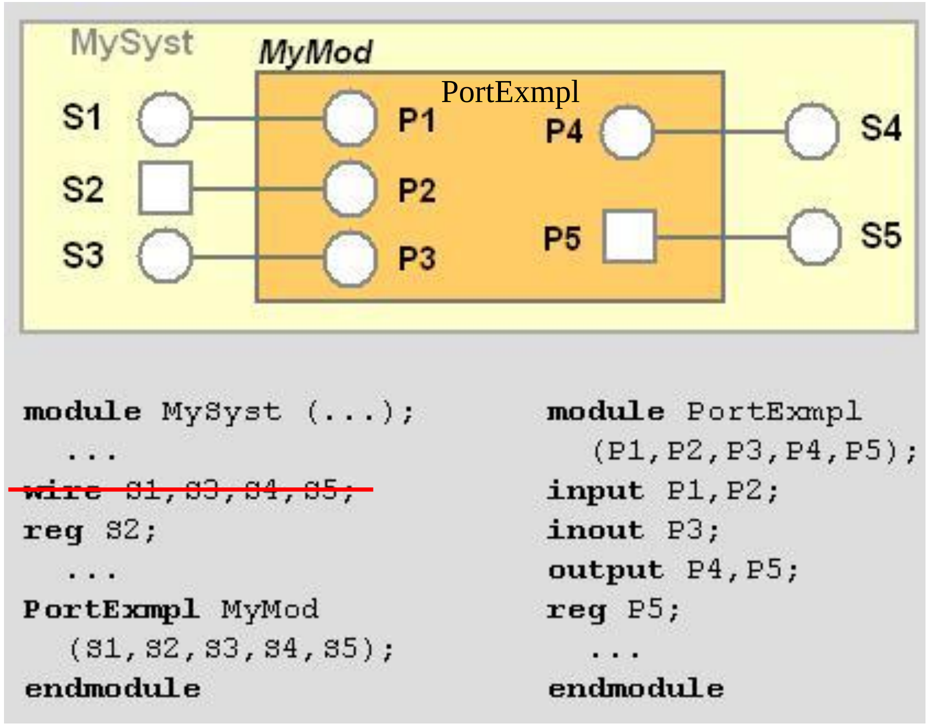


```
module FullAdd_1Bit (Sum,Cout,A,B, Cin);
output Sum, Cout;
input A, B, Cin;
~~wire AxorB, AandB, AandCin, BandCin;~~

xor #5 HalfSum (AxorB, A, B);
and #4 (AandB, A, B);
xor FullSum (Sum, AxorB, Cin);
and #4 And_2 (AandCin, A, Cin);
and #4 And_3 (BandCin, B, Cin);
or (Cout, AandB, AandCin, BandCin);

endmodule
```

(from ALDEC tutorial)

Module instantiation

- Module instantiation leads to hierarchy design



```
module MySyst (...);
   ...
~~wire S1,S3,S4,S5;~~
reg S2;
   ...
PortExmpl MyMod
   (S1,S2,S3,S4,S5);
endmodule
```

```
module PortExmpl
   (P1,P2,P3,P4,P5);
input P1,P2;
inout P3;
output P4,P5;
reg P5;
   ...
endmodule
```

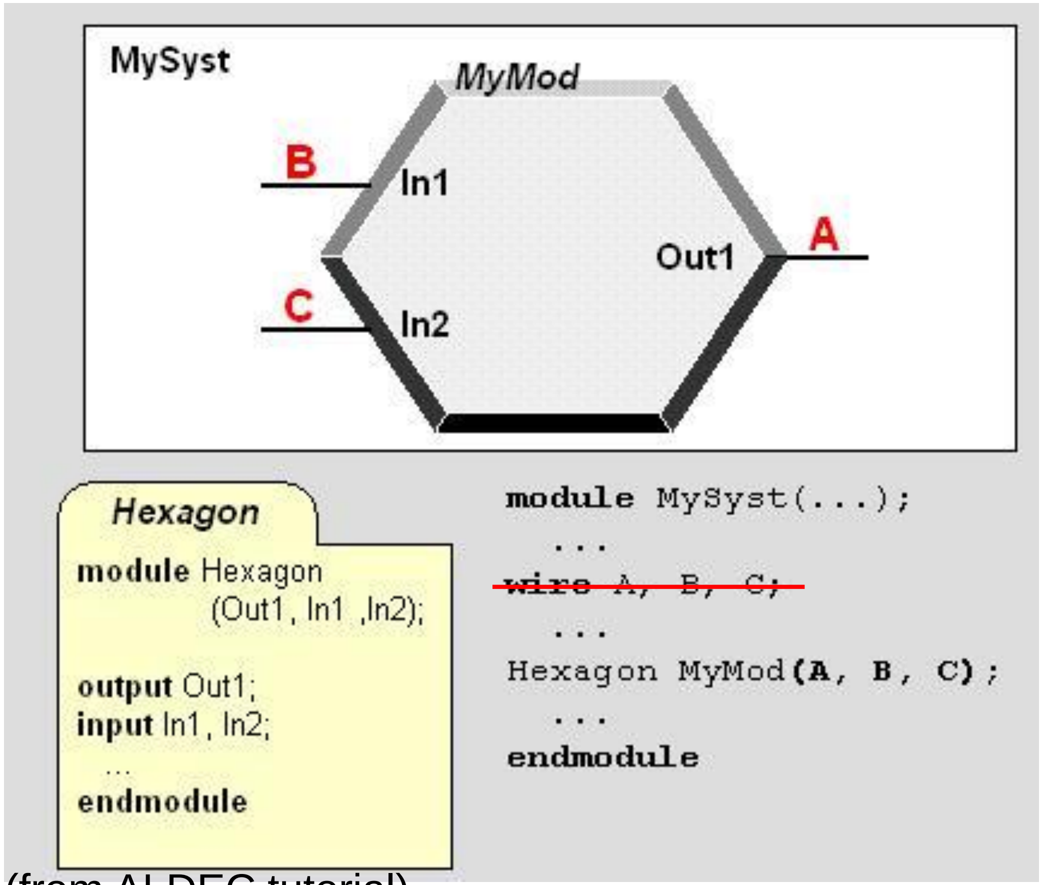
(from ALDEC tutorial)

- Port connecting rules

input	⟷	net or reg
inout	⟷	net
output	⟷	net

Module instantiation

- Signal assignment following port list order



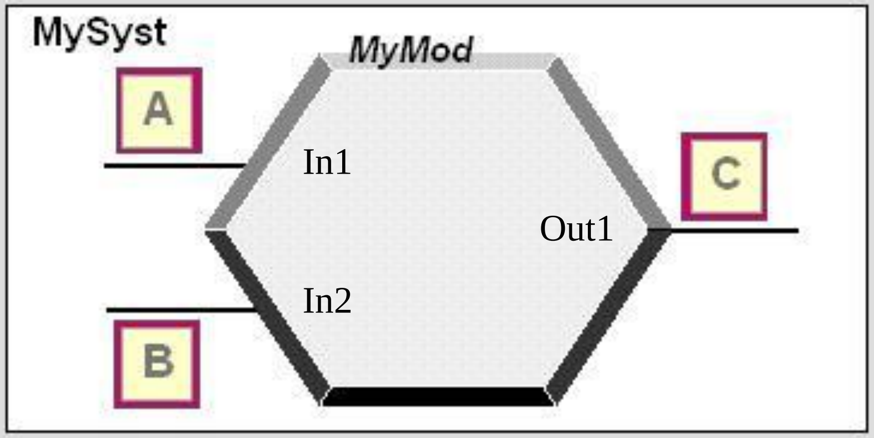
```
module Hexagon
        (Out1, In1 ,In2);

output Out1;
input In1, In2;
  ...
endmodule
```

```
module MySyst(...);
  ...
~~wire A, B, C;~~
  ...
Hexagon MyMod(A, B, C);
  ...
endmodule
```

(from ALDEC tutorial)

Module instantiation

- Signal assignment by port names



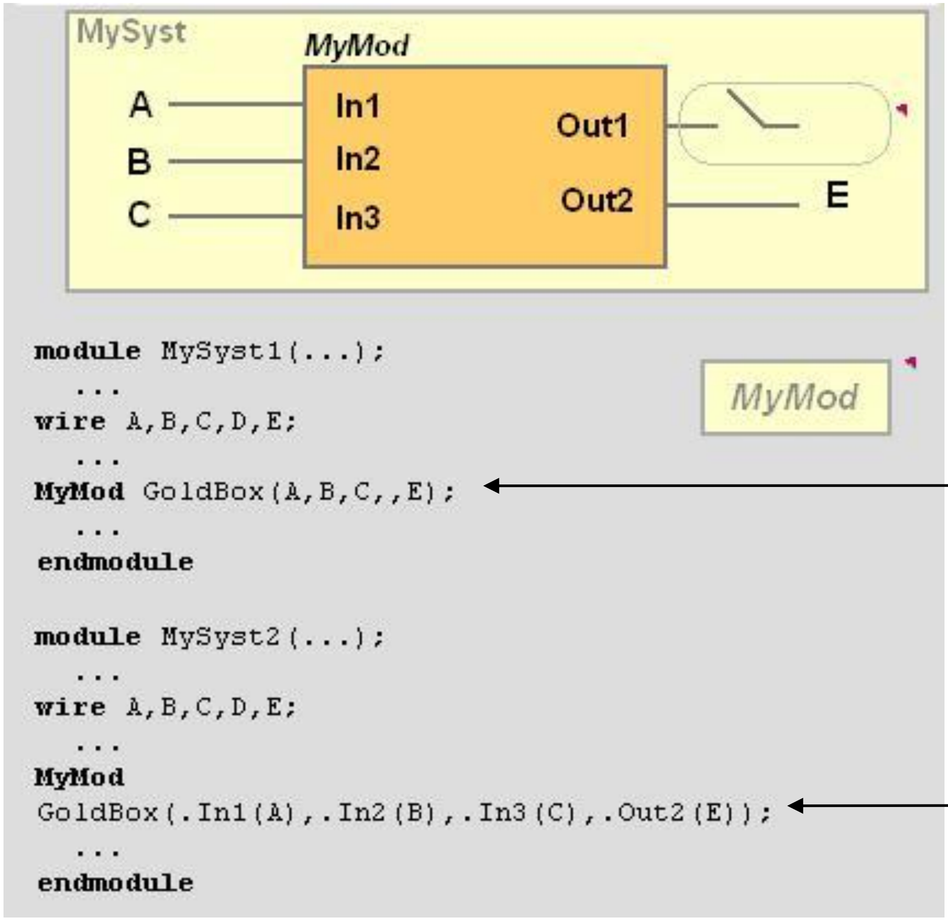
```
module MySyst(...);
   ...
~~wire A, B, C;~~
   ...
Hexagon MyMod(.In1(A) , .In2(B) , .Out1(C));
   ...
endmodule
```

(from ALDEC tutorial)

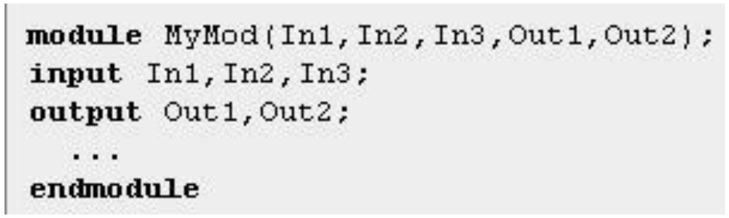
— The two methods cannot be mixed on the same!

Module instantiation

- Unconnected ports



```
module MySyst1(...);
   ...
wire A,B,C,D,E;
   ...
MyMod GoldBox(A,B,C,,E);
   ...
endmodule

module MySyst2(...);
   ...
wire A,B,C,D,E;
   ...
MyMod
GoldBox(.In1(A),.In2(B),.In3(C),.Out2(E));
   ...
endmodule
```



```
module MyMod(In1,In2,In3,Out1,Out2);
input In1,In2,In3;
output Out1,Out2;
   ...
endmodule
```

by port list order

by name

(from ALDEC tutorial)

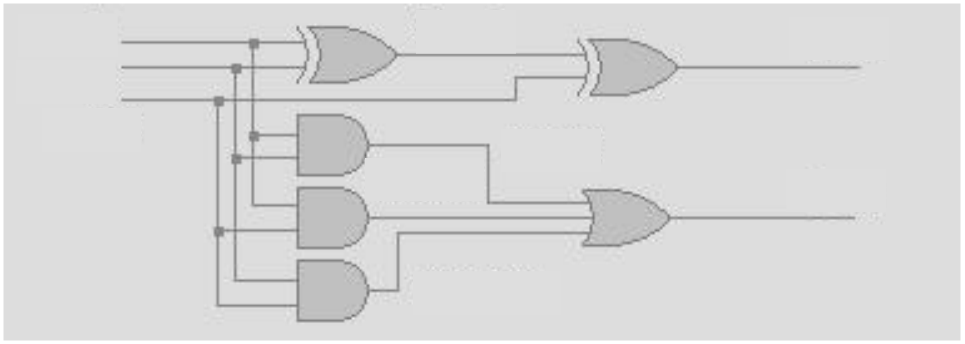
Creating Structural Verilog code

- Steps to convert a schematic to structural Verilog code:

1) **Decide a meaningful module name**
2) **Identify the module ports or I/O interface**
3) **Label the internal nets or wires in the schematic with unique names**
4) **Label each instance with a unique instance name**
5) **Enumerate each instance in the design with its module type, instance name, and signal bindings**

Example Logic Schematic

- Given following logic schematic



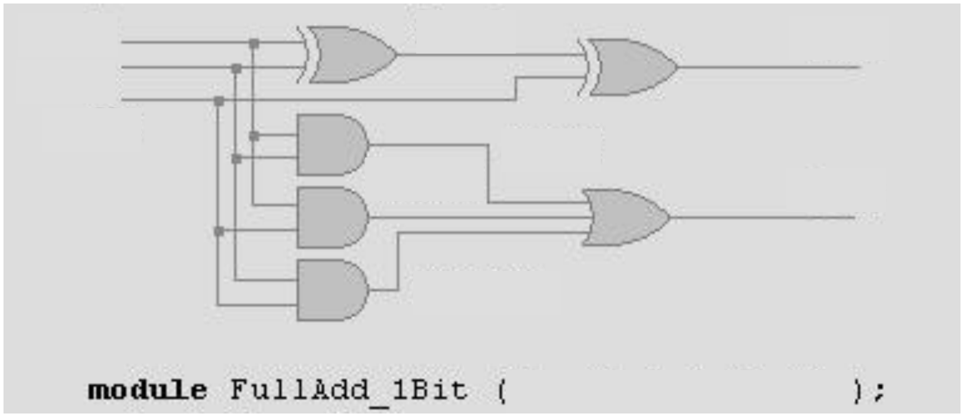
Creating Structural Verilog code

- Steps to convert a schematic to structural Verilog code:

1) **Decide a meaningful module name**
2) **Identify the module ports or I/O interface**
3) **Label the internal nets or wires in the schematic with unique names**
4) **Label each instance with a unique instance name**
5) **Enumerate each instance in the design with its module type, instance name, and signal bindings**

Example Logic Schematic

1) **Decide a meaningful module name**



```
module FullAdd_1Bit (                    );
```

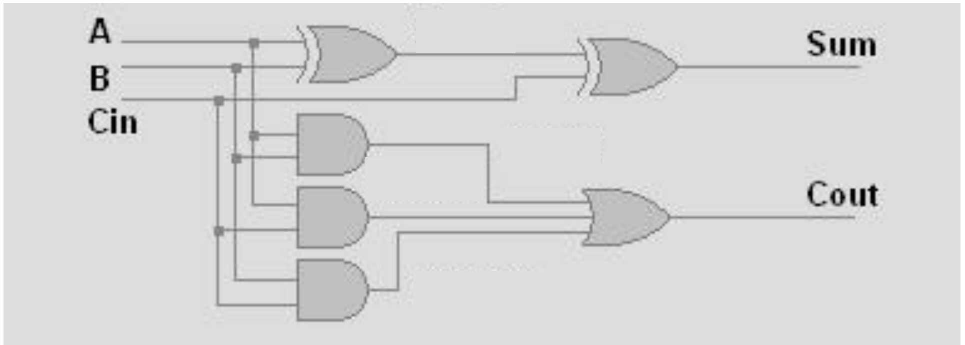
Creating Structural Verilog code

- Steps to convert a schematic to structural Verilog code:

1) **Decide a meaningful module name**
2) **Identify the module ports or I/O interface**
3) **Label the internal nets or wires in the schematic with unique names**
4) **Label each instance with a unique instance name**
5) **Enumerate each instance in the design with its module type, instance name, and signal bindings**

Example Logic Schematic

2) Identify the module ports or I/O interface



```
module FullAdd_1Bit (Sum,Cout,A,B, Cin);
output Sum, Cout;
input A, B, Cin;
```

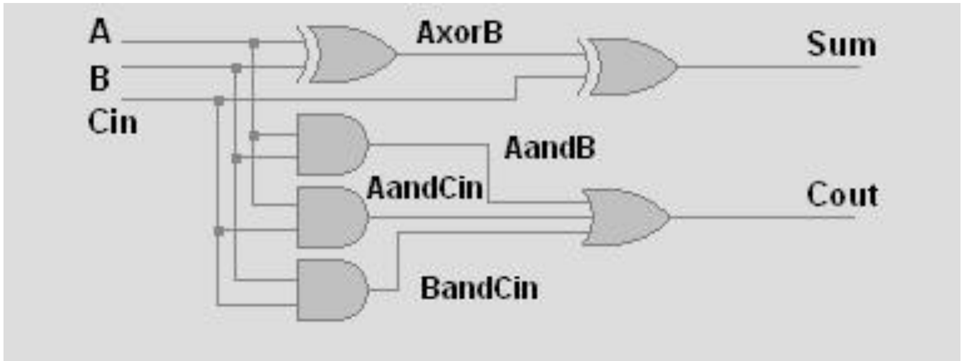
Creating Structural Verilog code

- Steps to convert a schematic to structural Verilog code:

1) **Decide a meaningful module name**
2) **Identify the module ports or I/O interface**
3) **Label the internal nets or wires in the schematic with unique names**
4) **Label each instance with a unique instance name**
5) **Enumerate each instance in the design with its module type, instance name, and signal bindings**

Example Logic Schematic

3) Label the internal nets or wires in the schematic with unique names



```
module FullAdd_1Bit (Sum,Cout,A,B, Cin);
output Sum, Cout;
input A, B, Cin;
```

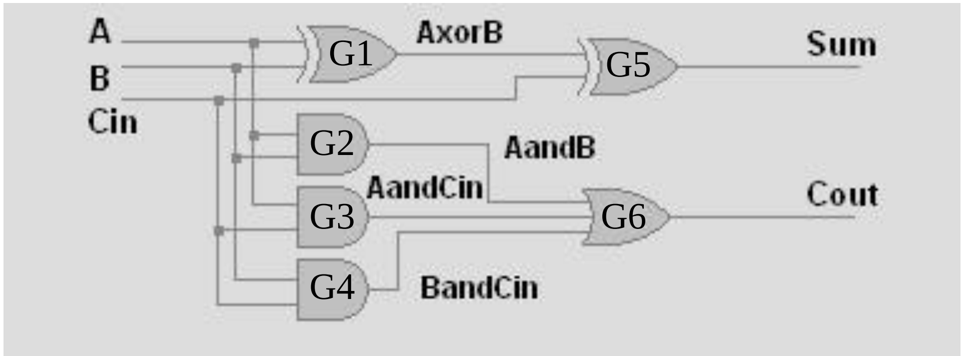
Creating Structural Verilog code

- Steps to convert a schematic to structural Verilog code:

1) **Decide a meaningful module name**
2) **Identify the module ports or I/O interface**
3) **Label the internal nets or wires in the schematic with unique names**
4) **Label each instance with a unique instance name**
5) **Enumerate each instance in the design with its module type, instance name, and signal bindings**

Example Logic Schematic

4) Label each instance with a unique instance name



```
module FullAdd_1Bit (Sum,Cout,A,B, Cin);
output Sum, Cout;
input A, B, Cin;
```

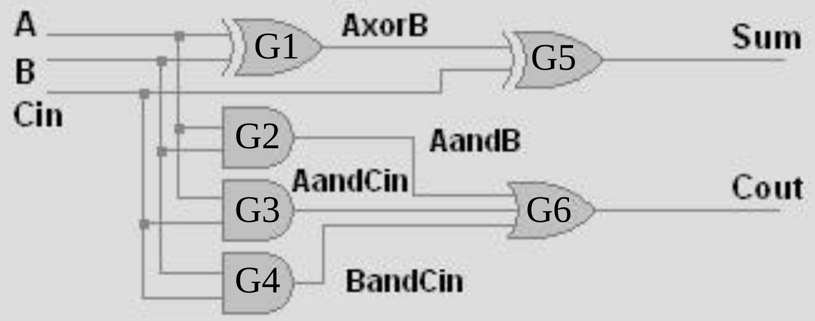
Creating Structural Verilog code

- Steps to convert a schematic to structural Verilog code:

1) **Decide a meaningful module name**
2) **Identify the module ports or I/O interface**
3) **Label the internal nets or wires in the schematic with unique names**
4) **Label each instance with a unique instance name**
5) **Enumerate each instance in the design with its module type, instance name, and signal bindings**

Completed Verilog Code

5) Enumerate each instance in the design with its module type, instance name, and signal bindings

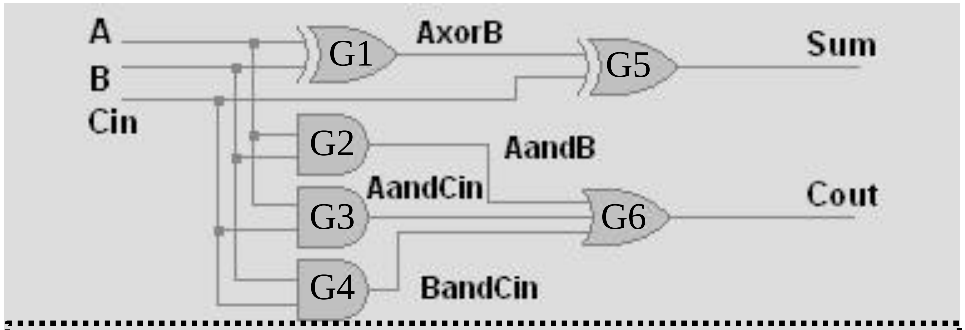


```
module FullAdd_1Bit (Sum,Cout,A,B, Cin);
output Sum, Cout;
input A, B, Cin;
xor G1 (AxorB, A, B) ;
and G2 (AandB, A, B) ;
and G3 (AandCin, A, Cin) ;
and G4 (BandCin, B, Cin) ;
xor G5 (Sum, AxorB, Cin) ;
or G6 (Cout, AandB, AandCin, BandCin) ;
endmodule
```

Completed Verilog Code

Using Verilog 2001 module interface

Recommended form



```
module FullAdd_1Bit(output Sum, Cout, input A, B, Cin) ;
        xor G1 (AxorB, A, B) ;
        and G2 (AandB, A, B) ;
        and G3 (AandCin, A, Cin) ;
        and G4 (BandCin, B, Cin) ;
        xor G5 (Sum, AxorB, Cin) ;
        or G6 (Cout, AandB, AandCin, BandCin) ;
        endmodule
```

Functional Verilog code

- So far, you learned how to write structural Verilog code

  Self evaluation:
  Can you translate any schematic into Verilog code?

- Sometimes, it is more convenient to use functional
  Verilog code. This is what are going to be discussed next.

Integer constants

- Un-sized integer example

  - 12 // decimal number 12
  - ’h12 // hex number 12 (18 decimal number)
  - ’o12 // octal number 12 (10 decimal number)
  - ’b1001 // binary number 1001 (9 decimal number)

- Sized integer example

  - 8’d12 // decimal number 12 taking 8 bits
  - 8’h12 // hex number 12 taking 8 bits
  - 8’b10010011 //
  - 8’b1 // binary number 00000001

<u>Note Verilog uses left padding</u>

Integer constants

- Negative numbers
  - Negative numbers are represented in 2’s complement form
  - - 8’d12 // stored as 11110100

```
      0000 1100         (+12)
      1111 0011
  +           1
      1111 0100         (-12)
```

- Use of ?, X, Z, _ characters
  - 8’h1? // 0001ZZZZ
  - 2’b1? // 1Z
  - 4’b10XX // 10XX
  - 4’b100Z // 100Z
  - 8’b1010_0011 // 10100011

Arithmetic operators

- Available operators: **+, -, *, /, % (modulo)**
- Arithmetic operators treat register operands as unsigned values
  - Example:

integer A;

A = -12;

A/4 $\Rightarrow$ -3

```
  0000 1100          (+12)
  1111 0011
+         1
  1111 0100          (-12)
111111 01~~00~~          (-3)
000000 10
+        1
000000 11            (+3)
```

reg [7:0] A;

A = -12;

A/4 $\Rightarrow$ 61

```
  0000 1100          (+12)
  1111 0011
+         1
  1111 0100          (-12)
001111 01~~00~~          (61)

32+16+8+4+1 = 61
```

Relation and equality operators

- Available relational operators: **<, <=, >, >=**
  - If any bit of an operand is X or Z, the result will be X
- Available equality operators: **===, !==, ==, !=**
  - ===, !== : case equality (inequality). X and Z values are considered in comparison
  - ==, != : logic equality (inequality). If any bit of an operand is X or Z, the result will be X

Example

Left Op.	Right Op.	===	!==	==	!=
0110	0110	1	0	1	0
0110	0XX0	0	1	X	X
0XX0	0XX0	1	0	X	X

Logic operators

- Logic operators:
  - && (logic and), || (logic or), ! (logic not)

Operand A	Operand B	A&B	A|B	!A	!B
1010	00	0	1	0	1
1010	011	1	1	0	0

- Bit-wise logic operators:
  - & (and), | (or), ~ (not), ^ (xor), ~^ (xnor)

Operand A	Operand B	A&B	A|B	~A	A^B	A~^B
1010	0011	0010	1011	0101	1001	0110

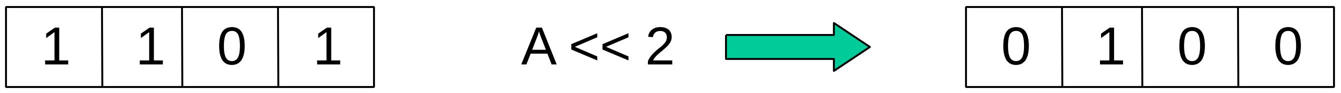
- Reducation operators:
  - & (and), ~& (nand), | (or), ~| (nor), ^ (xor), ~^ (xnor)

Operand A	&A	~&A	|A	~|A	^A	~^A
1010	0	1	1	0	0	1

Shifter operators

- << : shift left

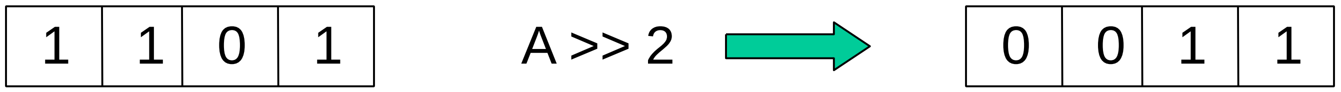
reg [3:0] A;



— zeros are moved in from the right end

- >> : shift right

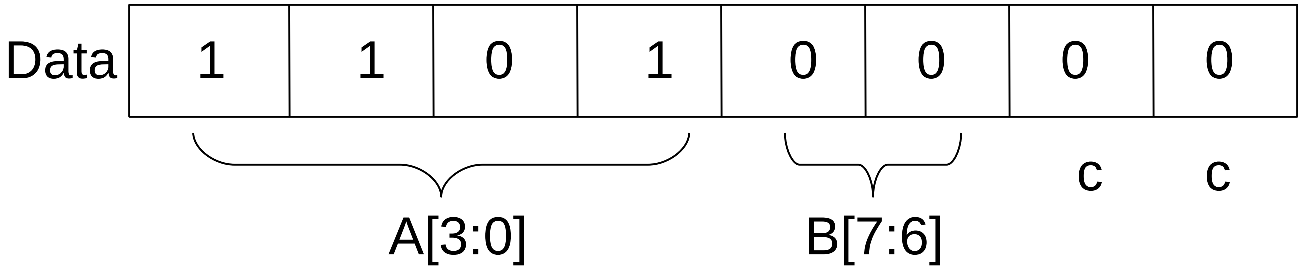
reg [3:0] A;



Concatenation operators

- Example

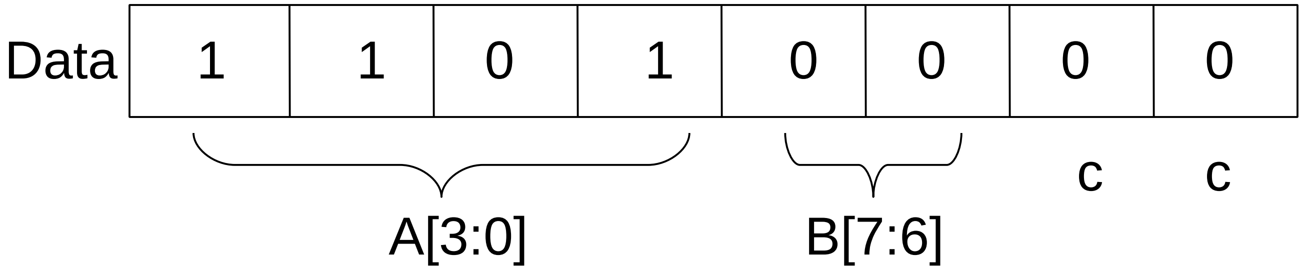
```
reg [7:0] A, B, Data;
reg c;
……
A = 10101101; B= 00110011;
c = 0;
Data = {A[3:0], B[7:6], c, c};       // Data = 11010000
```



Concatenation operators

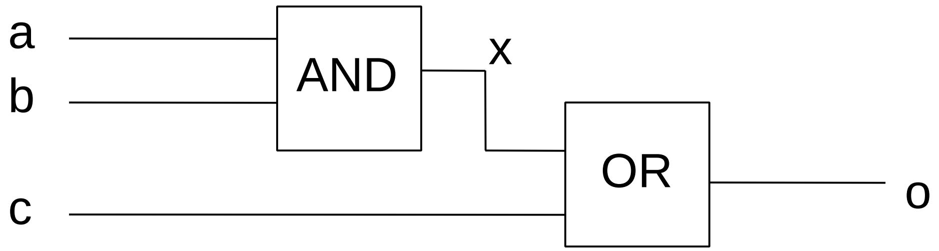
- Example

```
reg [7:0] A, B, Data;
reg c;
……
A = 10101101; B= 00110011;
c = 0;
Data = {A[3:0], B[7:6], c, c};      // Data = 11010000
```



Continuous assignment

- Continuous assignment starts with keyword **assign.**
- The left hand side of a continuous assignment command must be a net-type signal**.**
- Example



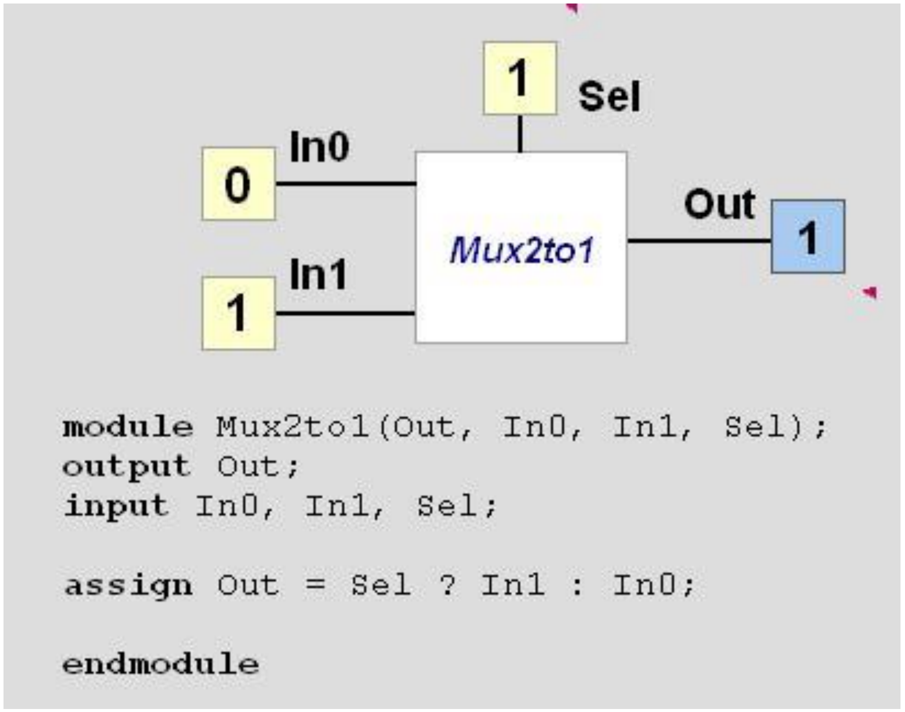
```
module cir1 (o, a, b, c);
output o;
input a, b, c;
wire x;
assign x = a & b;
assign o = x | c;
endmodule
```

OR

```
module cir1 (o, a, b, c);
output o;
input a, b, c;
wire x = a & b;
assign o = x | c;
endmodule
```

Conditional assignment

- A conditional assignment has three signals at the right hand side.
  - The first signal is the control signal
  - If the control signal is true, the second signal is assigned to the left hand side (LHS) signal ; otherwise, the third signal is assigned to LHS signal.



```
module Mux2to1(Out, In0, In1, Sel);
output Out;
input In0, In1, Sel;

assign Out = Sel ? In1 : In0;

endmodule
```

(from ALDEC tutorial)

Adding delay to continuous assignment

- Delay is added by # t after keyword **assign,** t is the number of delayed time unit.
- Time unit is defined by `timescale
- Example

```
`timescale  10ns/1ns          // <ref_time_unit>/<time_precision>
module buf1 (o, i);
output o;
input i;

assign #3 o = 1;            // delay for 3 time unit

endmodule
```

Behavioral blocks

- In additional to assignment, other functional description codes are included in two-type behavioral blocks:

  initial blocks and ***always blocks***

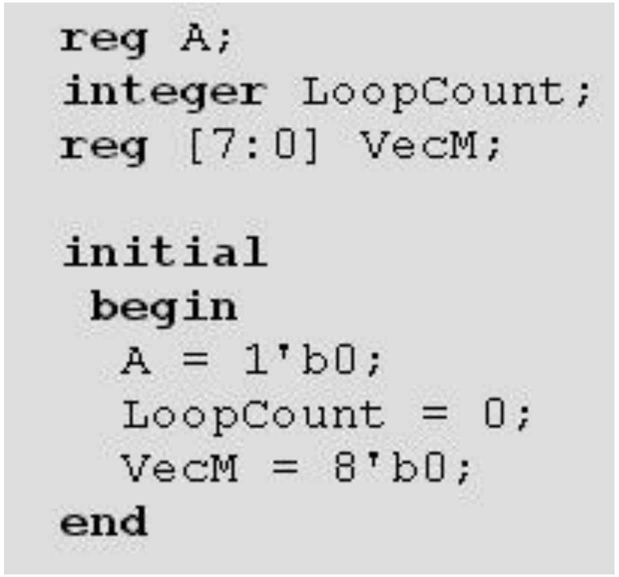
- A module can have multiple blocks, but blocks cannot be nested.
- When a block has multiple statements, they must be grouped using **begin** and **end** (for sequential statements) or **fork** and **join** (for concurrent statements).
- An initial block is executed at the beginning of simulation. It is executed only once.
- Always blocks are repeated executed until simulation is stopped.

Procedural assignment

- Procedural assignment is used to assign value to variables.
- A variable can be a signal defined by **reg** or a name defined by **integer** (another form of register-type signal). A variable cannot be a net type signal.
- Variable assignments must be in behavioral blocks.
- Difference between continuous assignment and procedural assignment
  - In continuous assignment changes the value of the target net whenever the right-hand-side operands change value.
  - Procedural assignment changes the target register only when the assignment is executed according to the sequence of operations

Procedural assignment examples

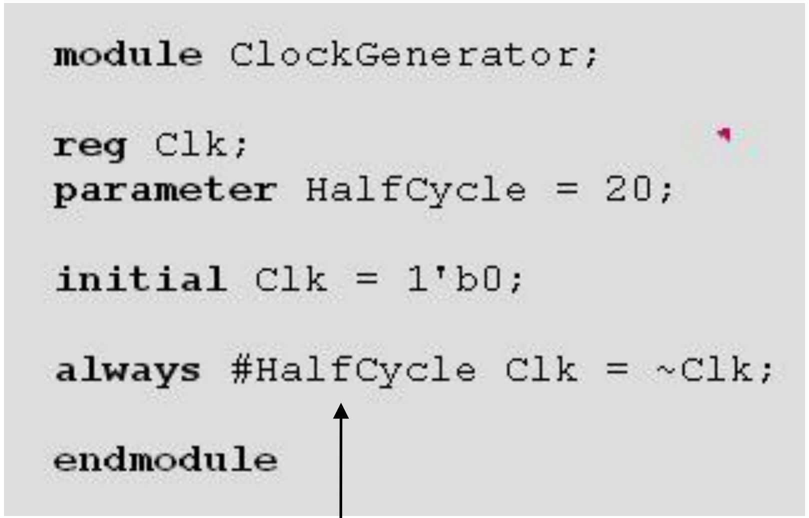
- Signal initialization



```
reg A;
integer LoopCount;
reg [7:0] VecM;

initial
 begin
  A = 1'b0;
  LoopCount = 0;
  VecM = 8'b0;
end
```

(from ALDEC tutorial)

- generating clock



```
module ClockGenerator;

reg Clk;
parameter HalfCycle = 20;

initial Clk = 1'b0;

always #HalfCycle Clk = ~Clk;

endmodule
```

Note how to specify delay in procedural assignment

Delay in procedural assignments

- Delay specified in front of procedural assignment statements (e.g. #3 a = b&c;) delay the execution of the entire statement.

```
Module delayTest;
integer a, b, c;

initial begin
a = 2; b = 3;
end

initial #3 a = 4;

initial #5 c = a+b;

endmodule
```

Change a from 2 to 4 after 3 time unit

Execution order:
1. delay
2. evaluation
3. assignment

Result: c=7

Delay in procedural assignments

- Delay specified right after = in procedural assignment statements (e.g. a = #3 b&c;) just delay the assignment operation. The evaluation of the right hand side expression is executed without delay.

```
Module delayTest;
integer a, b, c;

initial begin
a = 2; b = 3;
end

initial #3 a = 4;

initial c = #5 a+b;

endmodule
```

Change a from 2 to 4
after 3 time unit

Execution order:
1. evaluation
2. delay
3. assignment

Result: c=5

Blocking assignments ***vs.*** Non-blocking assignments

- Blocking assignments use = as assignment symbol (previously discussed procedural assignments). Assignments are performed sequentially.

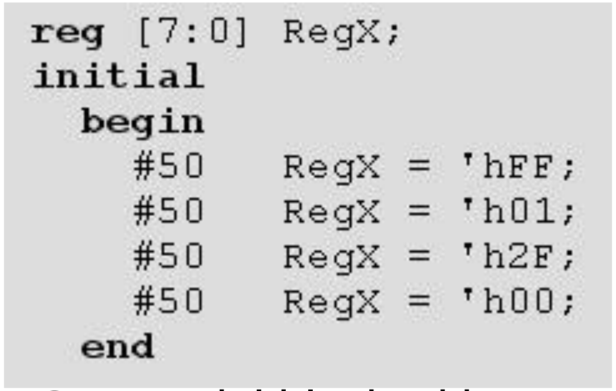
```
initial begin
a = #1 1;          // assignment at time 1
b = #3 0;          // assignment at time 4 (3+1)
c = #6 1;          // assignment at time 10 (6+3+1)
end
```

- Non-blocking assignments use <= as assignment symbol. Non-blocking assignments are performed concurrently.

```
initial begin
#1 a <=  1;        // assignment at time 1
#3 b <=  0;        // assignment at time 3
#6 c <=  1;        // assignment at time 6
end
```

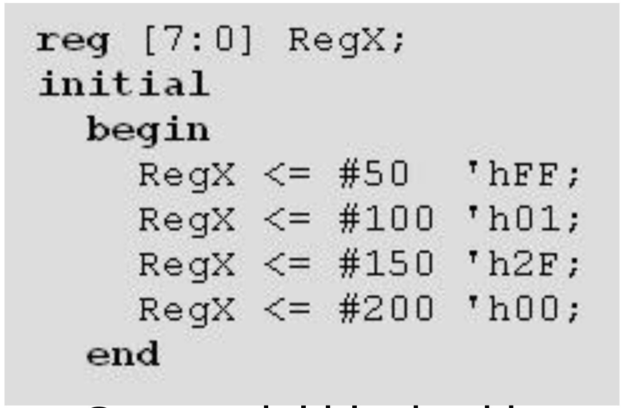
Parallel blocks

- Parallel block is a more flexible method to write concurrent statements. It uses **fork** and **join**, instead of **begin** and **end**, in block description.



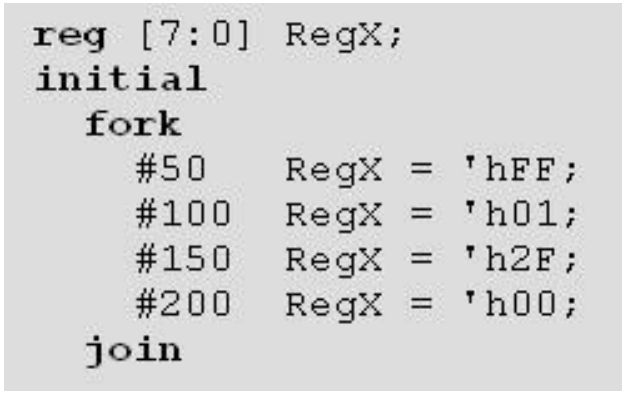
```
reg [7:0] RegX;
initial
  begin
    #50   RegX = 'hFF;
    #50   RegX = 'h01;
    #50   RegX = 'h2F;
    #50   RegX = 'h00;
  end
```

Sequential block with blocking assignments



```
reg [7:0] RegX;
initial
  begin
    RegX <= #50  'hFF;
    RegX <= #100 'h01;
    RegX <= #150 'h2F;
    RegX <= #200 'h00;
  end
```

Sequential block with Non-blocking assignments



```
reg [7:0] RegX;
initial
  fork
    #50   RegX = 'hFF;
    #100  RegX = 'h01;
    #150  RegX = 'h2F;
    #200  RegX = 'h00;
  join
```

Parallel block

(from ALDEC tutorial)

Event control statements

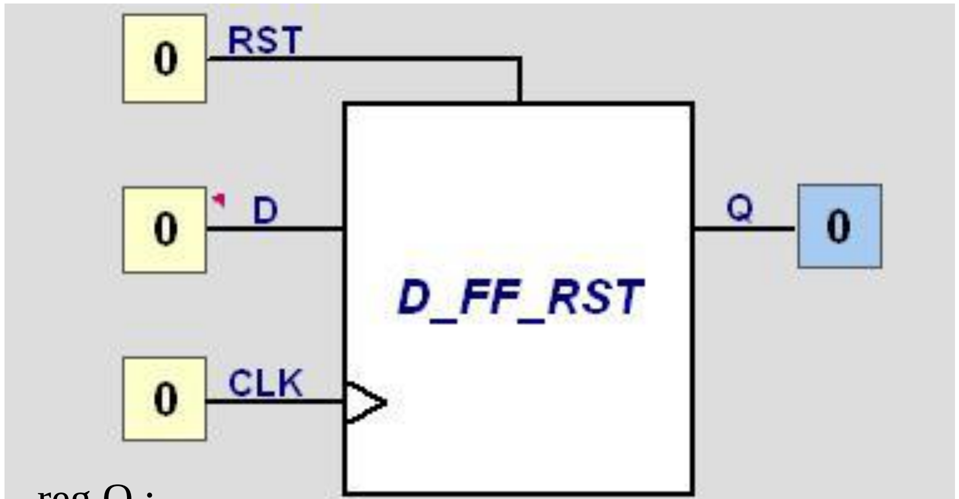
- ❑ An event occurs when a net or register changes it value. The event can be further specified as a rising edge (by **posedge**) or falling edge (by **negedge**) of a signal.

- ❑ An event control statement always starts with symbol @

```
@ (clk) Q = D;              // assignment will be performed whenever
                                   signal clk changes to its value

@ (posedge clk) Q = D; // assignment will be performed whenever
                                   signal clk has a rising edge (0→1, 0→X,
                                   0→Z, X→1, Z→1)

@ (negedge clk) Q = D; // assignment will be performed whenever
                                   signal clk has a falling edge (1→0, 1→X,
                                   1→Z, X→0, Z→0)
```

Sensitivity list

- Sensitivity list specifies events on which signals activating always blocks



```
reg Q ;
always @(posedge RST or posedge CLK)
  begin
      if (RST)
        Q = 1'b0;
      else if (posedge CLK)
        Q = D;
  end
```

(from ALDEC tutorial)

Wait statements

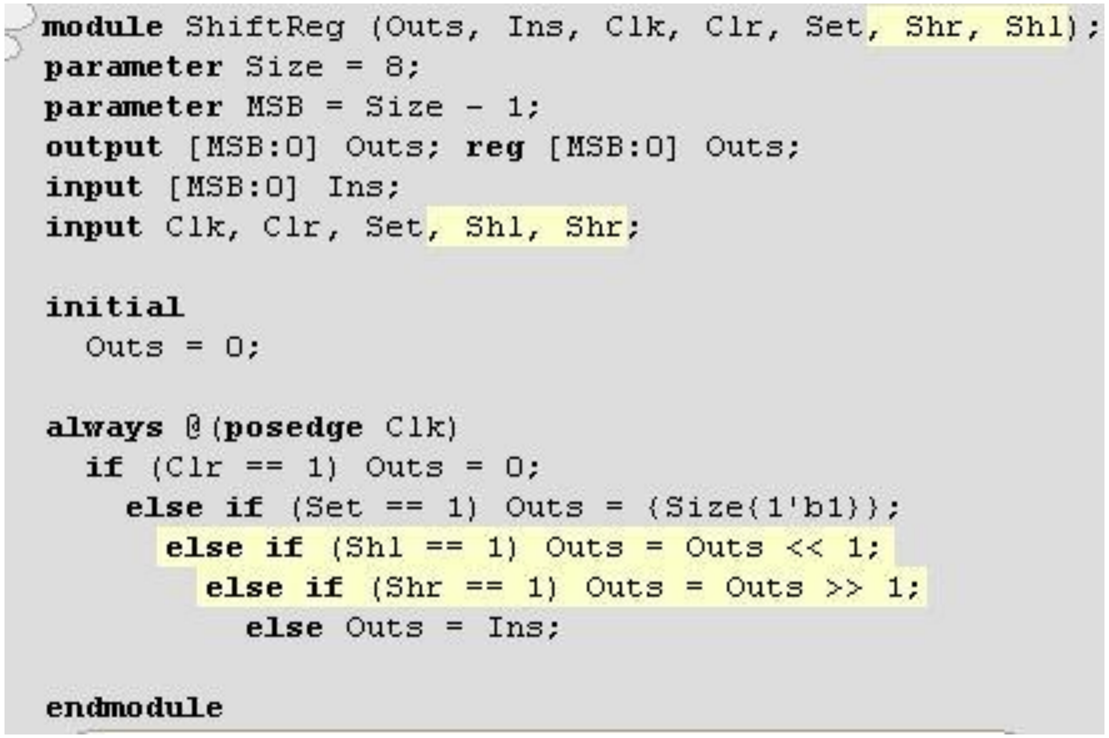
- Wait statements allow designers to more specifically control when to execute statements.
- A wait statement starts with keyword wait followed by:
  - A logic condition that determines when to execute the statements. The condition is specified in brackets.
  - Statements that will be executed

```
module testWait;
integer a, b, c;
reg en;

initial a = 0;
initial #3 a = 3;
intial #6 a = 7;

wait (a==7) b = 1; // assign 1 to b when a=7

wait (en) c = 2;     // assign 2 to c when en is true (en is like enable signal)

endmodule
```

Conditional statements

- Conditional statement is another method to control when statements are executed

If (*condition*) *true_statements*;
else *false_statements*;

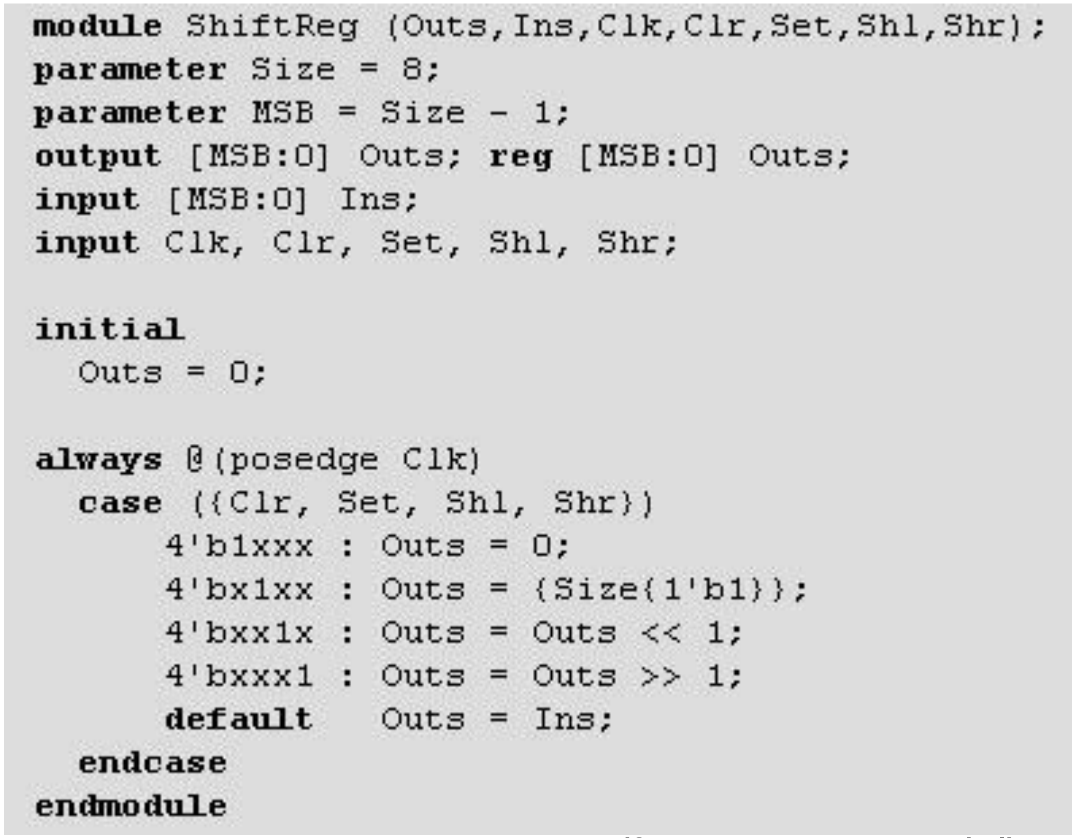


```
module ShiftReg (Outs, Ins, Clk, Clr, Set, Shr, Shl);
parameter Size = 8;
parameter MSB = Size - 1;
output [MSB:0] Outs; reg [MSB:0] Outs;
input [MSB:0] Ins;
input Clk, Clr, Set, Shl, Shr;

initial
  Outs = 0;

always @(posedge Clk)
  if (Clr == 1) Outs = 0;
    else if (Set == 1) Outs = {Size{1'b1}};
      else if (Shl == 1) Outs = Outs << 1;
        else if (Shr == 1) Outs = Outs >> 1;
          else Outs = Ins;

endmodule
```

(from ALDEC tutorial)

Multiple choice statements

- Multiple choice statement starts with keyword **case**. It offers a more readable alternative to nested if-else statements.

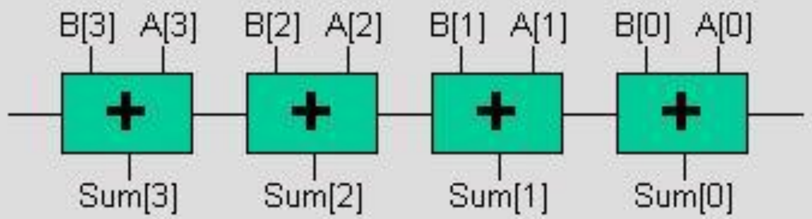


```
module ShiftReg (Outs,Ins,Clk,Clr,Set,Shl,Shr);
parameter Size = 8;
parameter MSB = Size - 1;
output [MSB:0] Outs; reg [MSB:0] Outs;
input [MSB:0] Ins;
input Clk, Clr, Set, Shl, Shr;

initial
  Outs = 0;

always @(posedge Clk)
  case ({Clr, Set, Shl, Shr})
      4'b1xxx : Outs = 0;
      4'bx1xx : Outs = {Size{1'b1}};
      4'bxx1x : Outs = Outs << 1;
      4'bxxx1 : Outs = Outs >> 1;
      default   Outs = Ins;
  endcase
endmodule
```

(from ALDEC tutorial)

Loop statements

- Loop statements include **forever, repeat, while**, and **for**

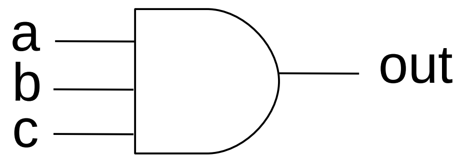


```
module RCAdd(Sum, A, B);
  output [3:0] Sum; reg [3:0] Sum;
  input [3:0] A, B;
  reg C;
  integer I;
  always
    begin
      C = 0;
      for (I = 0; I <= 3; I = I + 1)
        begin
          Sum[I] = A[I] ^ B[I] ^ C;
          C = A[I] & B[I] | A[I] & C | B[I] & C;
        end
    end
endmodule
```

(from ALDEC tutorial)

Concatenation operator revisited

- Example: exhaustive testing truth table



```
integer i = 0 ;
reg a, b, c ;
wire out ;
parameter num_inputs = 3 ;
parameter max_count = (1 << num_inputs) – 1 ;


for ( i = 0 ; i <=  max_count ; i = i + 1 )
   begin
      {a,b,c} = i ;
      # 100 ;
   end
end
```

treated as a number

Generate truth table

a	b	c	number
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Overriding Parameters

- Suppose we have defined a RAM with the following default parameters

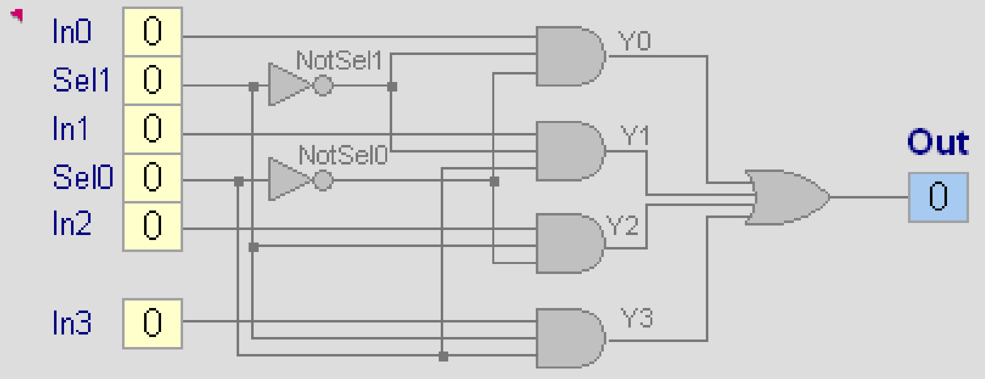
```
module  ram_sp_sr_sw (       clk          , // Clock Input
                             address      , // Address Input
                             data         , // Data bi-directional
                             cs           , // Chip Select
                             we           , // Write Enable/Read Enable
                             oe             // Output Enable ) ;

          parameter DATA_WIDTH = 8 ;
          parameter ADDR_WIDTH = 8 ;
          parameter RAM_DEPTH = 1 << ADDR_WIDTH;
          // Actual code of RAM here

endmodule
```

- In Verilog 2001, we can override the defaults during module instantiation in a structural definition

```
module  ram_controller () ;
  .
  .
  .
ram_sp_sr_sw #(
      .DATA_WIDTH(16),
      .ADDR_WIDTH(8),
      .RAM_DEPTH(256))  ram1(clk,address,data,cs,we,oe);

endmodule
```

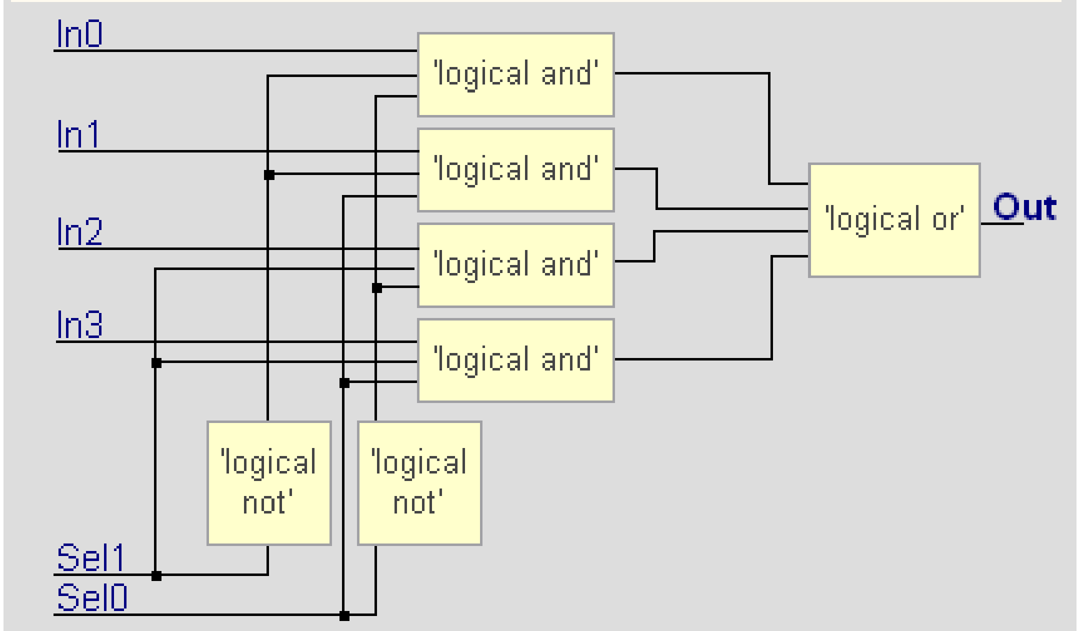
Here DATA_WIDTH is changed to 16

Structural style: Verilog Code



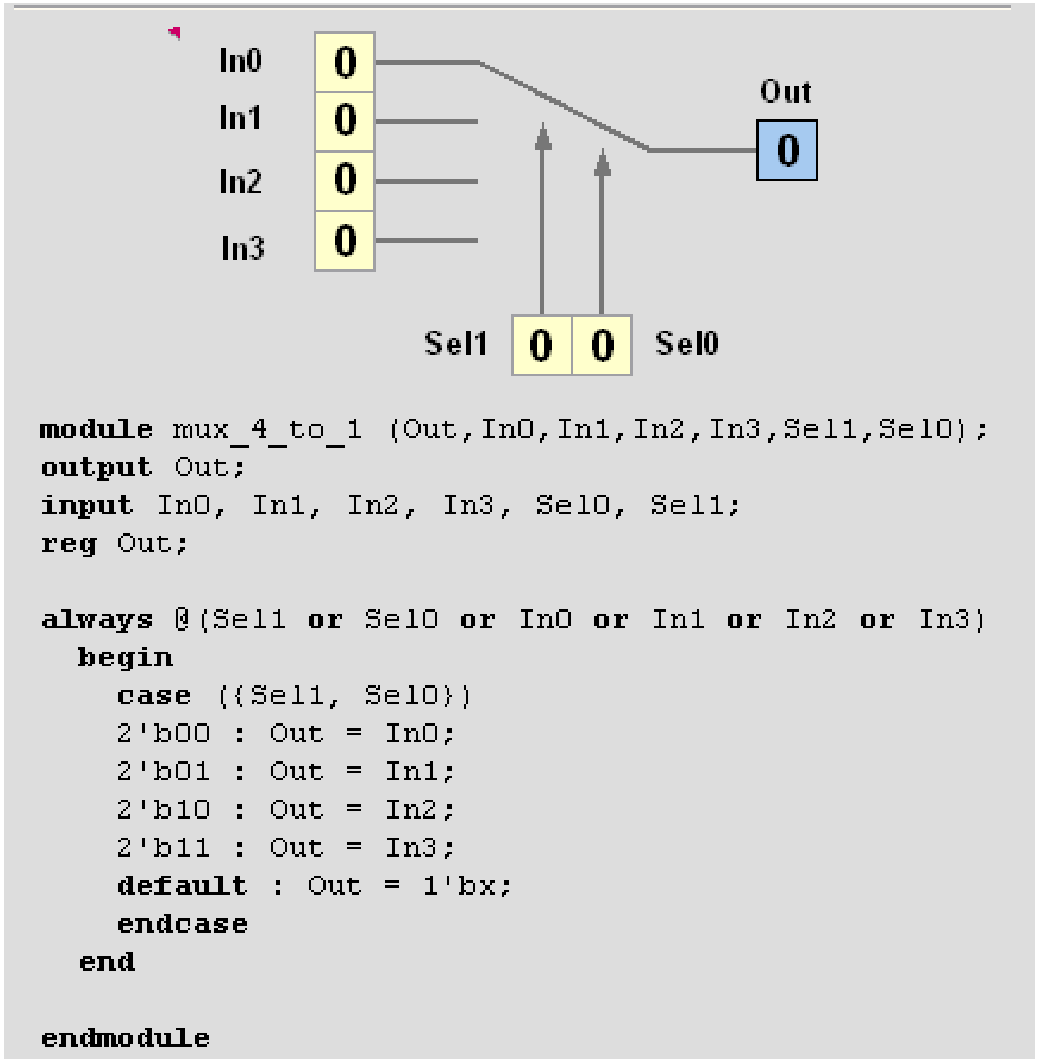
```
module mux_4_to_1 (Out,In0,In1,In2,In3,Sel1,Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;

~~wire NotSel0, NotSel1;~~
~~wire Y0, Y1, Y2, Y3;~~

not (NotSel0, Sel0);
not (NotSel1, Sel1);
and (Y0, In0, NotSel1, NotSel0);
and (Y1, In1, NotSel1, Sel0);
and (Y2, In2, Sel1, NotSel0);
and (Y3, In3, Sel1, Sel0);
or (Out, Y0, Y1, Y2, Y3);

endmodule
```

Dataflow style: Verilog Code



```
module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);

output Out;
input In0, In1, In2, In3, Sel0, Sel1;

assign Out = (~Sel1 & ~Sel0 & In0) | (~Sel1 & Sel0 & In1)
             | (Sel1 & ~Sel0 & In2)| (Sel1 & Sel0 & In3);

endmodule
```

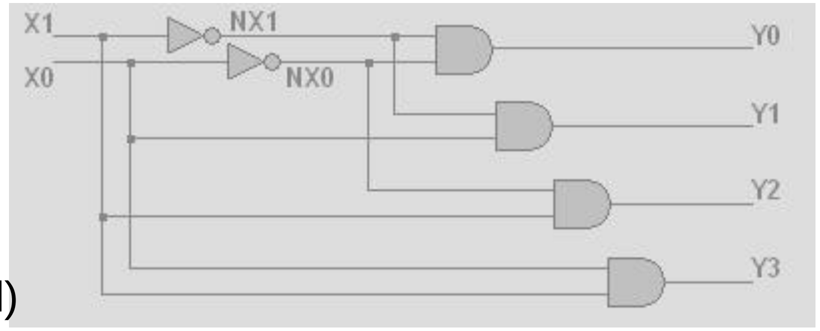
Behavioral style: Verilog Code



```
module mux_4_to_1 (Out,In0,In1,In2,In3,Sel1,Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;
reg Out;

always @(Sel1 or Sel0 or In0 or In1 or In2 or In3)
  begin
    case ({Sel1, Sel0})
    2'b00 : Out = In0;
    2'b01 : Out = In1;
    2'b10 : Out = In2;
    2'b11 : Out = In3;
    default : Out = 1'bx;
    endcase
  end

endmodule
```

A simple combinational circuit example

- A 2-to-4 decoder



(from ALDEC tutorial)

Circuit schematic

```
module Dec2to4 (Y, X);
  output [3:0] Y;
  input [1:0] X;

  ~~wire NX1, NX0;~~

  not (NX1, X[1]);
  not (NX0, X[0]);
  and (Y[0], NX1, NX0);
  and (Y[1], NX1, X[0]);
  and (Y[2], X[1], NX0);
  and (Y[3], X[1], X[0]);

endmodule
```

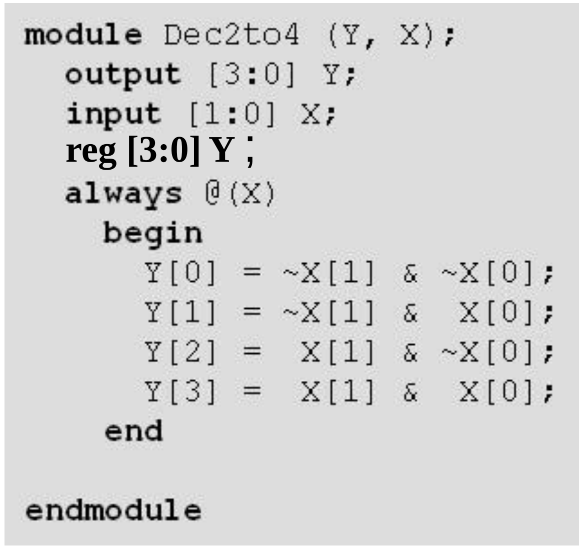
Structural code

```
module Dec2to4 (Y, X);
  output [3:0] Y;
  input [1:0] X;

  assign Y[0] = ~X[1] & ~X[0];
  assign Y[1] = ~X[1] &  X[0];
  assign Y[2] =  X[1] & ~X[0];
  assign Y[3] =  X[1] &  X[0];

endmodule
```

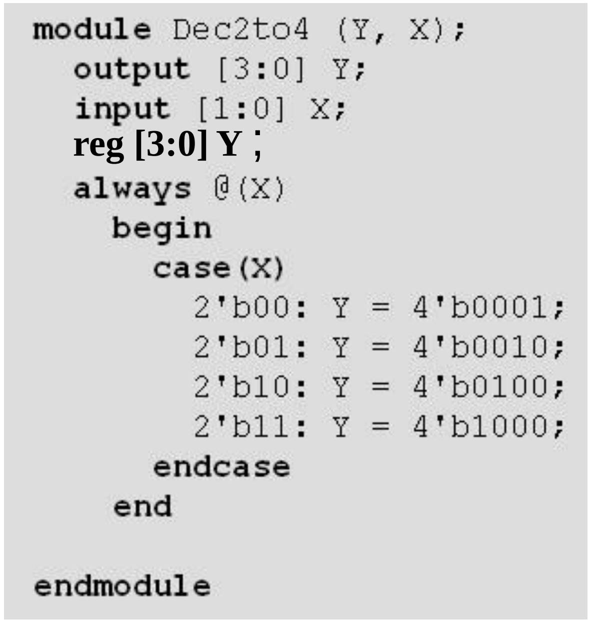
Data flow code

A simple combinational circuit example

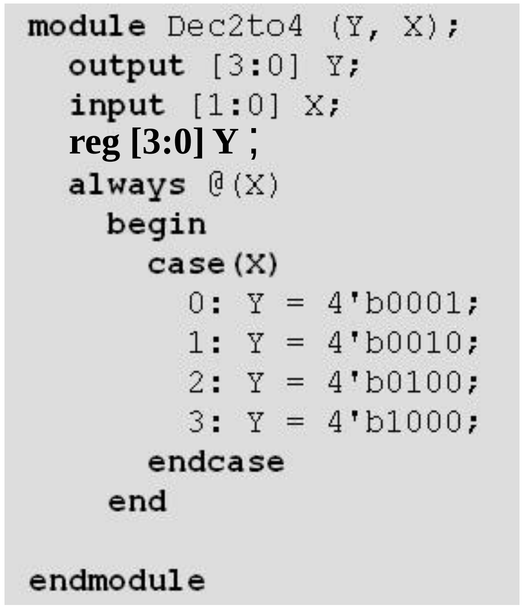
- A 2-to-4 decoder behavioral Verilog code



```
module Dec2to4 (Y, X);
  output [3:0] Y;
  input [1:0] X;
  reg [3:0] Y ;
  always @(X)
    begin
      Y[0] = ~X[1] & ~X[0];
      Y[1] = ~X[1] &  X[0];
      Y[2] =  X[1] & ~X[0];
      Y[3] =  X[1] &  X[0];
    end

endmodule
```



```
module Dec2to4 (Y, X);
  output [3:0] Y;
  input [1:0] X;
  reg [3:0] Y ;
  always @(X)
    begin
      case(X)
        2'b00: Y = 4'b0001;
        2'b01: Y = 4'b0010;
        2'b10: Y = 4'b0100;
        2'b11: Y = 4'b1000;
      endcase
    end

endmodule
```

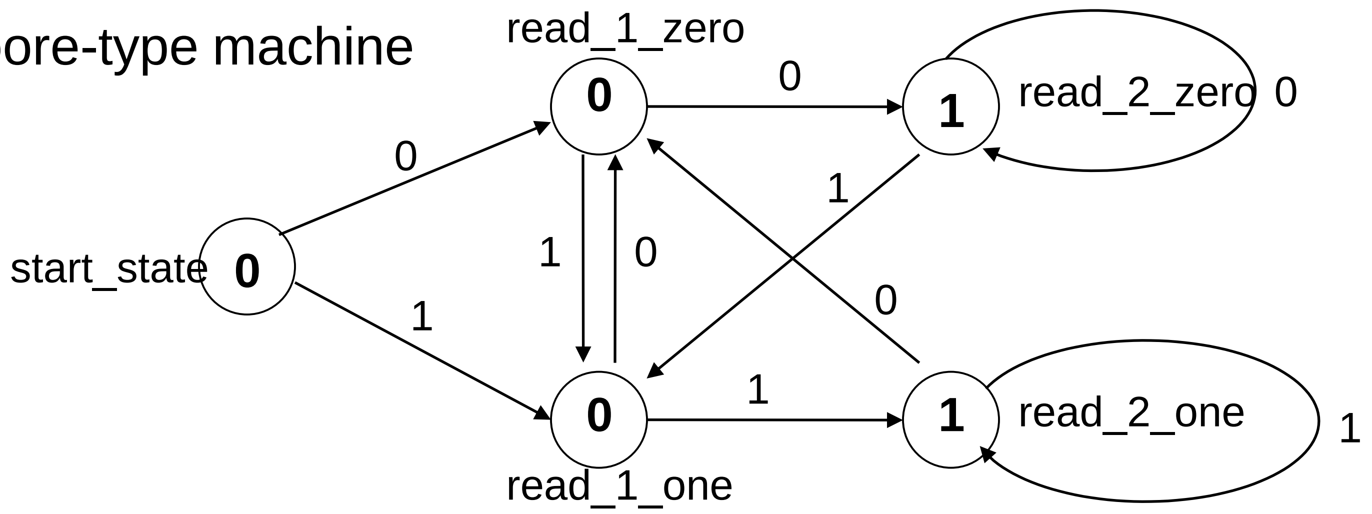


```
module Dec2to4 (Y, X);
  output [3:0] Y;
  input [1:0] X;
  reg [3:0] Y ;
  always @(X)
    begin
      case(X)
        0: Y = 4'b0001;
        1: Y = 4'b0010;
        2: Y = 4'b0100;
        3: Y = 4'b1000;
      endcase
    end

endmodule
```

(from ALDEC tutorial)

An FSM example

- Moore-type machine



```
module moore_explicit (clock, reset, in_bit, out_bit);
input                   clock, reset, in_bit;
output                  out_bit;
reg         [2:0]       state_reg, next_state;

parameter               start_state = 3'b000;
parameter               read_1_zero = 3'b001;
parameter               read_1_one  = 3'b010;
parameter               read_2_zero = 3'b011;
parameter               read_2_one  = 3'b100;
```

An FSM example

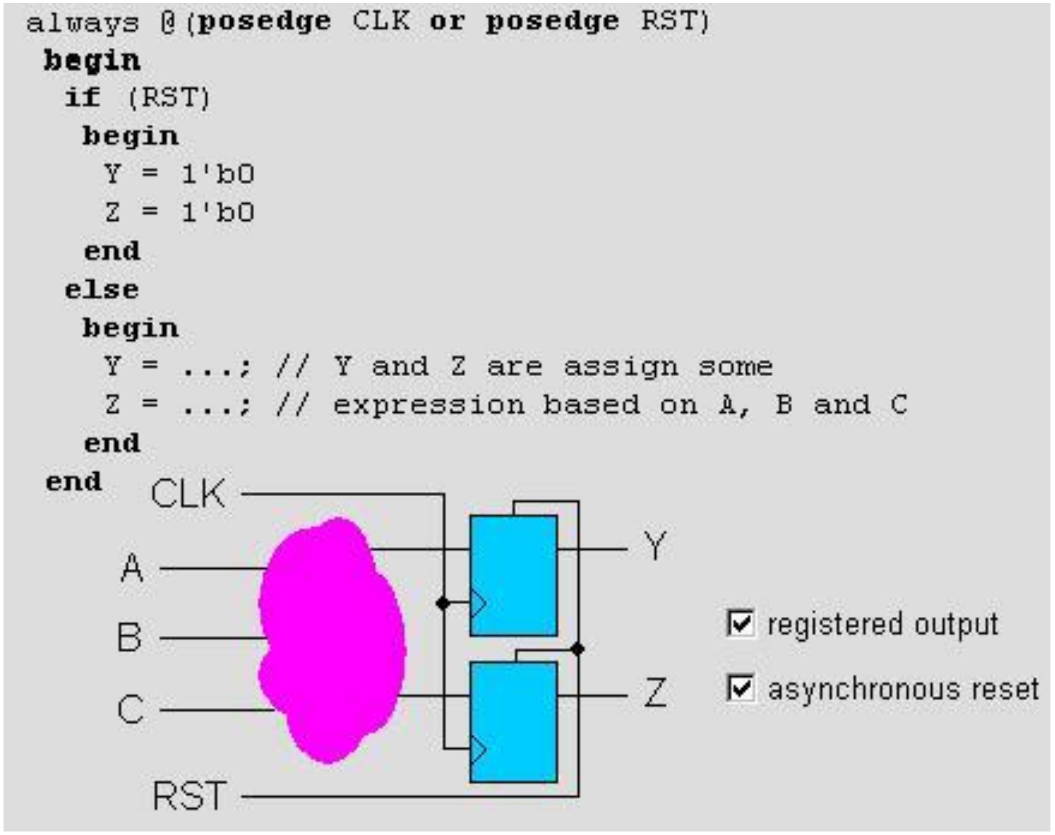
```
always @ (posedge clock or posedge reset)
    if (reset == 1) state_reg <= start_state; else state_reg <= next_state;

always @ (state_reg or in_bit)
    case (state_reg)
        start_state:
            if (in_bit == 0) next_state <= read_1_zero; else
                if (in_bit == 1) next_state <= read_1_one;

        read_1_zero:
            if (in_bit == 0) next_state <= read_2_zero; else
                if (in_bit == 1) next_state <= read_1_one;

        read_2_zero:
            if (in_bit == 0) next_state <= read_2_zero; else
                if (in_bit == 1) next_state <= read_1_one;
```

An FSM example

```
            read_1_one:
                if (in_bit == 0) next_state <= read_1_zero; else
                    if (in_bit == 1) next_state <= read_2_one;

            read_2_one:
                if (in_bit == 0) next_state <= read_1_zero; else
                    if (in_bit == 1) next_state <= read_2_one;

            default: next_state <= start_state;
        endcase

    assign out_bit =((state_reg == read_2_zero) || (state_reg ==read_2_one)) ? 1 : 0;
endmodule
```

Synthesizing registers

- Assignment inside a clocked always block will be synthesized as DFFs.



```
always @(posedge CLK or posedge RST)
 begin
  if (RST)
   begin
    Y = 1'b0
    Z = 1'b0
   end
  else
   begin
    Y = ...; // Y and Z are assign some
    Z = ...; // expression based on A, B and C
   end
 end
```

☑ registered output

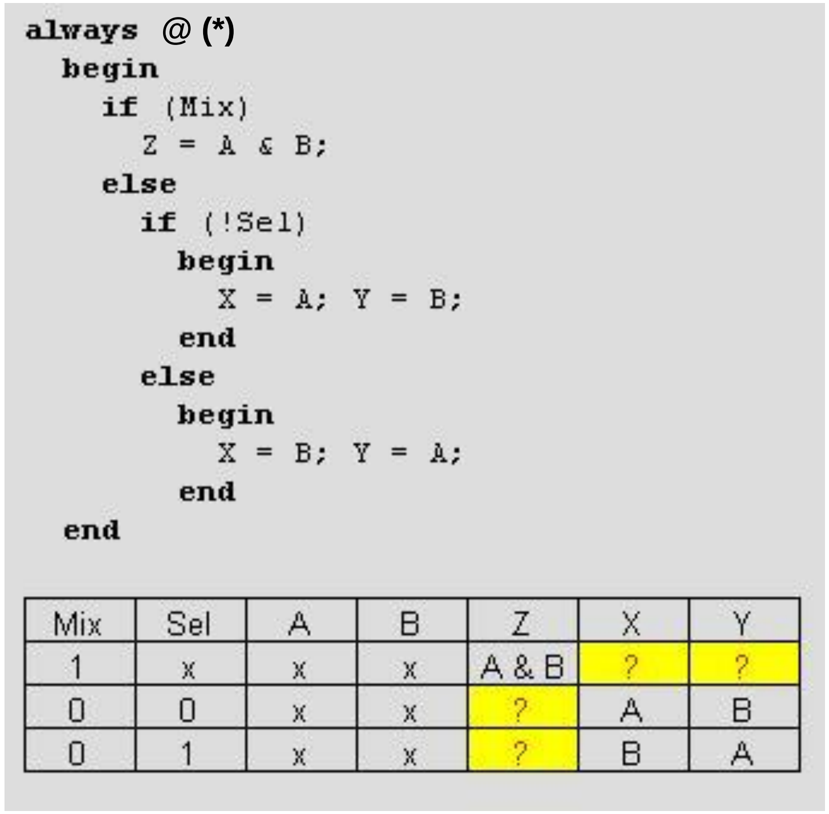
☑ asynchronous reset

(from ALDEC tutorial)

Avoiding unwanted latches

- Incomplete system specifications (if-else, or case) lead to unwanted latches
  - Latch-prone code
  - Latch-free code

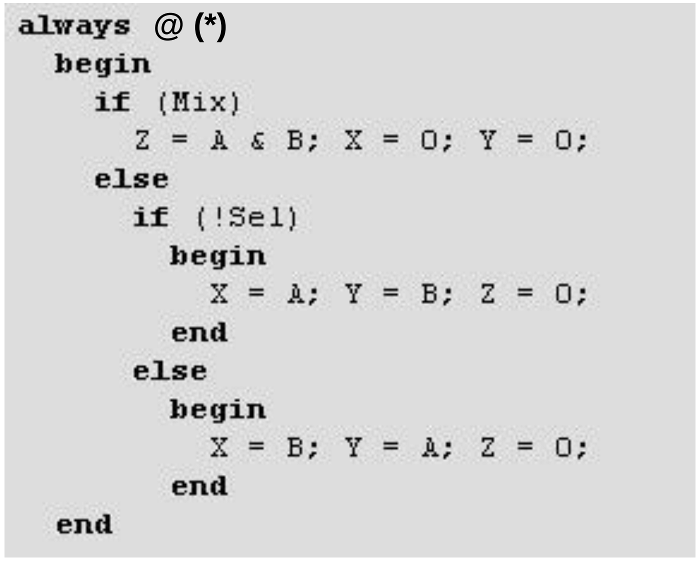
reg X, Y, Z ;



```
always  @ (*)
  begin
    if (Mix)
      Z = A & B;
    else
      if (!Sel)
        begin
          X = A; Y = B;
        end
      else
        begin
          X = B; Y = A;
        end
  end
```

Mix	Sel	A	B	Z	X	Y
1	x	x	x	A & B	?	?
0	0	x	x	?	A	B
0	1	x	x	?	B	A

reg X, Y, Z ;



```
always  @ (*)
  begin
    if (Mix)
      Z = A & B; X = 0; Y = 0;
    else
      if (!Sel)
        begin
          X = A; Y = B; Z = 0;
        end
      else
        begin
          X = B; Y = A; Z = 0;
        end
  end
```

(from ALDEC tutorial)

Other synthesis tips

- Nested if-else leads to lengthy mux-chain, which is normally slow. Using case instead. However, case results in mux with multiple inputs, which makes routing more difficult.

- Using instantiated module to implement arithmetic operators, instead of directly using arithmetic operators.

- Good partition leads to better results.

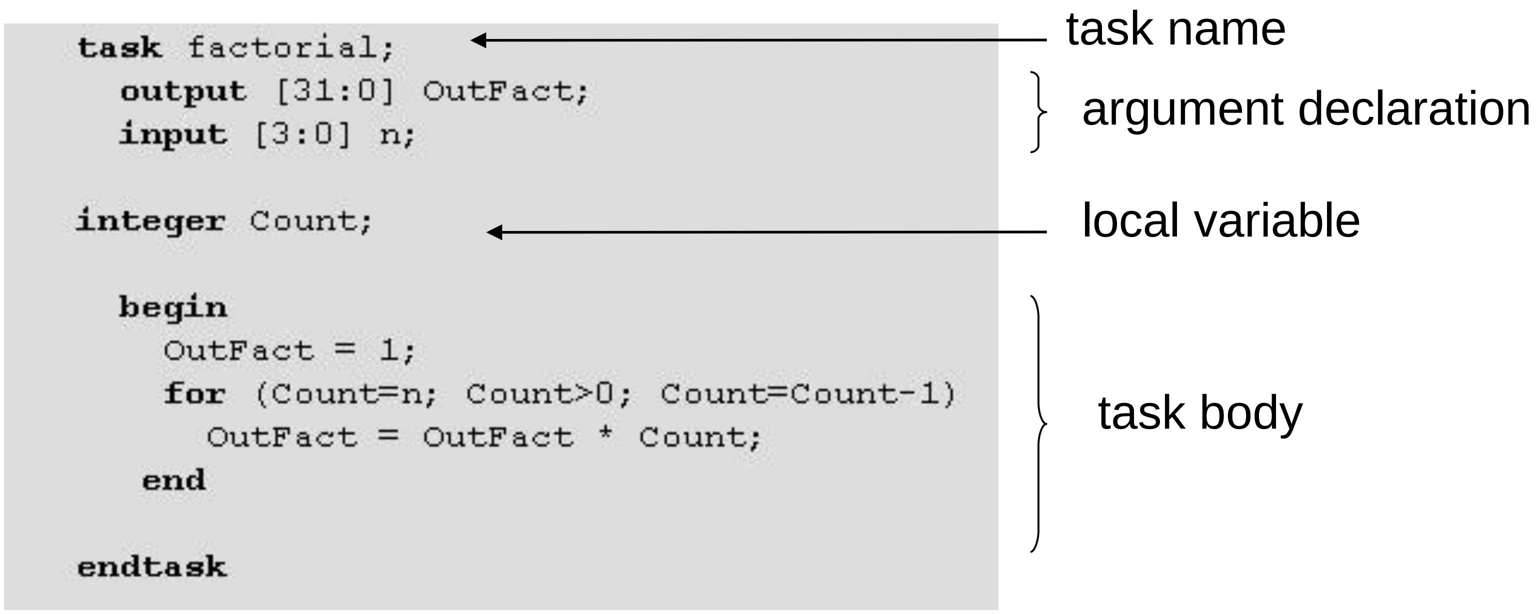
- Assign values to all outputs in all cases (to avoid unwanted latches).

Verilog subroutines

- ❑ Subroutines lead to more readable code and make code-reuse easy .
- ❑ Types of subroutines are: task and function.
- ❑ Subroutines can be used only in behavioral blocks and contain behavioral statements.
- ❑ Subroutines are declared with modules.
- ❑ Verilog offers a large set of system built-in tasks and functions.

Task example

- Task declaration



```
task factorial;
  output [31:0] OutFact;
  input [3:0] n;

integer Count;

  begin
    OutFact = 1;
    for (Count=n; Count>0; Count=Count-1)
      OutFact = OutFact * Count;
   end

endtask
```

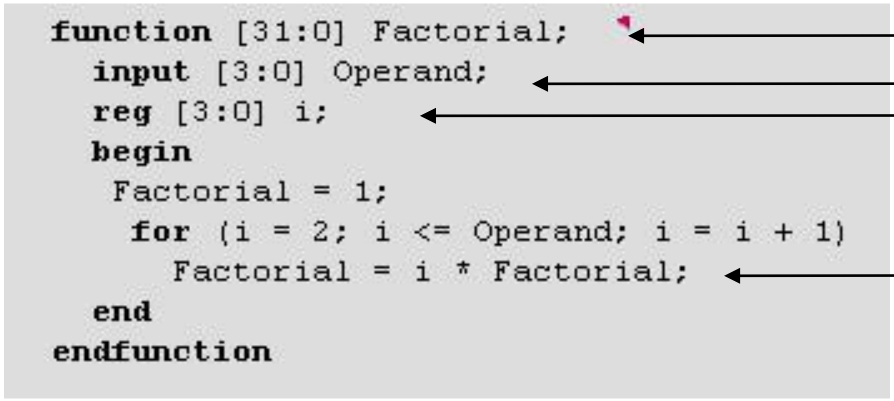
(from ALDEC tutorial)

- Task invocation

reg [31:0] result;
reg [3:0] data;
……
factorial (result, data);

Function example

- Function declaration



```
function [31:0] Factorial;
  input [3:0] Operand;
  reg [3:0] i;
  begin
   Factorial = 1;
    for (i = 2; i <= Operand; i = i + 1)
      Factorial = i * Factorial;
  end
endfunction
```

function name

input declaration

local variable

assign return value

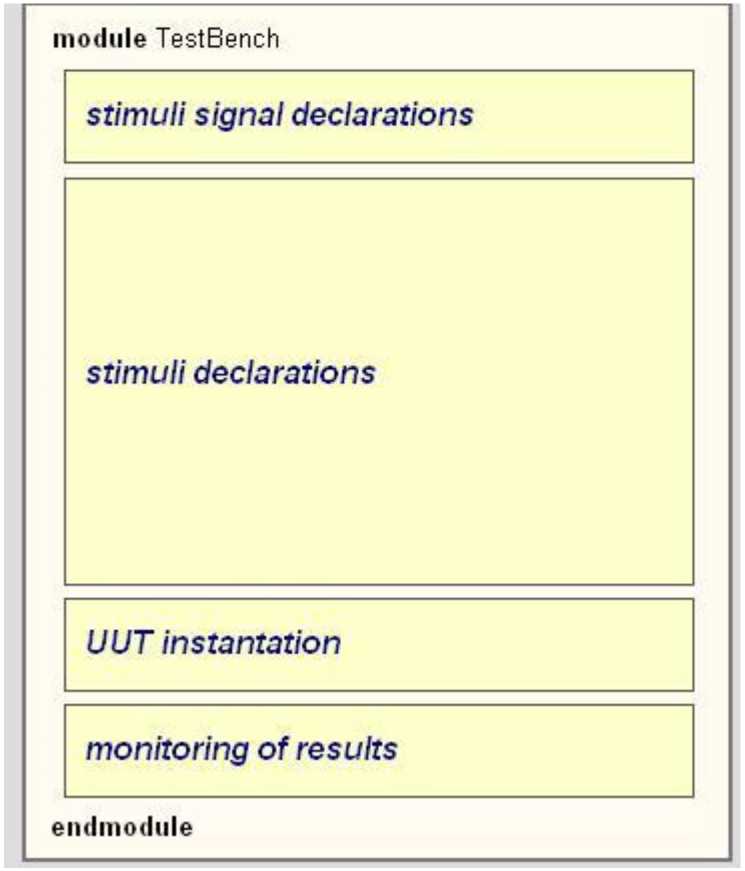
(from ALDEC tutorial)

- Function call

```
reg [31:0] result;
reg [3:0] data;
……
result = Factorial (data);
```

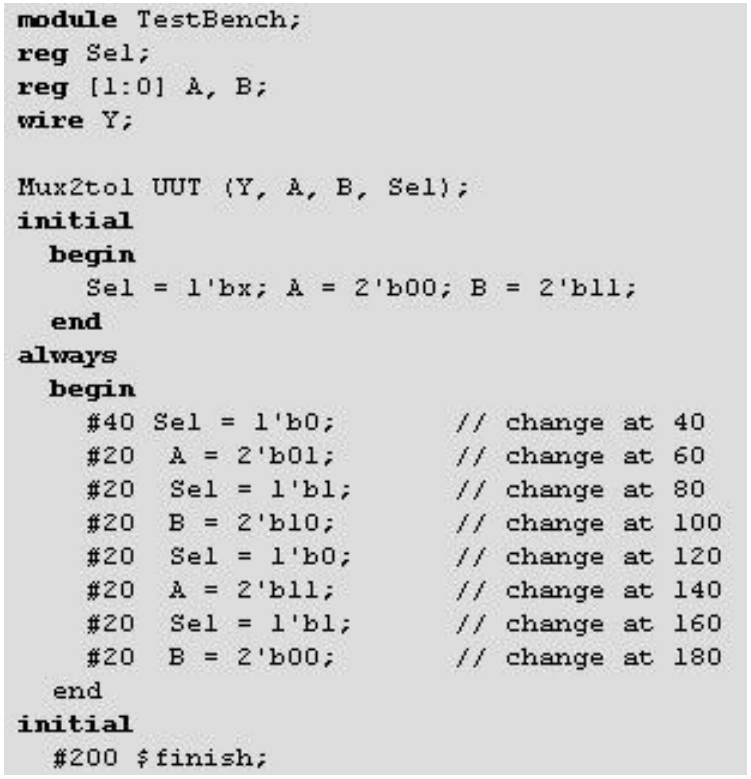
Creating testbench

- Testbench is used to verify the designed circuit.
- Most tools have template generator to aid creation process



Testbench example

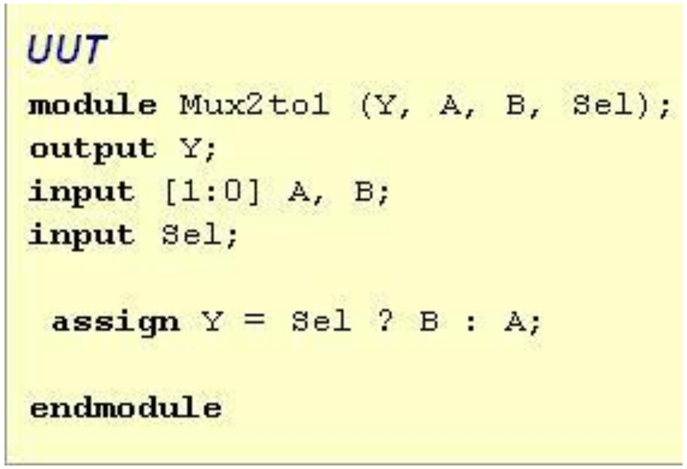
- Testbench



```
module TestBench;
reg Sel;
reg [1:0] A, B;
wire Y;

Mux2to1 UUT (Y, A, B, Sel);
initial
  begin
    Sel = 1'bx; A = 2'b00; B = 2'b11;
  end
always
  begin
    #40 Sel = 1'b0;        // change at 40
    #20  A = 2'b01;        // change at 60
    #20  Sel = 1'b1;       // change at 80
    #20  B = 2'b10;        // change at 100
    #20  Sel = 1'b0;       // change at 120
    #20  A = 2'b11;        // change at 140
    #20  Sel = 1'b1;       // change at 160
    #20  B = 2'b00;        // change at 180
  end
initial
  #200 $finish;
```

(from ALDEC tutorial)

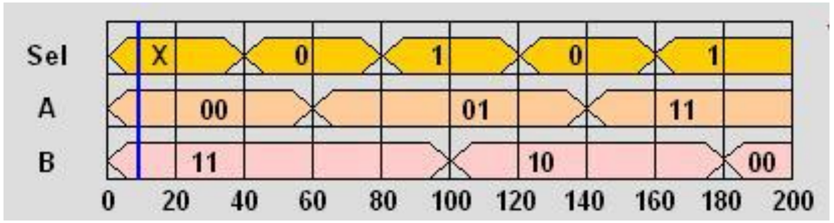
- Unit under test



UUT

```
module Mux2to1 (Y, A, B, Sel);
output Y;
input [1:0] A, B;
input Sel;

 assign Y = Sel ? B : A;

endmodule
```

- Simulation result



Some main points to remember

- Verilog is concurrent
- Think while writing your program.
- Blocking and Non-blocking Code