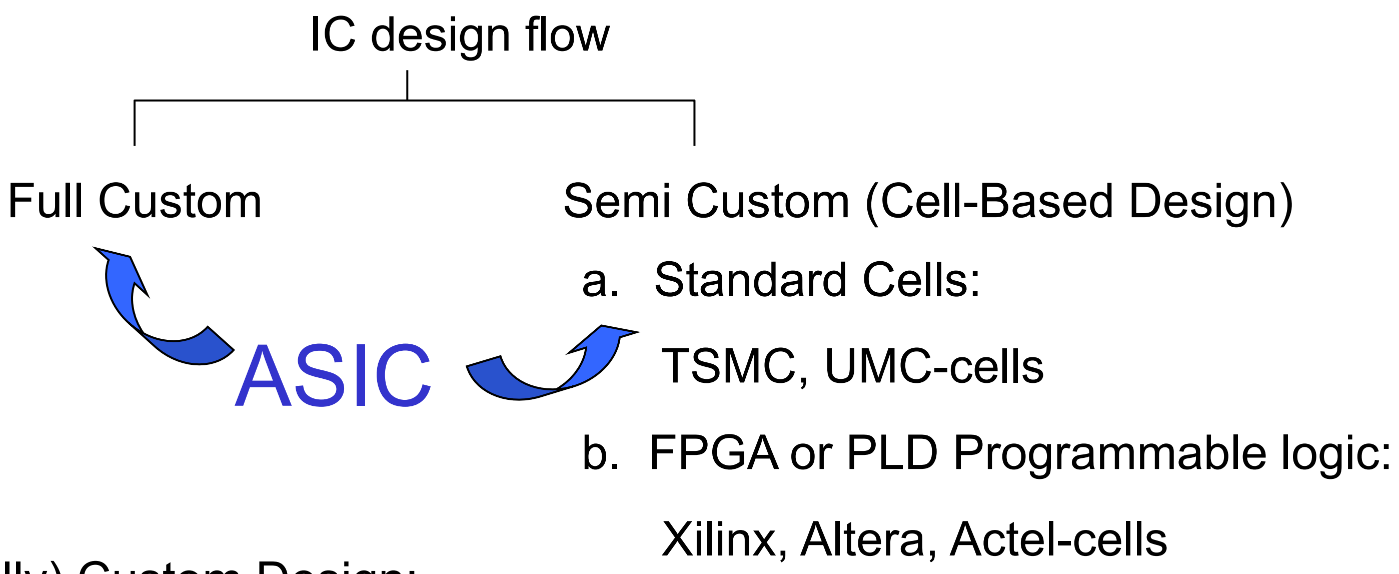
IC Design Flow and RTL Coding

Slides modified from Prof. Pei-yin Chen’s slides for Digital IC Design

IC Design flow


IC design flow
Full Custom
ASIC
Semi Custom (Cell-Based Design)
a. Standard Cells:
TSMC, UMC-cells
b. FPGA or PLD Programmable logic:
Xilinx, Altera, Actel-cells

Full (Fully) Custom Design:

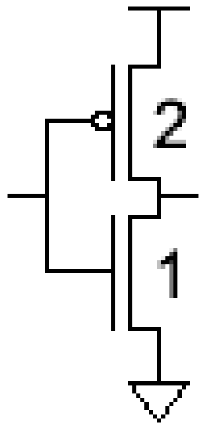
a. For analog circuits and digital circuits requiring custom optimization
b. Gates, transistors and layout are designed and optimized by the engineer

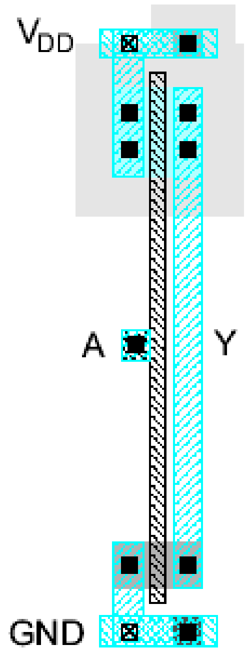
Semi Custom Design:

a. For larger digital circuits
b. Real gates, transistors and layout are synthesized and optimized by related software tools
c. Realization with hardware description language (HDL) such as VHDL and Verilog

Full Custom Design

- Digital circuits requiring custom optimization (smaller system)
- Analog circuits
- When no CPLD or FPGA solutions available
- Cons: Long design cycle (transistors and wires)





Semi Custom Design

a. Product specification

b. Modeling with HDL

c. Synthesis (by using suitable standard cell)

d. Simulation and verification

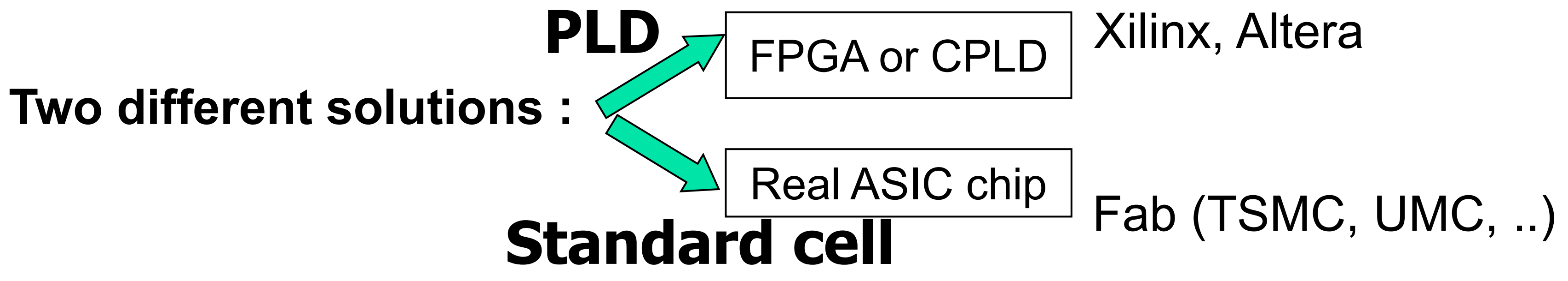
e. Physical placement and layout

-- implemented with suitable tools

f. Tape-out (real chip) -- implemented by suitable Fab companies

g. Testing -- implemented by suitable tools and mechanisms

more flexible, shorter design cycle, suitable for smaller production



less flexible, long design cycle, larger-scale production to reduce price

Standard Cells

Standard Cell

- Cells are characterized and stored in library
- Need update when technology advance
- Need technology mapping before layout for each design

Macro Cells

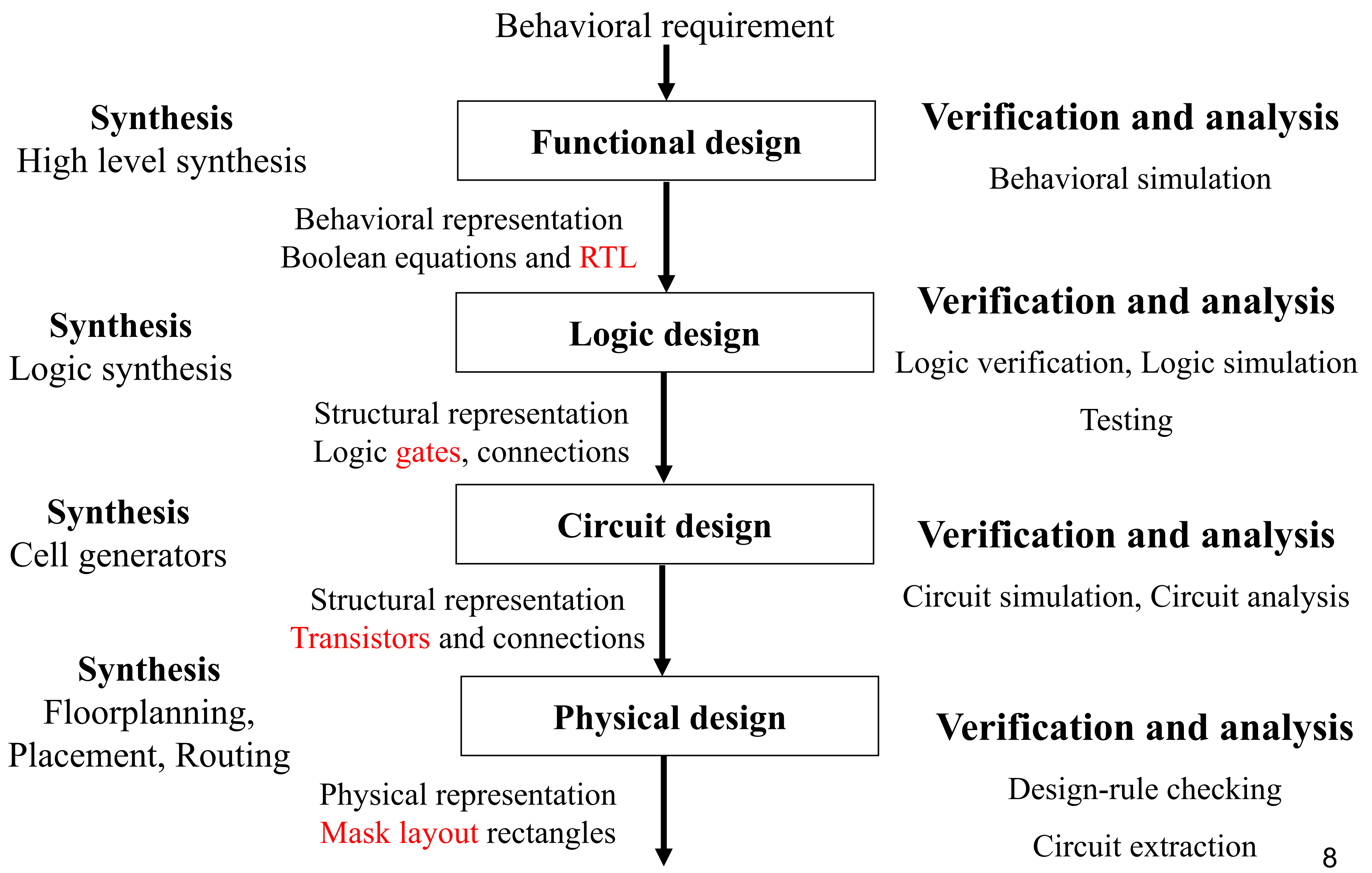
- Need parametrized capability in terms of speed and layout
- Examples : FARADAY Memory Compiler

  User Interface : memaker

  Single port RAM, Dual port RAM, ROM

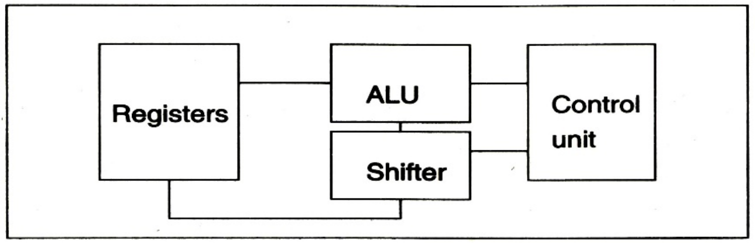
  Data sheet, Verilog simulation module, netlist simulation timing

Synthesis Flow of Semi Custom design (1/2)

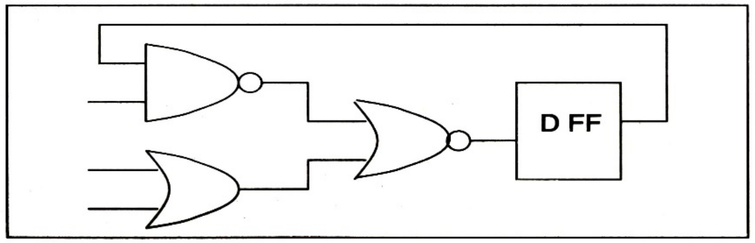


Synthesis Flow of Semi Custom design (2/2)

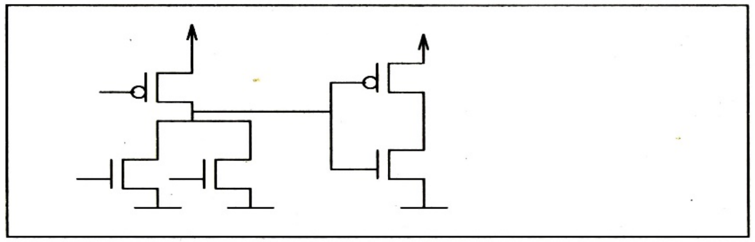
Functional design



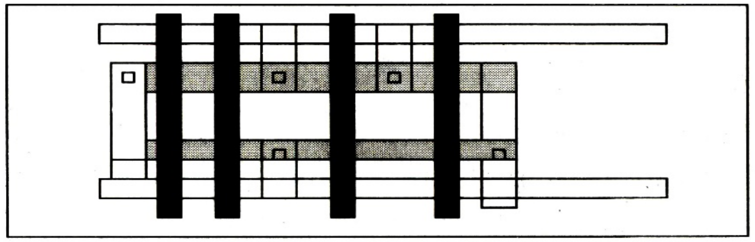
Logic design



Circuit design



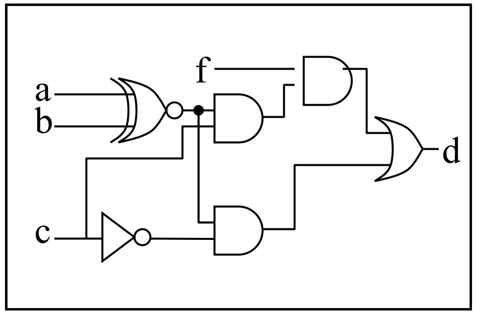
Physical design

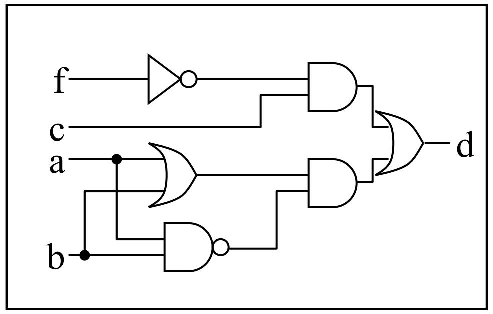


Logic Synthesis

- Synthesis =Translation + Optimization + Mapping

```
always @(…)
  if (a==b)
    if (c==1)
      d=f;
    else
      d=1;
  else
    d=0;
```



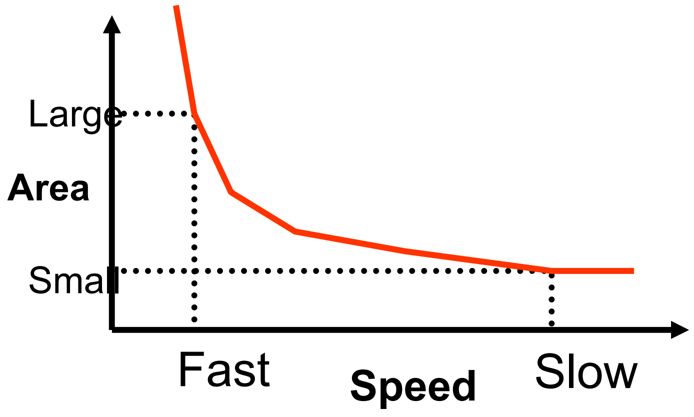


HDL Source

Translate into Boolean Representation

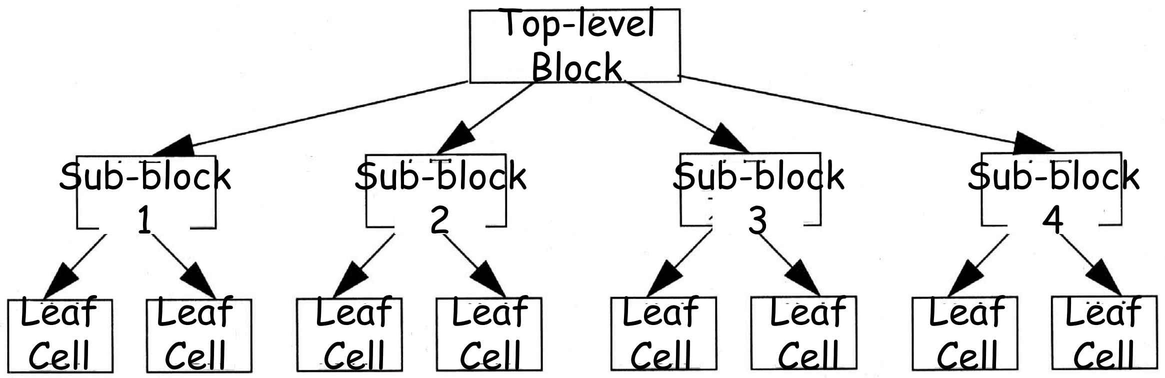
Optimize + Map for target technology

- Synthesis is **constraint-driven**
  - You set the goals. Design Compiler optimizes design toward goals.

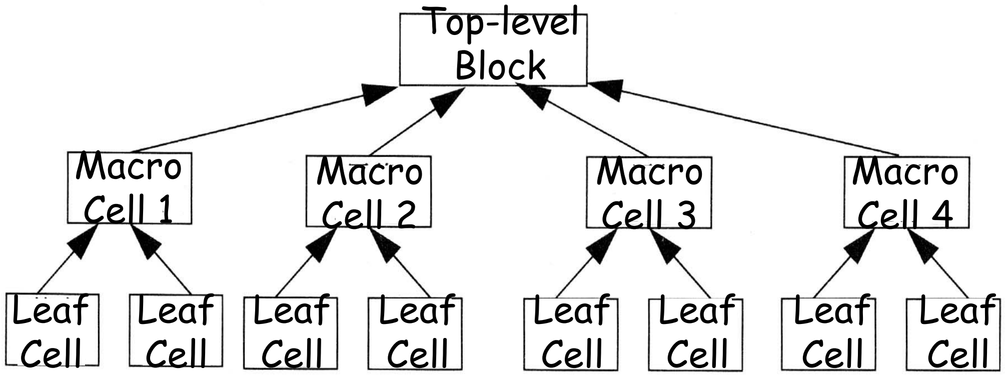


Design Methodology

- Top-down
  - Define the top-level block and identify the sub-blocks
- Bottom-up
  - Identify the building blocks first and combine the blocks to bigger blocks
- A combination of top-down and bottom-up flows is typically used.

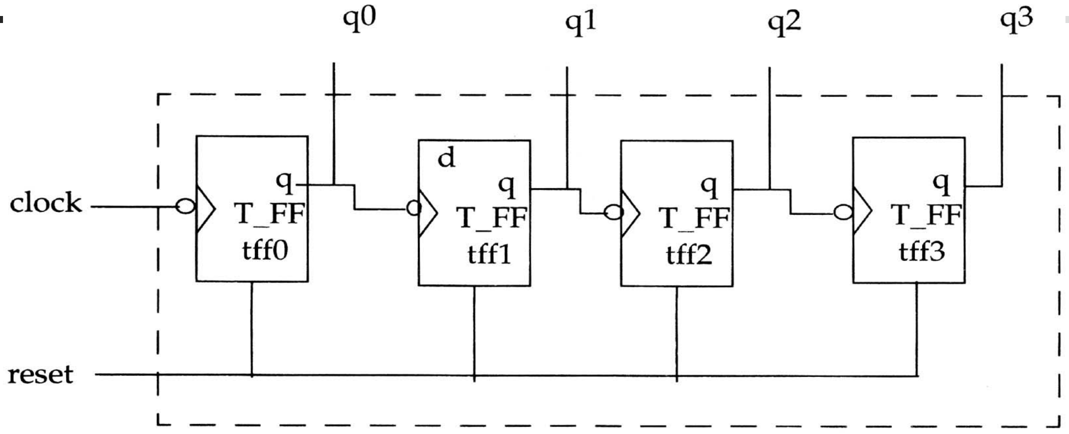


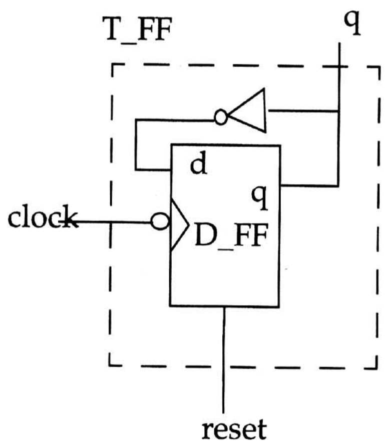
Top-down design Methodology

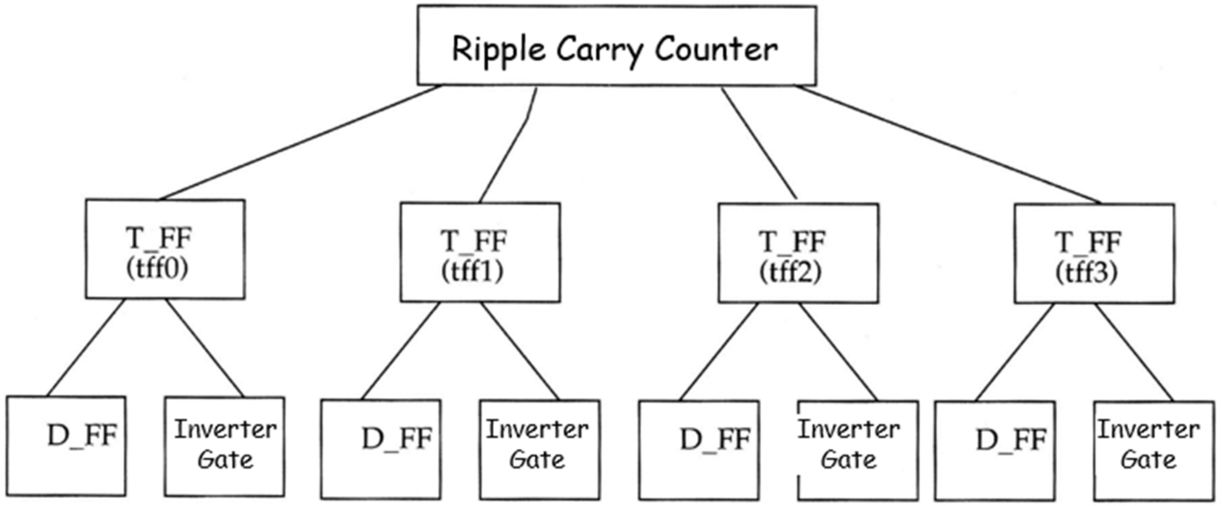


Bottom-up design Methodology

Design Hierarchy of 4-bit Ripple Carry Counter







Verilog Features

- Hardware description language allows you to describe circuit
- Procedural constructs for conditional, if-else, case and looping operations
- Arithmetic, logical, bit-wise, and reduction operations for expression
- Timing control

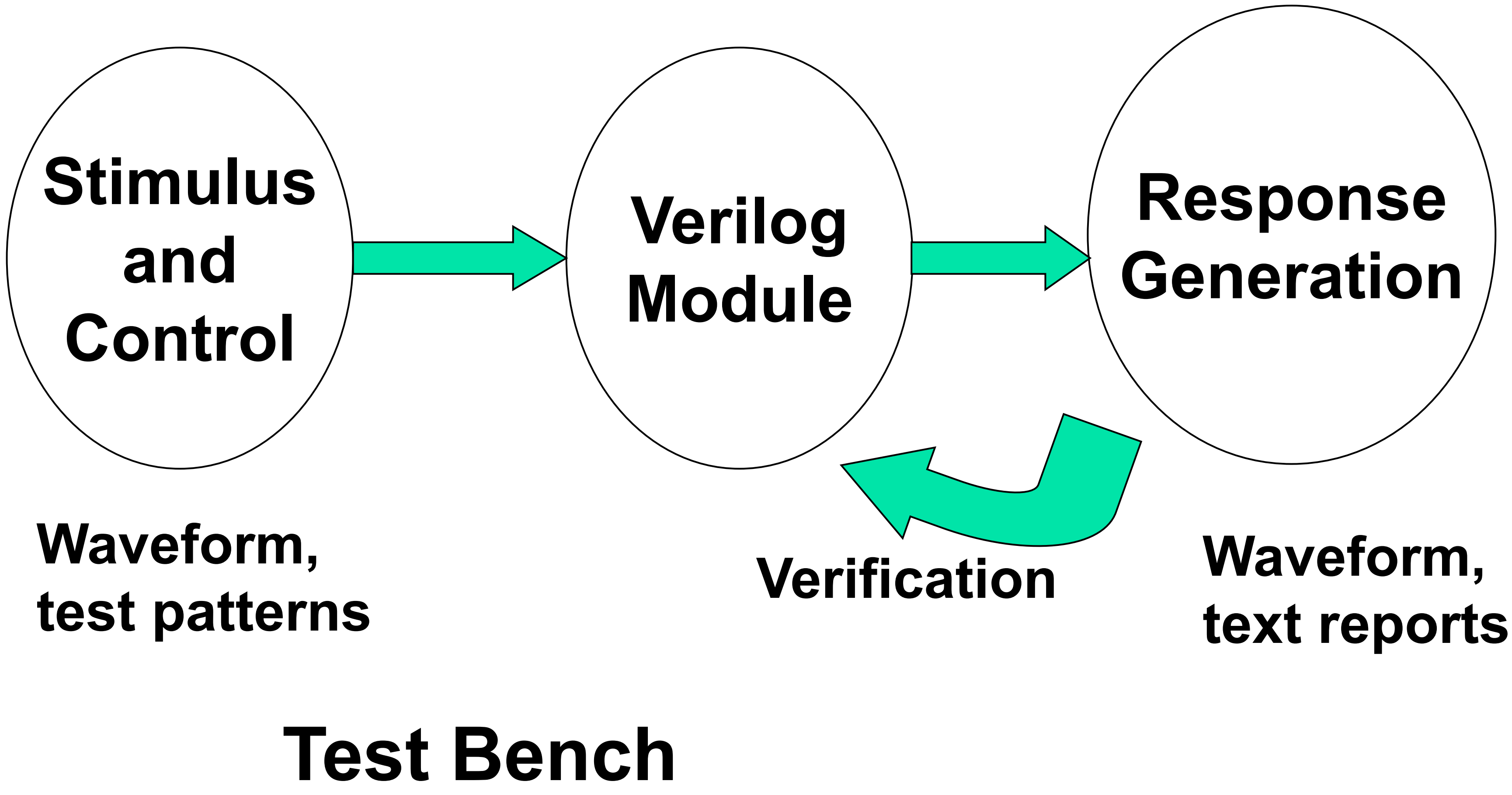
Basics of Verilog Language:

- Verilog Module	- Identifier	- Keyword
- Four Value Logic	- Data Types	- Numbers
- Port Mapping	- Operator	- Comments

Verilog Module (1/2)

Module is the basic building block

An element, or a collection of lower-level blocks

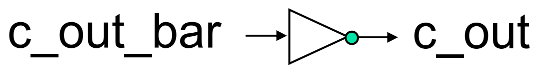


Verilog Module

Module is the basic building block

An element, or a collection of lower-level blocks

```
module module_name (port_name);
(1)  port declaration
(2)  data type declaration
(3)  module functionality or structure
endmodule
```



```
module Add_half(sum, c_out, a, b);
(1)  input          a, b;
     output   sum, c_out;

(2) wire      c_out_bar;

     xor           (sum, a, b);
(3) nand      (c_out_bar, a, b);
     not    (c_out, c_out_bar);
endmodule
```

Three Different Circuit Descriptions

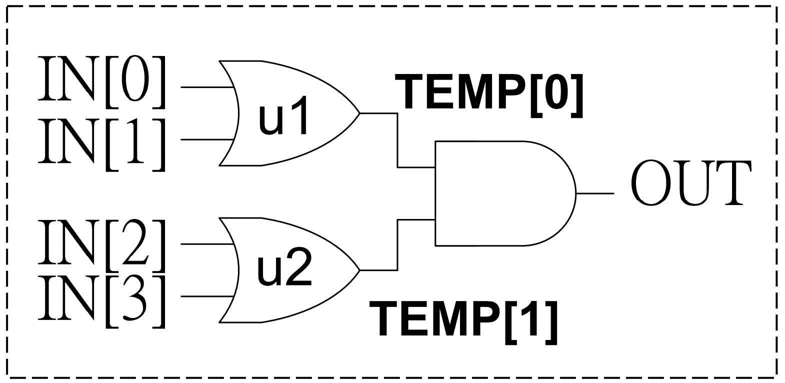
Verilog allows three kinds of descriptions for circuits:

(1) Structural description (2) Data flow description

(3) Behavioral description

Structural description:

```
module OR_AND_STRUCTURAL(IN,OUT);

input       [3:0]     IN;
output                OUT;
wire        [1:0]     TEMP;

or u1(TEMP[0], IN[0], IN[1]);
or u2(TEMP[1], IN[2], IN[3]);
and (OUT, TEMP[0], TEMP[1]);
endmodule
```



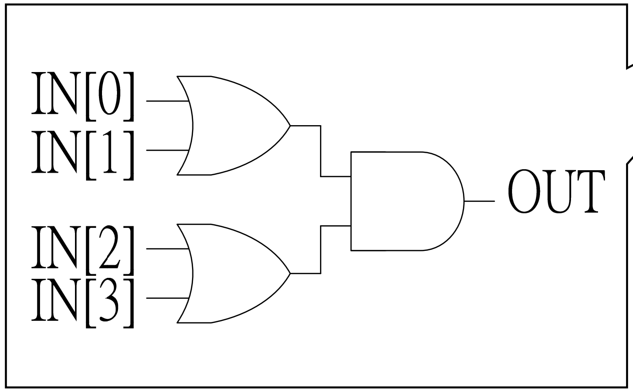
Synthesized **(synthesis) +**
optimized by tools

Data Flow Description

Data flow description

```
1.  module OR_AND_DATA_FLOW(IN, OUT);
2.  input      [3:0]      IN;
3.  output                OUT;

4.  assign OUT = (IN[0] | IN[1]) & (IN[2] | IN[3]);

    endmodule
```

Synthesized and optimized by tools



NOTE:

What is the difference between *C* and *Verilog* ?

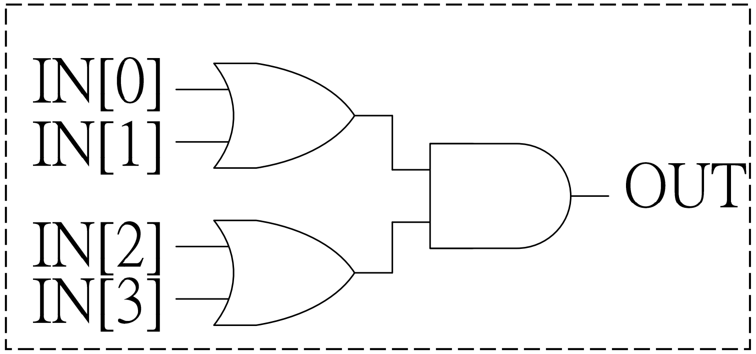
C : only one iteration (once) is implemented for assignment

Verilog : hard-wired circuit for assignment

Behavioral (RTL) Description (1/2)

Behavioral description #1

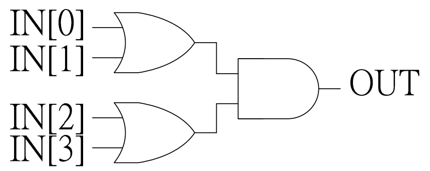
```
module OR_AND_BEHAVIORAL(IN, OUT);

input  [3:0]   IN;
output         OUT;
reg            OUT;

always @(IN)
begin
   OUT = (IN[0] | IN[1]) & (IN[2] | IN[3]);
end
endmodule
```



Activate OUT while any voltage transition

(0→1 or 1→0) happens at signal IN

Behavioral (RTL) Description (2/2)



Truth Table

IN[0]	IN[1]	IN[2]	IN[3]	OUT
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Behavioral description #2

```
module or_and(IN, OUT);
input [3:0] IN;   output   OUT;  reg   OUT;   (Note)

always @(IN)
  begin
     case(IN)
        4'b0000: OUT = 0; 4'b0001: OUT = 0;
        4'b0010: OUT = 0; 4'b0011: OUT = 0;
        4'b0100: OUT = 0; 4'b0101: OUT = 1;
        4'b0110: OUT = 1; 4'b0111: OUT = 1;
        4'b1000: OUT = 0; 4'b1001: OUT = 1;
        4'b1010: OUT = 1; 4'b1011: OUT = 1;
        4'b1100: OUT = 0; 4'b1101: OUT = 1;
        4'b1110: OUT = 1; default: OUT = 1;
      endcase
  end
endmodule
```

Synthesized and optimized by tools

Instantiation and Instances

- Instantiation: the process of creating objects from a module
  - The objects are call instances
- No nesting is allowed

```
module ripple_carry_counter(q, clk, reset);

output [3:0] q;
input clk, reset;

//4 instances of the module TFF are created.
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

endmodule
```

Create four T_FF instances

```
module TFF(q, clk, reset);
output q;
input clk, reset;
wire d;
D_FF dff0(q, d, clk, reset);
not n1(d, q); //
endmodule
```

Instance name

Create D_FF instances

Structural Description for Cell-Based Implementation

Structural description (cell-based):

```
module OR_AND_STRUCTURAL(IN,OUT);

input       [3:0]    IN;
output               OUT;
wire        [1:0]    TEMP;

orf203   u1(TEMP[0], IN[0], IN[1]);
orf203   u2(TEMP[1], IN[2], IN[3]);
andf201  (OUT, TEMP[0], TEMP[1]);
endmodule
```

Cell library from:
IBM, TSMC, UMC,..

This kind of design is not portable. Why ?

Bad design method !

Difference Between C and Verilog

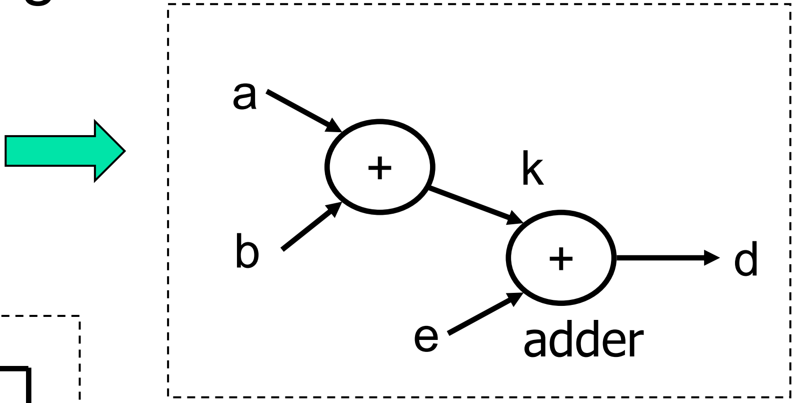
C language:

```
k=a+b;
d=k+e;
```

Only executed once

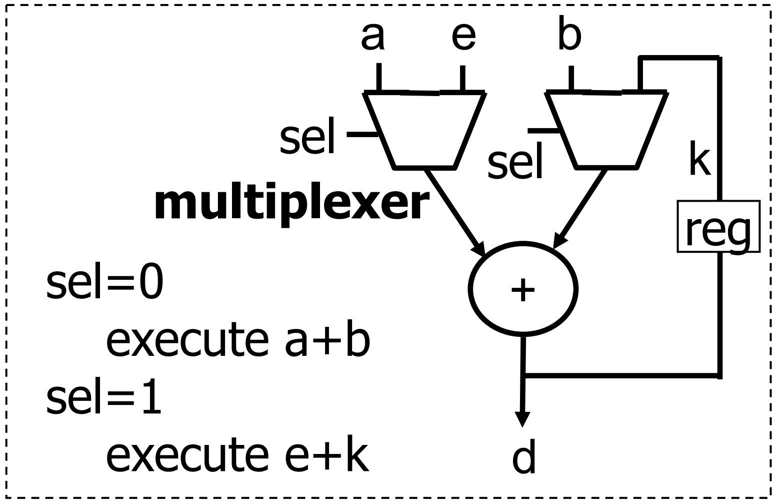
Verilog language:

```
k=a+b;
d=k+e;
```



Two adders
(higher cost, higher speed)

or



sel=0
execute a+b
sel=1
execute e+k

One adder
(lower cost, lower speed)
clock rate might be faster

In Verilog, the output is generated using hardware. As long as inputs change, the output changes

Identifier vs. Keyword

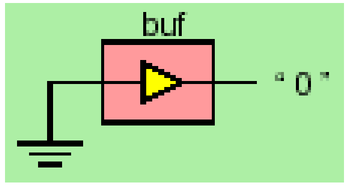
- Identifiers are names given to Verilog objects
- Names of modules, ports and instances are all identifiers
- First character must use a letter, and other character can use letter, number or "_"
- Upper case and lower case letters are different

Example:
m_y_identifier
_myidentifier$

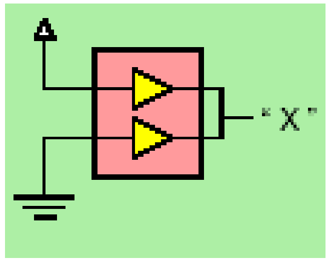
- Predefined identifiers to define the language constructs
- All keywords are defined in lower case
- Cannot be used as identifiers

Examples:
module, initial, assign,
always,
endmodule, …

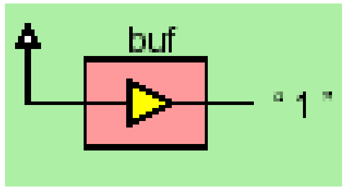
Four Value Logic



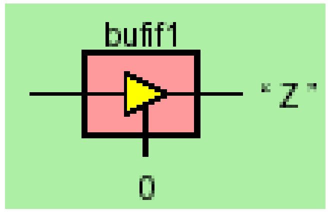
0: logic 0 / false



X: unknown logic value



1: logic 1 / true



Z: high-impedance

Number format: <size>'<radix> <value>

No of bits

Binary	→ b or B
Octal	→ o or O
Decimal	→ d or D
Hexadecimal	→ h or H

Consecutive chars 0-f, x, z

8'h ax = 1010xxxx

12'o 3zx7 = 011zzzxxx111

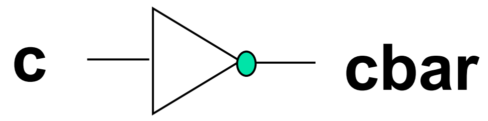
Timescale in Verilog

- The ‘timescale declares the time unit and its precision.

  ‘timescale <time_unit> / <time_precision>

  ex: ‘timescale 10 ns / 1 ns

delay= 20 ns



not #2 u1(cbar, c); **Delay=20 ns**

or #2.54 u2(TEMP[1], IN[2], IN[3]); **Delay=25 ns**

and # 3.55 (OUT, TEMP[0], TEMP[1]); **Delay=36 ns**

Data Types

- Nets
  - physical connection between devices
- Registers
  - abstract storage devices
- Parameters
  - run-time constants
- *The positions of three data types define whether they are global to a module or local to a particular always statement*
- *By default, net and register are one-bit wide (a scalar)*

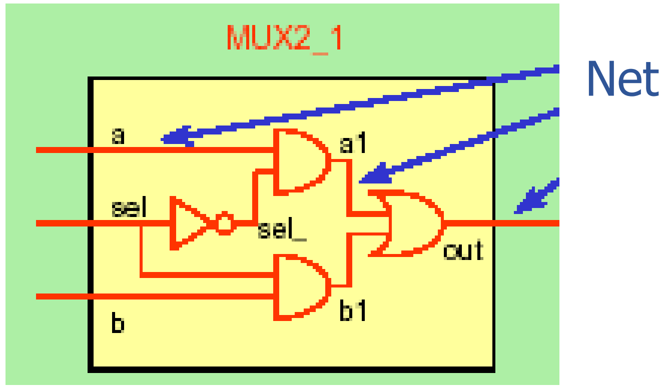
  not *multi-bit wide (a vector)*

Nets

- Connects between structural elements

  <nettype> <range>? <delay_spec>? <<net_name> <,net_name>*>

- Must be continuously driven by
  - Continuous assignment
  - Module or gate instantiation
- Default initial value for a wire is "Z"



Net Types	Functionality
wire, tri	for standard interconnection wires (default)
wor, trior	for multiple drivers that are Wire-ORed
wand, triand	for multiple drivers that are Wire-ANDed
trireg	for nets with capacitive storage
tri1	for nets which pull up when not driven
tri0	for nets which pull down when not driven
supply1	for power rails
supply0	for ground rails

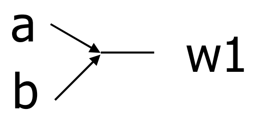
Note: Some of those net types are un-synthesizable **(不能電路合成的)**

Nets (1/2)

wire, wand, wor, tri, supply0, supply1

```
wire k;  // single-bit wire
wire [0:31] w1, w2;   // Two 32-bit wires
```

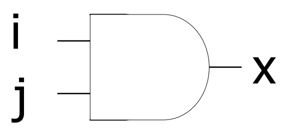
```
   wire w1;
    assign w1=a;
    assign w1=b; (error)
```



(error)

Method 1:

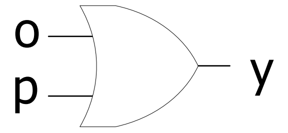
```
   wand x;
   assign x=j;     assign x=i;
```



(ok)

Method 2:

```
   wor y;
   assign y=o;   assign y=p;
```



(ok)

Nets (2/2)

tri: all variables that drive the tri must have a value of Z except one (ensured by the designer).

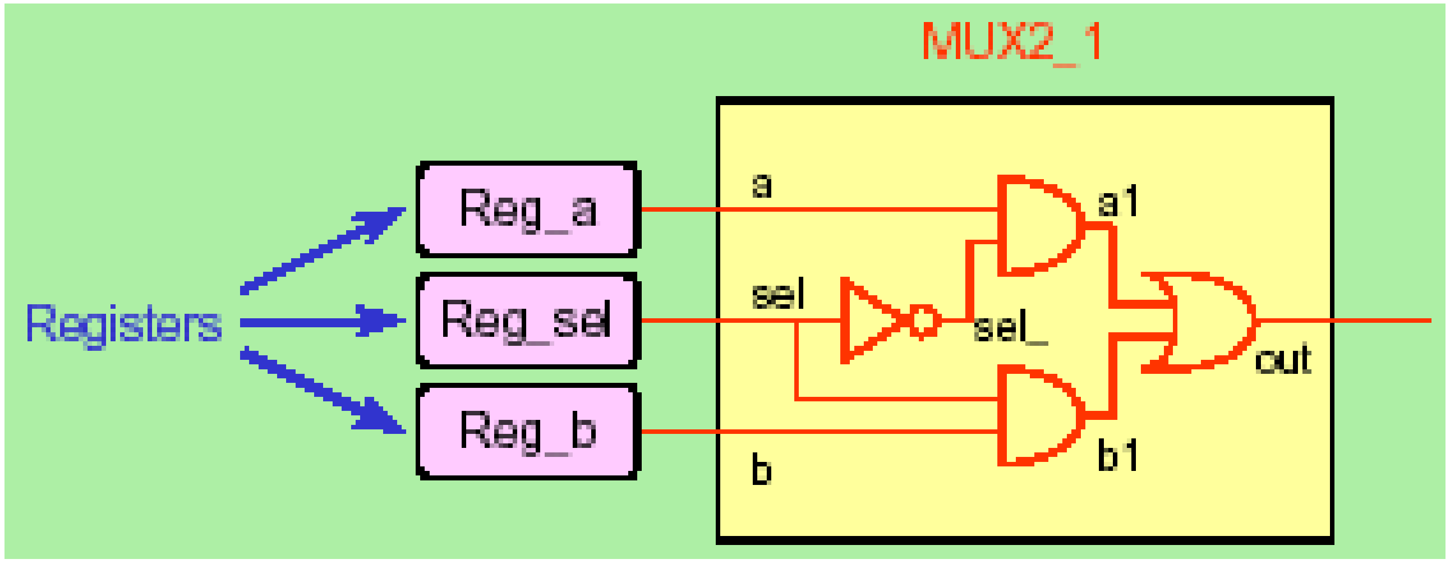
```
module tri-test(out, condition)
input [1:0] condition;  output out;
reg a, b, c;
tri out;
  always@(condition)
  begin
     a=1'bz;  b=1'bz; c=1'bz;
     case (condition)
       2'b00: a=1'b1;
       2'b01: b=1'b0;
       2'b10: c=1'b1;
     endcase
  end
assign out=a;    assign out=b; assign out=c;
endmodule
```

supply0 ⇨ **wires tied to logic 0 (ground)**

supply1 ⇨ **wires tied to logic 1 (power)**

Registers

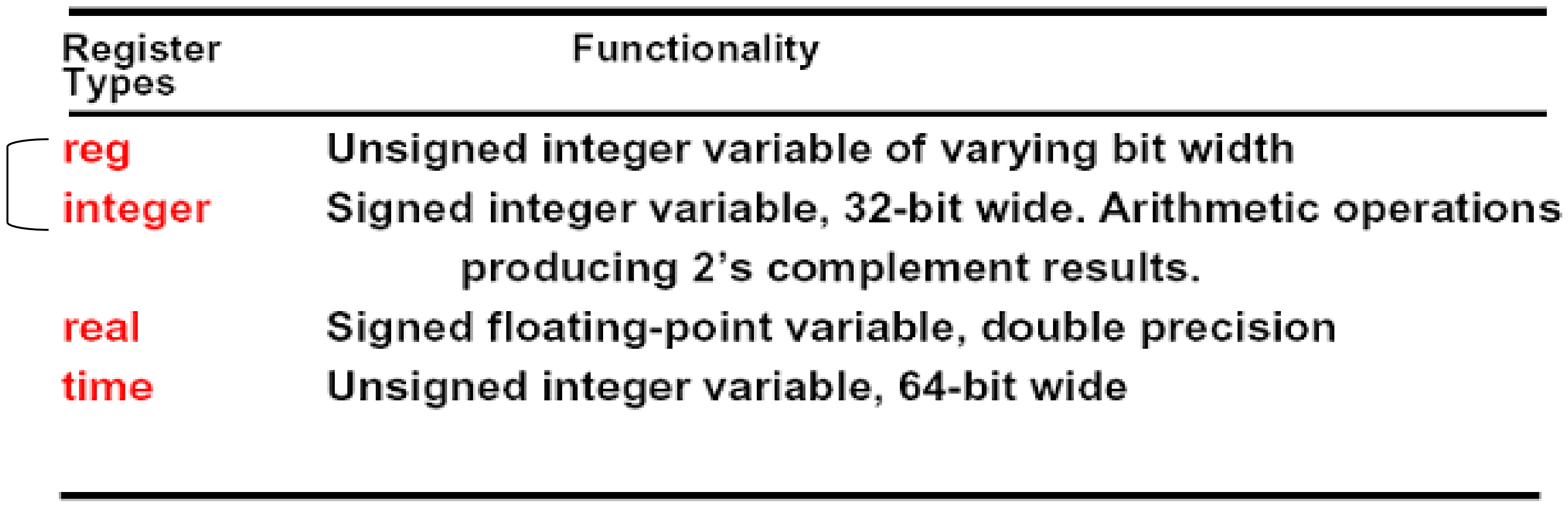
- Represent abstract data storage elements
- Hold its value until a new value is assigned to it
- Registers are used extensively in behavioral modeling
- Default initial value for a register is "X"



Types Of Registers

- Register declaration

  <register_type> <range>? <<register_name> <,register_name>*>



Register Types	Functionality
reg	Unsigned integer variable of varying bit width
integer	Signed integer variable, 32-bit wide. Arithmetic operations producing 2's complement results.
real	Signed floating-point variable, double precision
time	Unsigned integer variable, 64-bit wide

```
reg a;              // a scalar register
reg [3:0] b,c;// two 4-bit vector registers
reg [7:0] byte_reg;  // a 8-bit registers
reg  [7:0]  memory_block  [255:0];
```

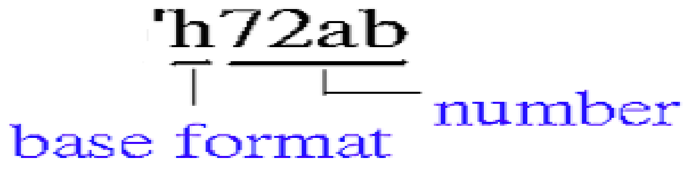
memory-block is an array of 256 registers, each of which is 8 bits width. You can access individual register, but you cannot access individual bits of register directly.

```
byte_reg=memory_block [120];
bit=byte_reg [7];            // wire bit;
```

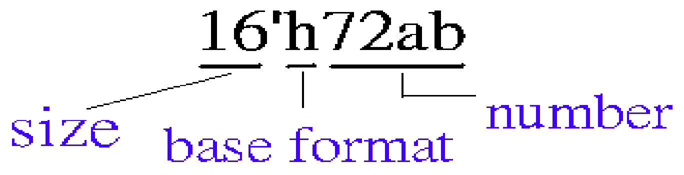
Numbers

- Numbers are integer or real constants. Integer constants are written as <size>'<base format><number>
- Real number can be represented in decimal or scientific format.
- A number may be **sized** or **unsized**
- The base format indicates the type of number
  - Decimal (d or D)
  - Hex (h or H)
  - Octal (o or O)
  - Binary (b or B)

unsize



size



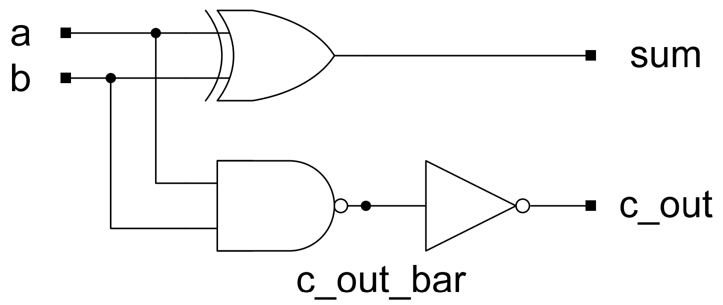
Half Adder (1/4)

a\b	0	1
0	0	1
1	1	0

sum=a⊕b

a\b	0	1
0	0	0
1	0	1

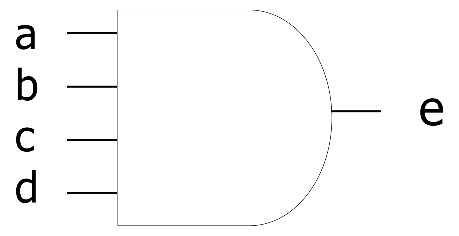
c_out=ab



Structural description

```
module Add_half(sum, c_out, a, b);
   input        a, b;
   output       sum, c_out;
   wire         c_out_bar;

   xor          (sum, a, b);
   nand         (c_out_bar, a, b);
   not          (c_out, c_out_bar);
endmodule
```

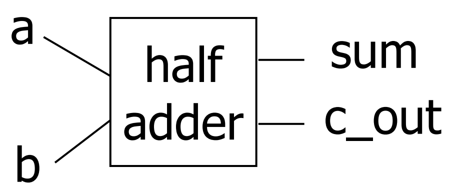
```
and (e, a, b,c,d);
```



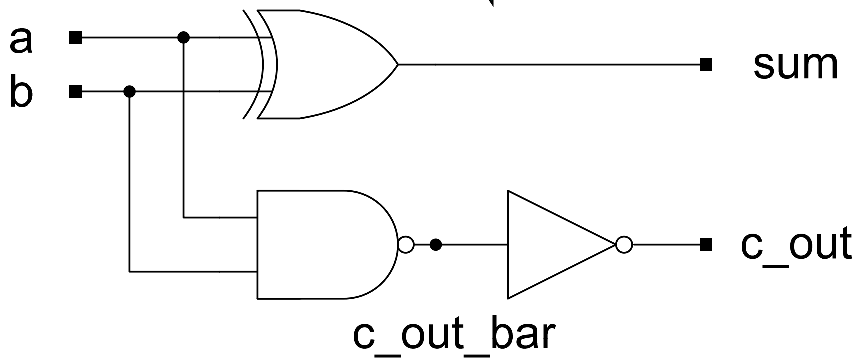
Half Adder (2/4)

Data flow description

```
module Add_half(sum, c_out, a, b);
   input      a, b;
   output     sum, c_out;

   assign     {c_out, sum} = a + b;
endmodule
```



Synthesized and optimized by tools

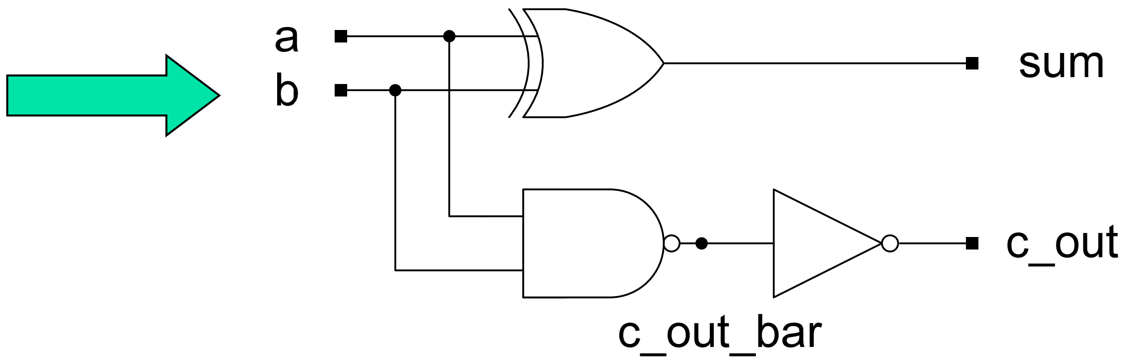


assign: continuous assignment

Half Adder (3/4)

Behavioral description #1

```
module Add_half(sum, c_out, a, b);
   input      a, b;
   output     sum, c_out;
   reg        sum, c_out;

   always @ (a or b)
     begin
       sum = a ^ b;
       c_out = a & b;
     end
endmodule
```



Half Adder (4/4)

Behavioral description #2

```
module Add_half(sum, c_out, a, b);
input  a, b;
output sum, c_out;
reg    sum, c_out;
always @(a or b)
begin
     case({a,b})
        2'b00:begin
              sum = 0;c_out = 0;
              end
        2'b01:begin
              sum = 1; c_out = 0;
              end
        2'b10:begin
              sum = 1;  c_out = 0;
              end
        default:begin
              sum = 0; c_out = 1;
              end
      endcase
end
endmodule
```

a\b	0	1
0	0	1
1	1	0

sum

a\b	0	1
0	0	0
1	0	1

c_out

Parameter

- Parameter declaration

  parameter<range>?<list_of_assignments>

- You can use a parameter anywhere that you can use a literal.

ex:
```
module mod(ina, inb,out);
    ........
    parameter m1=8;
    ....
     wire [m1:0]  w1;
    ........
endmodule
```

w1 can be set as a (n+1)-bit wire if we change m1 to n

(i.e., m1=10 ⇨ w1 becomes a 11-bit wire

m1=4 ⇨ w1 becomes a 5-bit wire)

Parameterized Design

```
module PARAM(A , B , C);
input [7 : 0] A , B;
output [7 : 0] C;
wire f;
    or     o1(f,A,B);
    test  #(4)  u1(A , f , C);

endmodule
```

```
module test (a , b , c);
parameter width = 8;
input [width - 1 : 0]  a, b;
output [width - 1 : 0] c;

    assign c = a & b;

endmodule
```

Override the value of width when the test module is instantiated

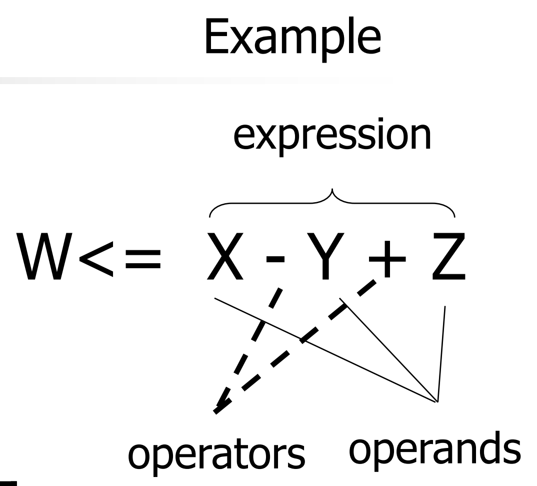
Save the file as PARAM.v and compile (synthesis) it

➡ **the width value become 4**

Expressions

- An expression comprises of operators and operands, see Example, and are covered separately in the following two sections.
- Operands: Four data objects

Verilog Operands	
Identifiers (mentioned before)	
Literals	
string (bit & character)	4'b 1101
numeric (integer, real⁺)	34
Function Call	
Index & Slice Names	



Identifier and Literal Operands

```
module LITERALS(A1, A2, B1, B2, Y1, Y2);
        input     A1, A2, B1, B2;
        output [7:0]  Y1;  output [5:0] Y2;
        parameter CST = 4'b 1010, TF=25;
        reg [7:0] Y1; reg [5:0] Y2;

 always @(A1 or A2 or B1 or B2)
 begin
        if (A1 == 1)
             Y1 = {CST, 4'b 0000};
        else if (A2 == 1)
             Y1 = {CST, 4'b 0101};
        else
             Y1 = {CST, 4'b 1111};
        if (B1 == 0)    Y2 = 10;
        else if (B2 == 1) Y2 = 15;
        else    Y2 = TF +10 +15;
  end
endmodule
```

Identifier

Bit string literals

Numeric (integer) literal

Identifier

Function Call Operands

- Function calls, which must reside in an expression, are operands. The single value returned from a function is the operand value used in the expression.

```
module FUNCTION_CALLS (A1, A2, A3, A4, B1, B2, B3, B4, Y1, Y2);
    input     A1, A2, A3, A4, B1, B2, B3,B4;   output [2:0] Y1, Y2;
    reg  [2:0] Y1, Y2;

 function [2:0] Fn1;
    input     F1, F2, F3;
    begin
        Fn1 = F1+F2+F3;
    end
 endfunction

  always @(A1 or A2 or A3 or A4 or B1 or B2 or B3 or B4)
    begin
    Y1 = Fn1(A1, A2, A3)+A4;
    Y2 = Fn1(B1, B2, B3)-B4;
    end
endmodule
```

```
function [2:0] Fn1;
  input  F1, F2, F3;
  begin
     Fn1 = F1+F2+F3;
  end
  endfunction
```

Function call operand

Index and Slice Name Operands (1/2)

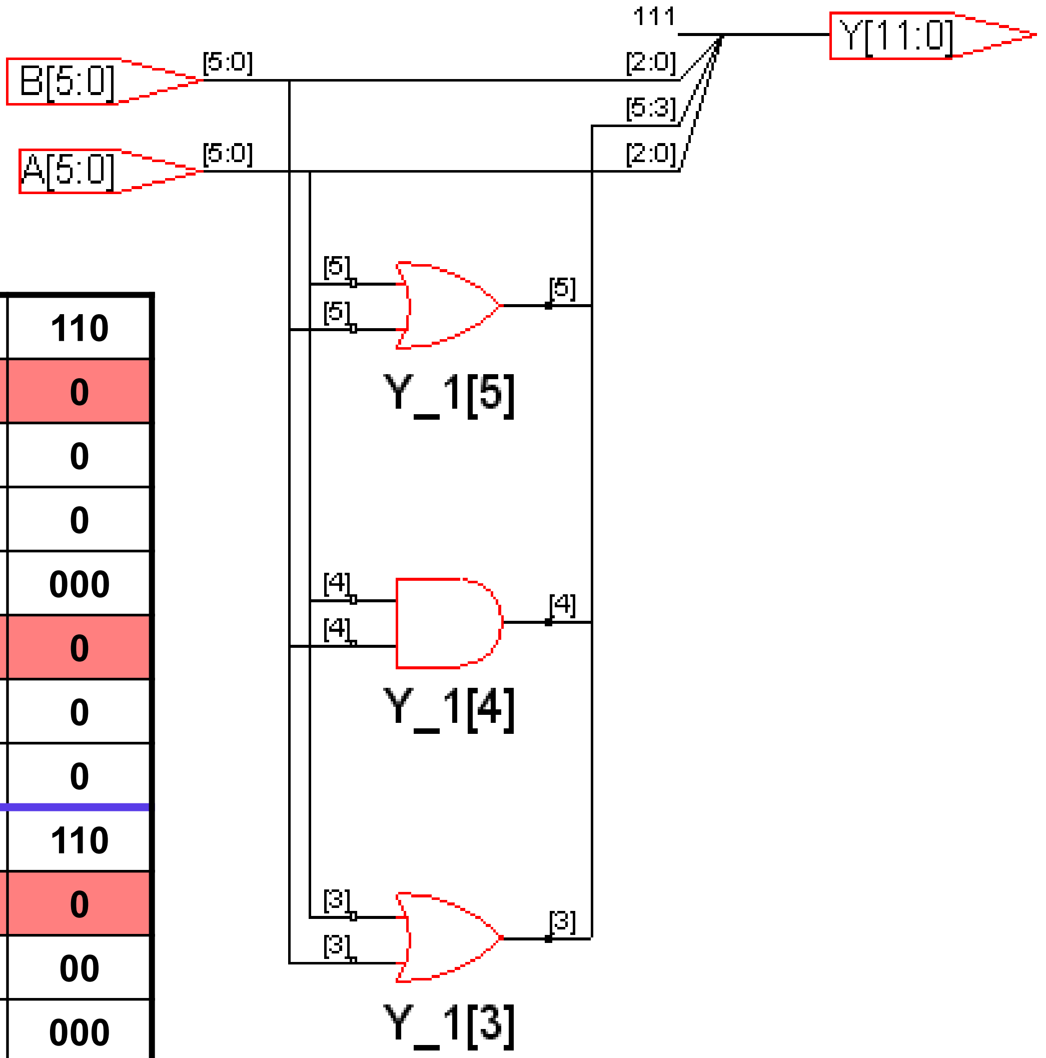
- Index operand specifies a single element of an array and slice operand specifies a sequence of elements within an array

```
module INDEX_SLICE_NAME (A, B, Y);
        input [5:0]          A, B;
        output [11:0]Y;
        parameter            C = 3'b111;
        reg [11:0]           Y;

 always @(A or B)
 begin
        Y[2:0]    = A[2:0];
        Y[3]      = A[3] | B[3];
        Y[5:4]    = {A[5] | B[5], A[4] & B[4]};
        Y[8:6]    = B[2:0];
        Y[11:9]   = C;
 end
endmodule
```

Index and Slice Name Operands (2/2)

```
parameter        C = 3'b111;
Y[2:0]    = A[2:0];   Y[3]= A[3] | B[3];
Y[5:4]    = {A[5] | B[5], A[4] & B[4]};
Y[8:6]    = B[2:0];   Y[11:9]= C;
```

A[2:0]	000	010	100	101	110
A[3]	0	1	0	1	0
A[4]	0	1	0	1	0
A[5]	0	1	0	1	0
B[2:0]	110	111	101	001	000
B[3]	0	0	1	1	0
B[4]	0	0	1	1	0
B[5]	0	0	1	1	0
Y[0:2]	000	010	100	101	110
Y[3]	0	1	1	1	0
Y[5:4]	00	10	10	11	00
Y[8:6]	110	111	101	001	000
Y[11:9]	111	111	111	111	111



Operators (1/2)

- Operators perform an operation on one or more operands within an expression. An expression combines operands with appropriate operators to produce the desired function expression.

Name	Operator
bit-select or part-select	[ ]
parenthesis	( )
Arithmetic Operators	
multiplication	*
division	/
addition	+
subtraction	-
modulus	%
Sign Operators	
identity	+
negation	-

It is not recommended to use the following three operators:

*
/ : (not synthesizable)
% : (not synthesizable)

Operators (2/2)

Name	Operator
Relational Operators	
less than	<
less than or equal to	<=
greater than	>
greater than or equal to	>=
Equality Operators	
logic equality	==
logic inequality	!=
case equality	===
case inequality	!==
Logical Comparison Operators	
NOT	!
AND	&&
OR	||
Logical Bit-Wise Operators	
unary negation NOT	~
binary AND	&
binary OR	|
binary XOR	^
binary XNOR	^~ or ~^

Name	Operator
Shift Operators	
logical shift left	<<
logical shift right	>>
Concatenation & Replication Operators	
concatenation	{ }
replication	{{ }}
Reduction Operators	
AND	&
OR	|
NAND	~&
NOR	~|
XOR	^
XNOR	^~ or ~^
Conditional Operator	
conditional	?:

Relational Operators

```
module RELATIONAL_OPERATORS(A, B, Y);
      input [2:0]          A;
      input [2:0]          B;
      output [3:0]         Y;
      reg [3:0] Y;

 always @(A or B)
 begin
      Y[0] = A > B;
      Y[1] = A >= B;
      Y[2] = A < B;
      if ( A <= B)
         Y[3] = 1;
      else
         Y[3] = 0;
 end
endmodule
```

Relational operators:
(1) <
(2) <=
(3) >=
(4) >

A[2:0]	1	2	3	4	5	6	7	3	0
B[2:0]	3	5	6	7	2	1	0	3	6
Y[3]	1	1	1	1	0	0	0	1	1
Y[2]	1	1	1	1	0	0	0	0	1
Y[1]	0	0	0	0	1	1	1	1	0
Y[0]	0	0	0	0	1	1	1	0	0

Equality & Inequality Operators

```
module EQUALITY_OPERATORS(A, B, Y);
      input [2:0]        A;
      input [2:0]        B;
      output [4:0]       Y;
      reg [4:0] Y;

 always @(A or B)
 begin
      Y[0] = A != B;
      Y[1] = A == B;
      if ( A == B)
          Y[4:2] = A;
      else
          Y[4:2] = B;
end

endmodule
```

Equality operators:

(1) ==　　(等於)
(2) !=　　(不等於)
(3) ===　　(++) non-syn.
(4) !==　　(++) non-syn.

When comparison is true, the result is 1

When comparison is false, the result is 0

A[2:0]	0	7	4	1	0
B[2:0]	3	7	3	4	5
Y[0]	1	0	1	1	1
Y[1]	0	1	0	0	0
Y[4:2]	3	7	3	4	5

Logical Bit-Wise Operators

```
module BITWISE(A, B, Y1, Y2, Y3, Y4, Y5);
    input [3:0]    A;
    input [3:0]    B;
    output [3:0]  Y1, Y2, Y3, Y4, Y5;

    reg [3:0]       Y1, Y2, Y3, Y4, Y5;

 always @(A or B)
 begin
  Y1 =   ~A;
  Y2 =  A & B;
  Y3 =  A | B ;
  Y4 =  A ^ B;
  Y5 =  A ^ ~ B;
 end
endmodule
```

Logic bit-wise operators:

(1) ~	(unary NOT)
(2) &	(binary AND)
(3) |	(binary OR)
(4) ^	(binary XOR)
(5) ^~ or ~^	(binary XNOR)

```
   0110
&  0011
   0010
```

```
   0110
|  0011
   0111
```

A[3:0]	0000	0001	0010	0110
B[3:0]	0000	0000	0001	0011
Y1[3:0]	1111	1110	1101	1001
Y2[3:0]	0000	0000	0000	0010
Y3[3:0]	0000	0001	0011	0111
Y4[3:0]	0000	0001	0011	0101
Y5[3:0]	1111	1110	1100	1010

Concatenation & Replication Operators

```
module CONCATENATION (A, B, Y);
    input [2:0]      A;
    input [2:0]      B;
    output [14:0]    Y;

    reg [14:0]       Y;

    parameter        C=3'b011;

 always @(A)
 begin
    Y = { B, A, { 2 { C } }, 3'b 001};
end
endmodule
```

3+3+2*3+3=15

Concatenation	{}
Replication	{{}}

A[2:0]	000	001	010	011
B[2:0]	000	010	100	110
Y[14:0]	000 000 011 011 001	010 001 011 011 001	100 010 011 011 001	110 011 011 011 001

Reduction Operators

```
module REDUCTION (A, Y);
        input [3:0] A;
        output [5:0]          Y;

        reg [5:0]   Y;

 always @(A)
 begin
  Y[0] = & A;
  Y[1] = | A;
  Y[2] = ~& A;
  Y[3] = ~| A;
  Y[4] =  ^ A;   // XOR,奇同位
  Y[5] = ~^ A;  //XNOR,偶同位
end
endmodule
```

Reduction operators:
(1) &   (2) |
(3)~&   (4) ~|
(5) ^   (6) ~^

& A $\Rightarrow$ A[0] & A[1] & A[2] & A[3]

| A $\Rightarrow$ A[0] | A[1] | A[2] | A[3]

^ A $\Rightarrow$ A[0] ^ A[1] ^ A[2] ^ A[3]

A[3]	0	0	0	0
A[2]	0	0	0	0
A[1]	0	0	1	1
A[0]	0	1	0	1
Y[5:0]	101100	010110	010110	100110

Conditional Operators

```
module ADD_SUB (A, B, SEL, Y);
input [7:0] A;   input [7:0]      B;
input                         SEL;
output [8:0]                  Y1,Y2;
reg [8:0]    Y2,Y1;
  always @( A or B)
   begin
    Y1 = ( SEL == 1 ) ? A + B :  A – B ;
    Y2 = (!SEL) ? A : B;
   end
endmodule
```

Conditional operators:

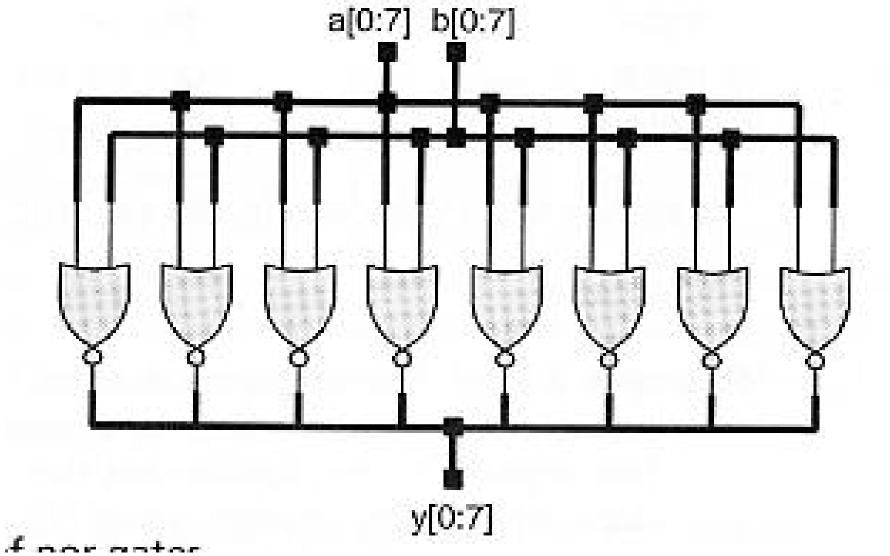
? :

SEL	0	0	1	1
A[7:0]	00000100	00000001	00000100	00000001
B[7:0]	00000001	00000010	00000001	00000010
Y1[8:0]	00000011	11111111	00000101	00000011
Y2[8:0]	00000100	00000001	00000001	00000010

Array of Instances

```
module array_of_nor(y, a, b);
   input        [0:7]    a, b;
   output       [0:7]    y;

   nor          (y[0], a[0], b[0]);
   nor          (y[1], a[1], b[1]);
   nor          (y[2], a[2], b[2]);
   nor          (y[3], a[3], b[3]);
   nor          (y[4], a[4], b[4]);
   nor          (y[5], a[5], b[5]);
   nor          (y[6], a[6], b[6]);
   nor          (y[7], a[7], b[7]);
endmodule
```

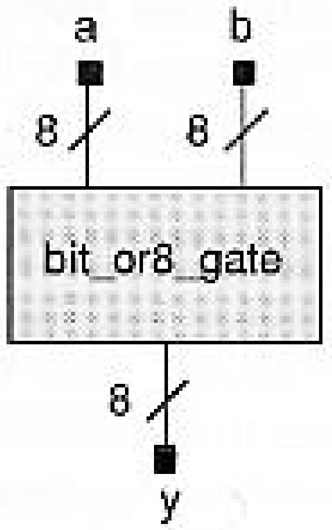
```
module array_of_nor(y, a, b);
   input        [0:7]    a, b;
   output       [0:7]    y;

   nor          [0:7]    (y, a, b);
endmodule
```

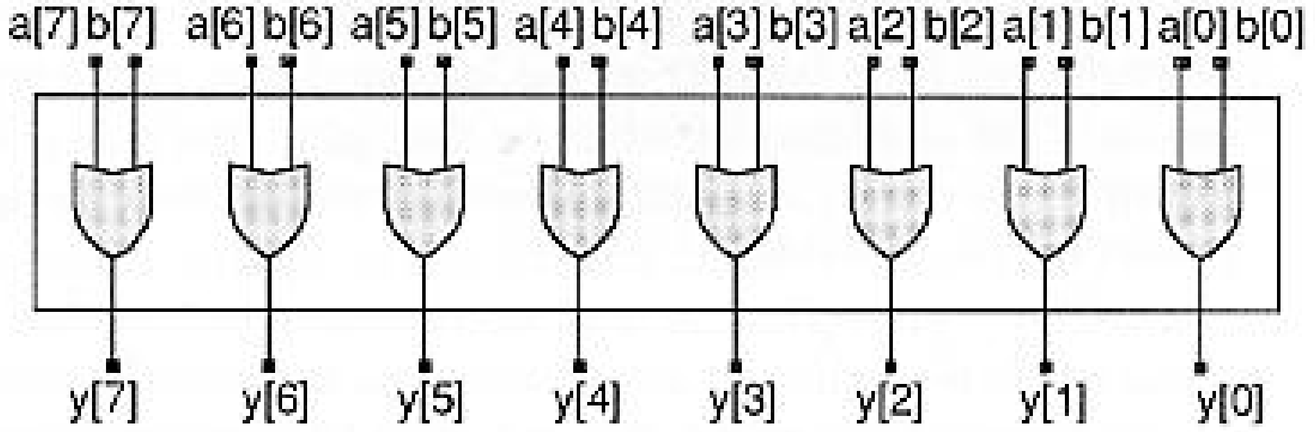


Two alternatives for Continuous Assignment

```
module bit_or8_gate1(y, a, b);

    input        [7:0] a, b;
    output       [7:0] y;

    assign       y = a | b;
endmodule
```

```
module bit_or8_gate2(y, a, b);
          input  [7:0] a, b;
         output  [7:0] y;
         wire    [7:0] y = a | b;
endmodule
```

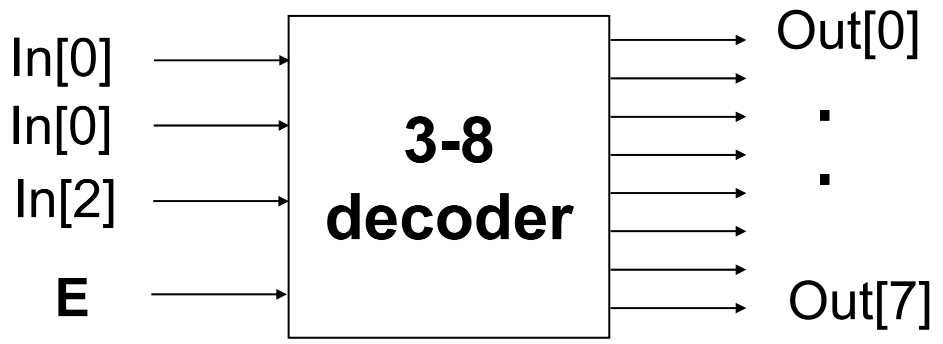



a[7] b[7] a[6] b[6] a[5] b[5] a[4] b[4] a[3] b[3] a[2] b[2] a[1] b[1] a[0] b[0]
y[7] y[6] y[5] y[4] y[3] y[2] y[1] y[0]

Multiple Instantiations and Assignments

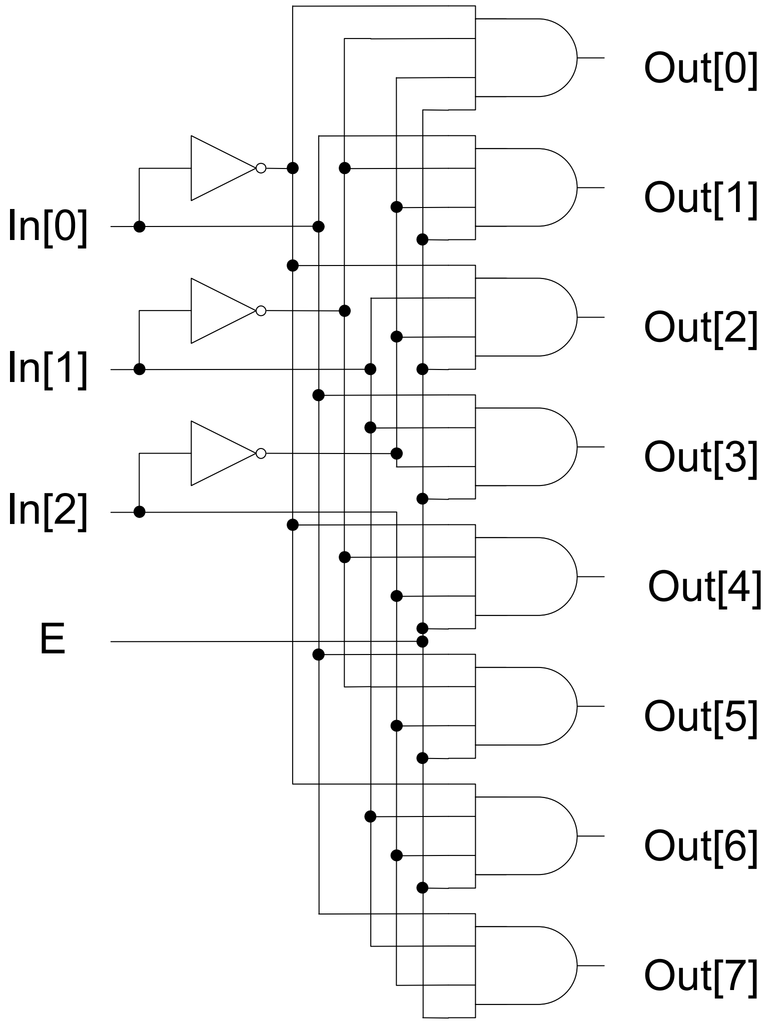
```
module Multiple_Gates(y1, y2, y3, a1, a2, a3, a4);
   input       a1, a2, a3, a4;
   output      y1, y2, y3;

nand #1 G1(y1, a1, a2, a3), (y2, a2, a3, a4), (y3, a1, a4);
endmodule


module Multiple_Assigns(y1, y2, y3, a1, a2, a3, a4);
   input       a1, a2, a3, a4;
   output      y1, y2, y3;

   assign #1    y1 = a1 ^ a2, y2 = a2 | a3, y3 = a1 + a2;
endmodule
```

3 to 8 Decoder (1/4)



E	In[2]	In[1]	In[0]	Out[7]	Out[6]	Out[5]	Out[4]	Out[3]	Out[2]	Out[1]	Out[0]
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

3 to 8 Decoder (2/4)



Structural description

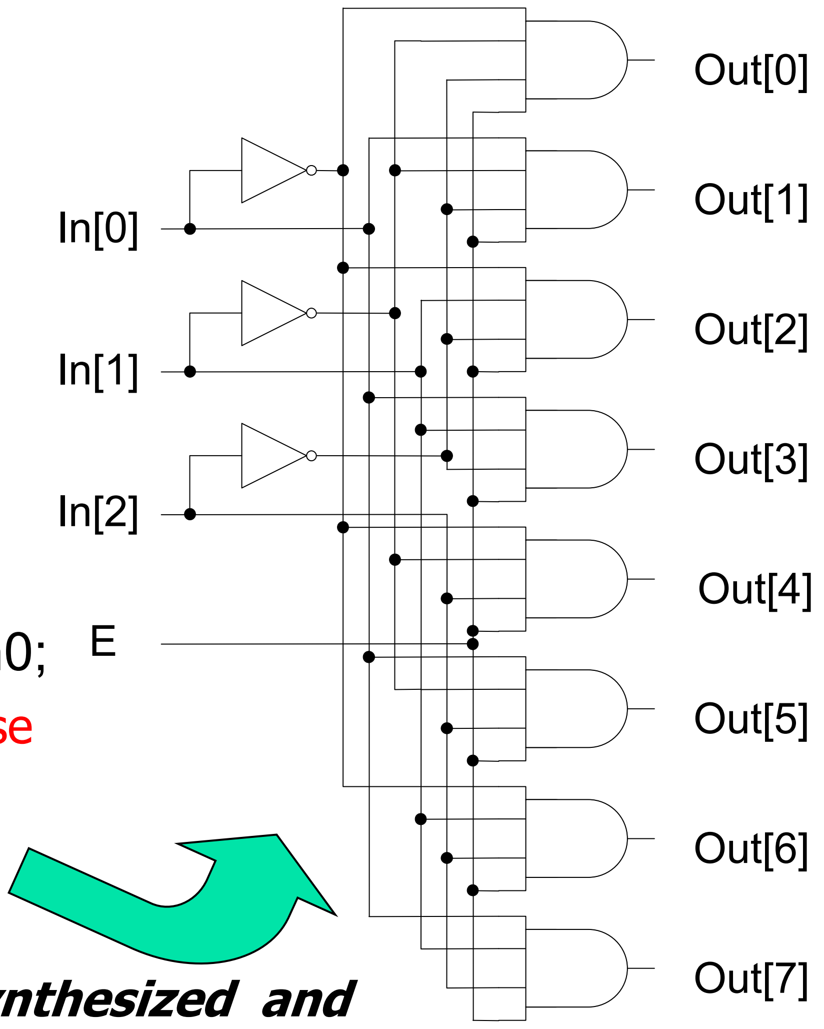
```
module decoder (E , In , Out);
input E;   input [2:0] In;
output [7:0] Out;
wire [7:0] Out;
wire tmp0 , tmp1 , tmp2;

not not1(tmp0,In[0]);  not not2(tmp1,In[1]);
not not3(tmp2,In[2]);
and and0(Out[0] , E , tmp0 , tmp1 , tmp2);
and and1(Out[1] , E , In[0] , tmp1 , tmp2);
and and2(Out[2] , E , tmp0 , In[1] , tmp2);
and and3(Out[3] , E , In[0] , In[1] , tmp2);
and and4(Out[4] , E , tmp0 , tmp1 , In[2]);
and and5(Out[5] , E , In[0] , tmp1 , In[2]);
and and6(Out[6] , E , tmp0 , In[1] , In[2]);
and and7(Out[7] , E , In[0] , In[1] , In[2]);
endmodule
```

3 to 8 Decoder (3/4)

Data flow description

1. module decoder(E , In , Out);
2. input E;
3. input [2:0]In;
4. output [7:0]Out;
5. wire [7:0]Out;
6. assign Out = E ? 1'b1 << In : 8'h0;

   true false
7. endmodule



Synthesized and optimized by tools

3 to 8 Decoder (4/4)

Behavioral description

```
module Decoder_Behavioral(E, In, Out);
  Input           E;
  input   [2:0]   In;
  output  [7:0]   Out;
  reg     [7:0]   Out;
  always @(E or In)
  begin
  if(!E)
       Out = 8'h00;
  else
       begin
          case(In)
            3'b000: Out = 8'h01;
            3'b001: Out = 8'h02;
            3'b010: Out = 8'h04;
            3'b011: Out = 8'h08;
            3'b100: Out = 8'h10;
            3'b101: Out = 8'h20;
            3'b110: Out = 8'h40;
            default: Out = 8'h80;
          endcase
       end
  end
  endmodule
```

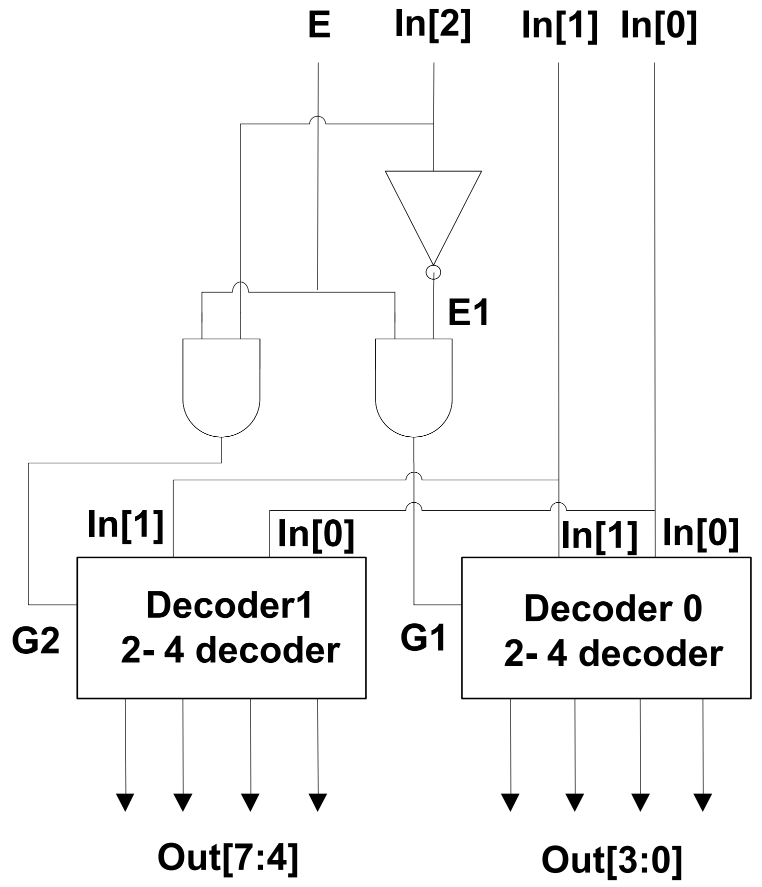
E	In			Out							
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

Hierarchical Design of 3-8 decoder

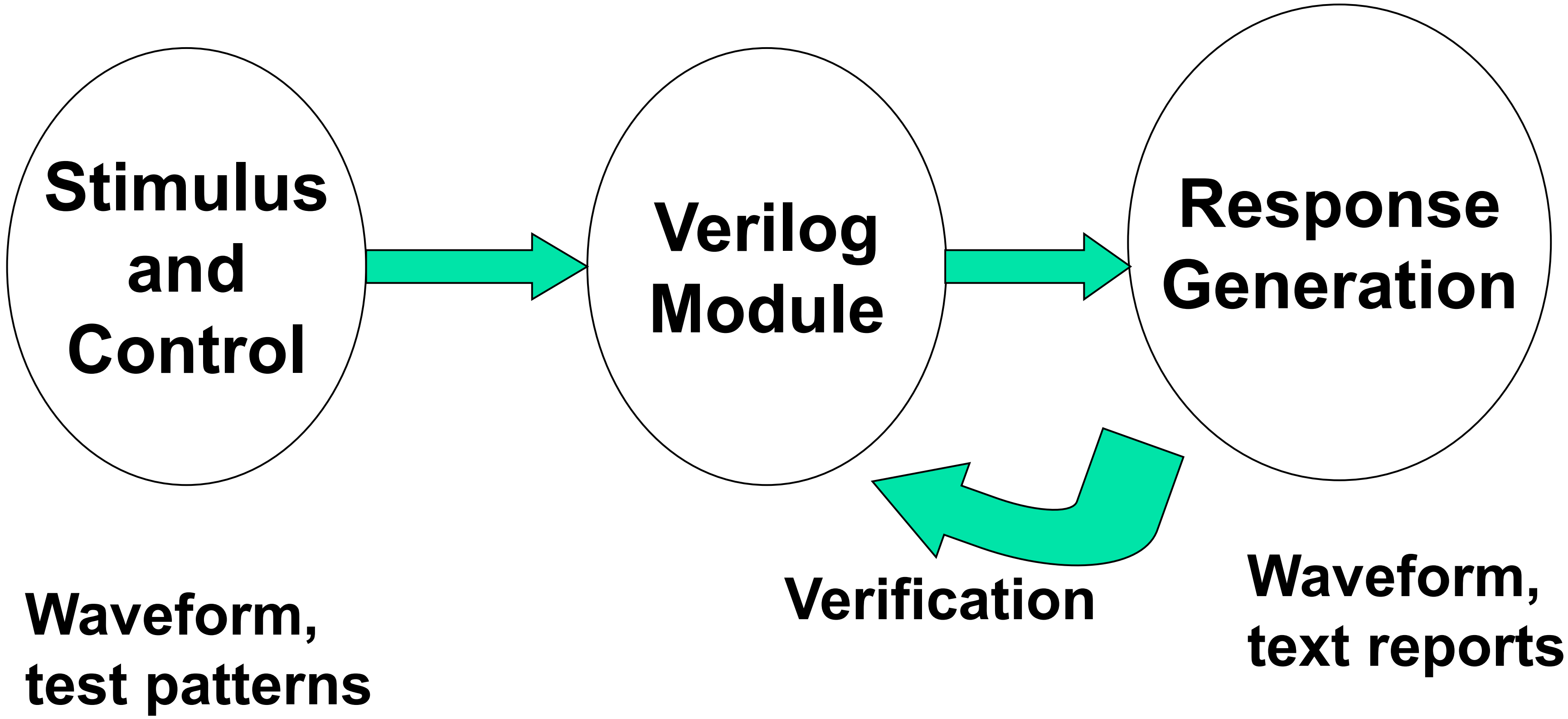
```
module decoder_2_4(E , In , Out);
input E;          input [1:0]  In;
output [3:0]Out;   wire [3:0]  Out;

assign Out = E ? 1'b1 << In : 4'h0;

endmodule
```

2 to 4 decoder

```
module decode_3_8(E , In , Out);
input E;        input [2:0] In;
output                    [7:0] Out;   wire E1 , G1 ,
        G2;
  not u1(E1 , In[2]);  and a1(G1 , E , In[2]);
  and a2(G2 , E , E1);
  decoder_2_4 M(G1 , In[1 : 0] , Out[7 : 4]);
  decoder_2_4 L(G2 , In[1 : 0] , Out[3 : 0]);
endmodule
```



Simulation



Test Bench

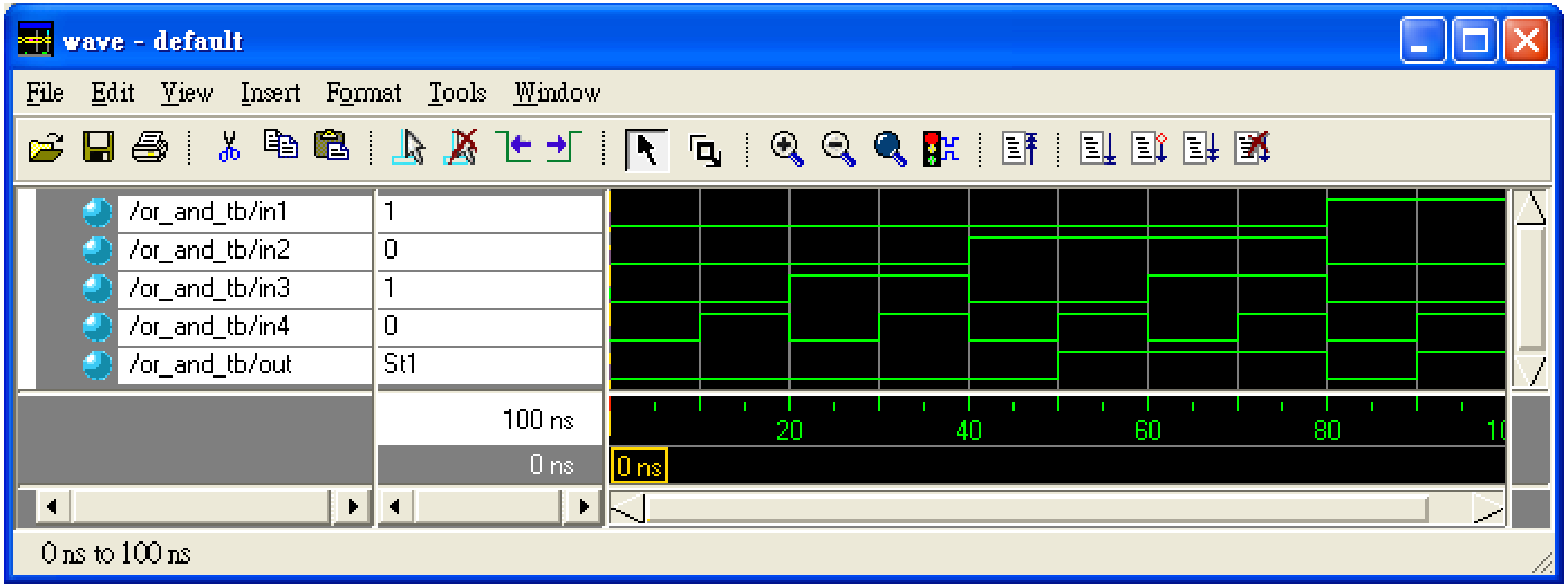
ModelSim Test_Bench Simulation (1/4)

Test_bench

```
module or_and_tb;
reg in1, in2, in3, in4;
wire out;
or_and ok(.in1(in1), .in2(in2),
        .in3(In3), .in4(in4), .out(out
       ));
initial
begin
#0    in1=0; in2=0; in3=0; in4=0;
#10  in1=0; in2=0; in3=0; in4=1;
#10  in1=0; in2=0; in3=1; in4=0;
#10  in1=0; in2=0; in3=1; in4=1;
#10  in1=0; in2=1; in3=0; in4=0;
#10  in1=0; in2=1; in3=0; in4=1;
#10  in1=0; in2=1; in3=1; in4=0;
#10  in1=0; in2=1; in3=1; in4=1;
#10 in1 = 1; in2 = 0; in3 = 0; in4 = 0;
#10 in1 = 1; in2 = 0; in3 = 0; in4 = 1;
#10 in1 = 1; in2 = 0; in3 = 1; in4 = 0;
#10 in1 = 1; in2 = 0; in3 = 1; in4 = 1;
#10 in1 = 1; in2 = 1; in3 = 0; in4 = 0;
#10 in1 = 1; in2 = 1; in3 = 0; in4 = 1;
#10 in1 = 1; in2 = 1; in3 = 1; in4 = 0;
#10 in1 = 1; in2 = 1; in3 = 1; in4 = 1;

end

endmodule
```

ModelSim Test_Bench Simulation (2/4)

Functional simulation only (no delay is introduced)


wave - default

/or_and_tb/in1	1
/or_and_tb/in2	0
/or_and_tb/in3	1
/or_and_tb/in4	0
/or_and_tb/out	St1

100 ns

0 ns

0 ns to 100 ns

In ModelSim, Input: test patterns (using comds.)   Output: waveform

In Altera,   Input: waveform Output: waveform

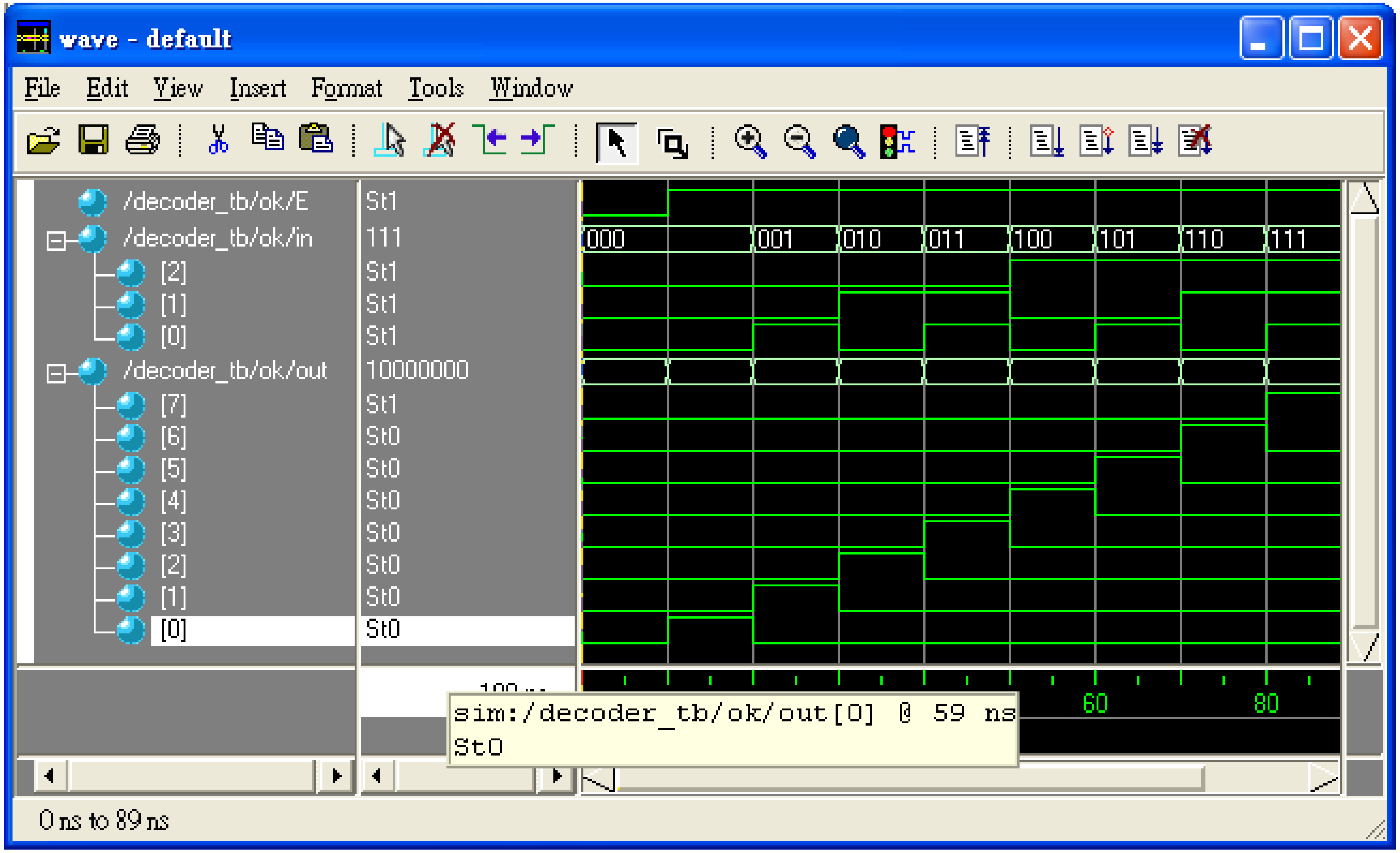
ModelSim Test_Bench Simulation (3/4)

Test_bench

```
module decoder_3_8_tb;

reg         E;
reg [2:0] in;

wire [7:0] out;

decoder
    ok(.E(E), .in(in), .out(out));

initial
begin
#0   E = 0; in = 3'b000;
#10 E = 1; in = 3'b000;
#10 E = 1; in = 3'b001;
#10 E = 1; in = 3'b010;
#10 E = 1; in = 3'b011;
#10 E = 1; in = 3'b100;
#10 E = 1; in = 3'b101;
#10 E = 1; in = 3'b110;
#10 E = 1; in = 3'b111;

end
endmodule
```

Exhaustive test or partial test for functional simulation in test_bench **? How about chip test ? Exhaustive or partial testing?**

ModelSim Test_Bench Simulation (4/4)



/decoder_tb/ok/E	St1
/decoder_tb/ok/in	111
[2]	St1
[1]	St1
[0]	St1
/decoder_tb/ok/out	10000000
[7]	St1
[6]	St0
[5]	St0
[4]	St0
[3]	St0
[2]	St0
[1]	St0
[0]	St0

```
sim:/decoder_tb/ok/out[0] @ 59 ns
St0
```

0 ns to 89 ns

Begin_End Statements

Begin-end block statements:

1. Block statements is a way of syntactically grouping several statements into a single statement.
2. Sequential blocks are delimited by the keywords begin and end. These begin…end pairs are commonly used in conjunction with if, case, and for statements to group several statements.
3. Functions and always blocks that contain more than one statement require a begin…end pair to group the statements.
4. Verilog provides a construct called a named block.

```
begin [: block_name

    reg local_variable_l ;

    integer local_variable_2 ;

    parameter local_variable_3 ;]

    . . . statements . . .

end
```

Verilog allows you to declare variables (reg, integer and parameter) locally within a named block but not in an unnamed block.

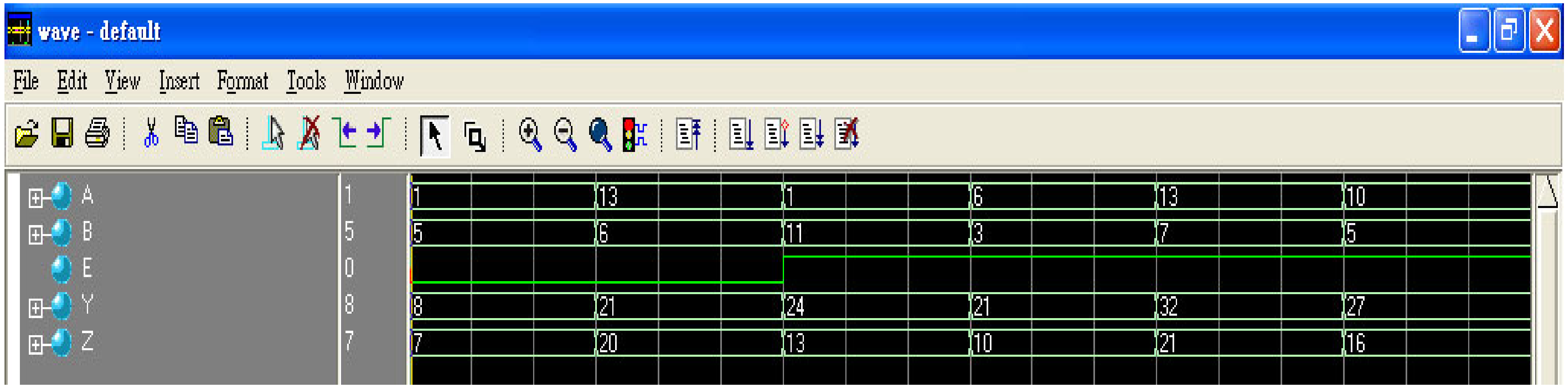
Named Block (1/2)

```
module UNNAMED_BLOCK(A , B , E , Y);
parameter FI = 12;
input [3 : 0]A;
input [3 : 0]B;
input E;
output [4 : 0]Y;

reg [4 : 0]Y;

always @(A or B or E)
begin
    if(E)
            Y = (A + B) & FI;
    else
            Y = (A - B) & FI;
end
endmodule
```

```
module NAMED_BLOCK(A , B , E , Y);
input [3 : 0]A;   input [3 : 0]B;
input E;  output [4 : 0]Y;  reg [4 : 0] Y;

always @(A or B or E)
begin
    if(E)
    begin: Add_And
    parameter FI = 12; // in a named block
    Y = (A + B) & FI;
    end
    else
    begin: Sub_And
    parameter FII = 2; // in a named block
    Y = (A - B) & FII;
    end
end
endmodule
```

Named Block (2/2)

```
module NAMED_BLOCK(A , B , E , Y);
input [3 : 0]A;  input [3 : 0]B;
input E;     output [4 : 0]Y;
reg [4 : 0]Y;

parameter FI = 1;
parameter FII = 8;
always @(A or B or E)
begin
if(E)
begin: Add_And
     parameter FI = 12;
     Y = (A + B) + FI; //FI is 12 not 1
end
else
begin: Sub_And
      parameter FII = 2;
     Y = (A +B) + FII; //FII is 2 not 8
end    end

assign Z = (A + B) + FI; //FI is 1
endmodule
```



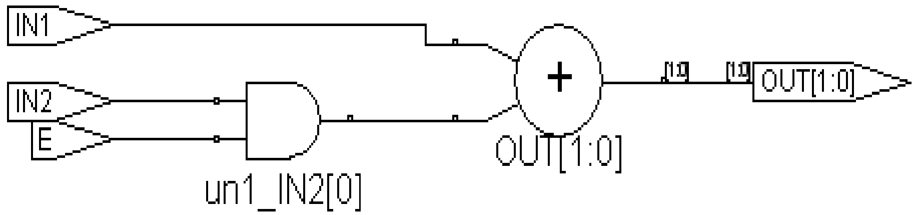
```
A  1   1   13  1   6   13  10
B  5   5   6   11  3   7   5
E  0
Y  8   8   21  24  21  32  27
Z  7   7   20  13  10  21  16
```

If-Else Statements (1/4)

1. The if statement is followed by a statement or block of statements enclosed by begin and end.
2. If the value of the expression is nonzero, the expression is true and the statement block that follows is executed. If the value of the expression is zero, the expression is false and the statement block following else is executed.
3. If..else statements can cause synthesis of latches.

```
If (expression)
     begin
     . . . statements . . .
     end
[eIse
     begin
     . . . statements . . .
    end]
```

```
module IF_ELSE(IN1 , IN2 , E , OUT);
input IN1, IN2, E;
output [1 : 0]  OUT;   reg [1 : 0]OUT;
always @(IN1 or IN2 or E)
begin
        if(E == 1)
              OUT = IN1 + IN2;
        else
              OUT = IN1;        end
endmodule
```

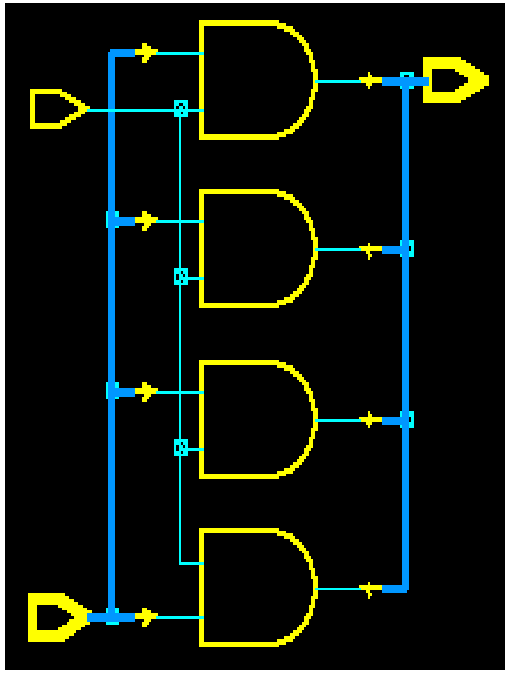


If-Else Statements (2/4)

```
Module Latch(In, Enable, Out);
input         Enable;
Input   [3:0] In;
output [3:0] Out;
always @(In or Enable)
begin
  if(Enable)
          Out=In;
  else
          Out=0;
end
endmodule
```

No latch inference

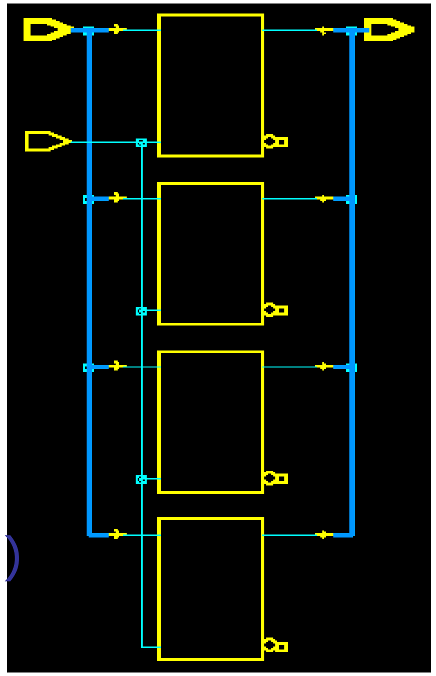
```
Always@ (In or
Enable)
begin
    Out=0;
    if(Enable)
    Out=In;
end   // no latch
```



Watch for unintentional Latches

```
Module Latch(In, Enable, Out);
input         Enable;
input   [3:0] In;
output [3:0] Out;

always @(In or Enable)
begin
  if(Enable)
          Out=In;
end
endmodule
```

If Enable ==1
Out (new) = In
If Enable==0
Out (new) = Out (old)



If-Else Statements (3/4)

```
module IF_ELSE_VALUE(IN , OUT);
input [1 : 0]IN;
output [1 : 0]OUT;
reg [1 : 0]OUT;
always @(IN)
begin
        if(IN==1)
                OUT = 2'b00;
        else
                OUT = 2'b11;
end
endmodule
```

IN=0,OUT=11
IN=1,OUT=00
IN=2,OUT=11
IN=3,OUT=11

true

false

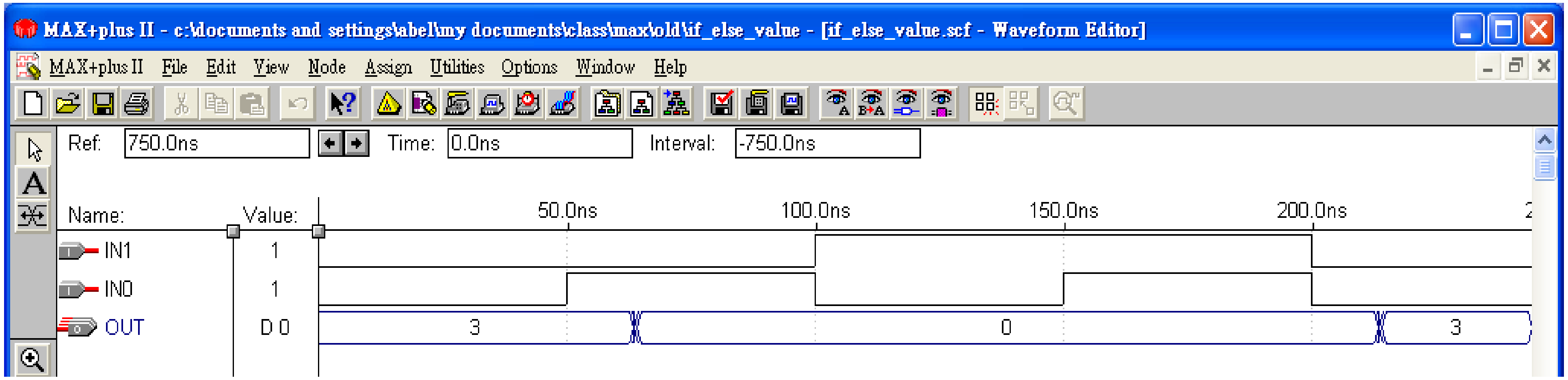
```
module IF_ELSE_VALUE(IN , OUT);
input [1 : 0]IN;
output [1 : 0]OUT;
reg [1 : 0]OUT;
always @(IN)
begin
        if(IN)
                OUT = 2'b00;
        else
                OUT = 2'b11;
end
endmodule
```

IN=0,OUT=11
IN=1,OUT=00
IN=2,OUT=00
IN=3,OUT=00

nonzero ⇨ true

zero ⇨ false

What is the difference between them?



MAX+plus II - c:\documents and settings\abel\my documents\class\max\old\if_else_value - [if_else_value.scf - Waveform Editor]

MAX+plus II File Edit View Node Assign Utilities Options Window Help

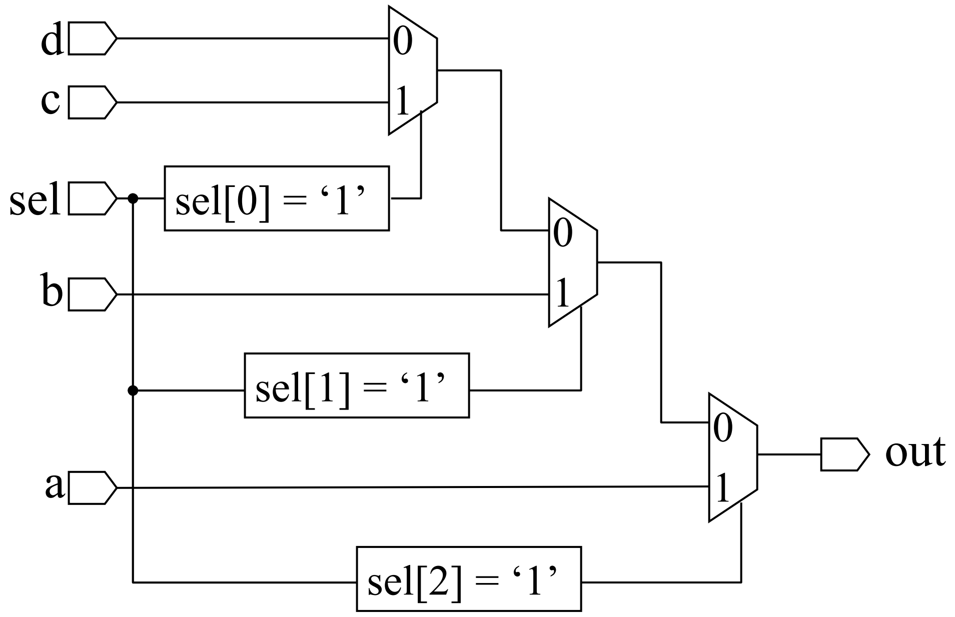
Ref: 750.0ns Time: 0.0ns Interval: -750.0ns

```
Name:   Value:          50.0ns        100.0ns        150.0ns        200.0ns
IN1     1
IN0     1
OUT     D 0       3                    0                                   3
```

If-Else Statements (4/4)

if-then-else statement implies priority-encoded MUXs

```
always @(sel or a or b or c or d)
   if (sel[2] == 1'b1)
      out = a;       //sel=1XX
   else if (sel[1] == 1'b1)
      out = b;       //sel=01X
   else if (sel[0] == 1'b1)
      out = c;       //sel=001
   else
      out = d;       //sel=000
```

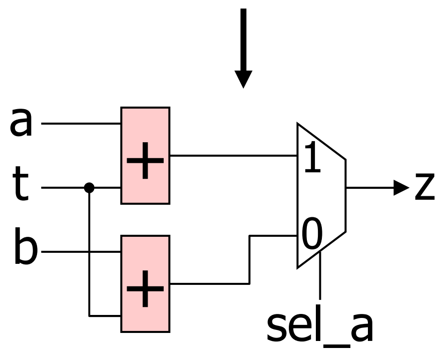


Resource Sharing (1/2)

- Assign similar operations to a common netlist cell
  - reduce hardware
  - may degrade performance
  - resource sharing within the same **always block**
  - resource sharing **not done** for **conditional operator**

Without resource sharing

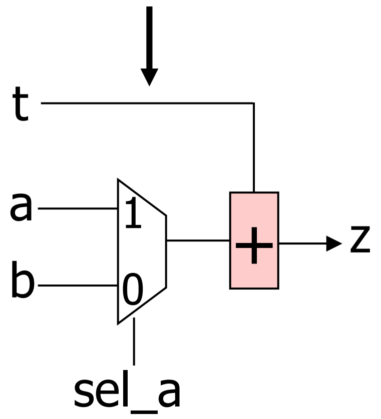
```
assign z = sel_a ? a+t : b+t;
```



Hence, in this case you better use if statement not conditional operator "?" to save a lot of cost and area

Resource sharing

```
if (sel_a)
    z = a + t;
else
    z = b + t;
```



Resource Sharing (2/2)

```
module noshare(v, w, x, z, k);
input  [2:0]k,v,w;
input  x;
output [3:0]z;
wire   [3:0]y;

assign y=x?k+w:k+v;
assign z=x?y+w:y+v;

endmodule
```

Without resource sharing

With resource sharing

```
module  share(v, w, z,k);
input       [2:0] k,v,w;
input       x;  output    [3:0]z;
reg          [3:0] y,z;

always@(x or k or v or w)
begin
     if(x)
       y=k+w;
     else
       y=k+v;
end
always@(y or x or w or v)
begin
     if(x)
       z=y+w;
     else
       z=y+v;
end
endmodule
```

Case Statements (1/4)

The case statement consists of the keyword case, followed by an expression in parentheses, followed by one or more case items (and associated statements to be executed), followed by the keyword endcase. A case item consists of an expression (usually a simple constant) or a list of expressions separated by commas, followed by a colon (:).

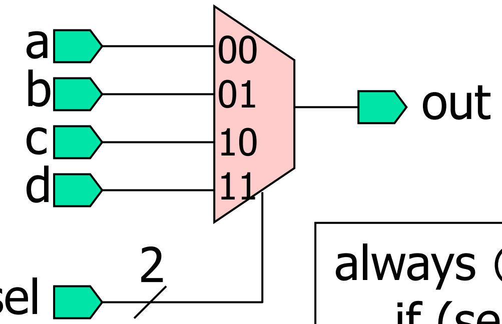
```
case (expression)

    case_item 1:
        begin
        . statements . .
        end
    case_item 2:
        begin
        . statements. .
        end
    . . . . . .
    default:
        begin
        . statements. .
        end
endcase
```

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

```
module FULL_CASE(IN , OUT);
input  [1 : 0]IN;output [3 : 0] OUT;
reg [3 : 0]OUT;
always @(IN)
begin
        case(IN)
        2'b00: OUT = 4'b0001;
        2'b01: OUT = 4'b0010;
        2'b10: OUT = 4'b0100;
        endcase // not full-case, latches are inferred
end
endmodule
```

Case Statements (2/4)

If the conditional expression used is mutually exclusive (i.e. parallel) and the functional outputs are the same, then the hardware synthesized will be same.



```
always @ (sel or a or b or c or d)
begin
    case (sel)
        2'b00: out = a;
        2'b01: out = b;
        2'b10: out = c;
        default : out = d;
    endcase
end
```

```
always @(sel or a or b or c or d)
    if (sel == 2'b00)
        out = a;
    else if (sel == 2'b01)
        out = b;
    else if (sel == 2'b10)
        out = c;
    else
        out = d;
```

Case Statements (3/4)

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

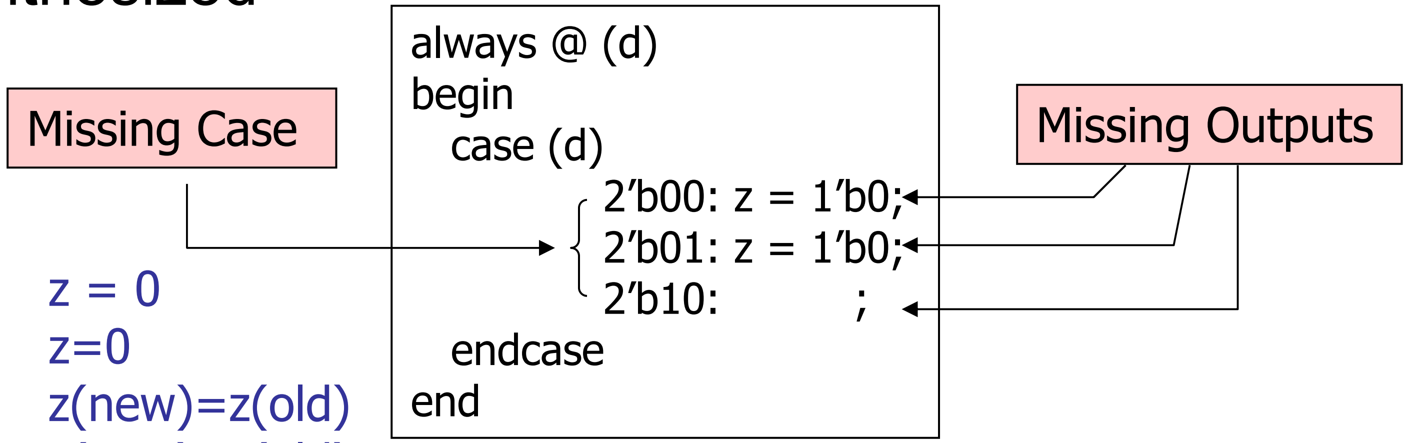
```
module FULL_CASE(IN , OUT);
input  [1 : 0]IN;  output [3 : 0] OUT;
reg [3 : 0]OUT;
always @(IN)
begin
        case(IN)
        2'b00: OUT = 4'b0001;
        2'b01: OUT = 4'b0010;
        2'b10: OUT = 4'b0100;
        default: OUT = 4'b1000;
        endcase
end
endmodule
```

It is always a good idea to use default-case-item in all conditions to make sure no latch is inferred.

Case Statements (4/4)

Watch Out for Unintentional Latches

- Completely specify all clauses for every ***case*** and ***if*** statement
- Completely specify all output for every clause of each ***case*** or ***if*** statement
- Failure to do so will cause latches or flip-flops to be synthesized



```
always @ (d)
begin
   case (d)
            2'b00: z = 1'b0;
            2'b01: z = 1'b0;
            2'b10:          ;
   endcase
end
```

d ==00	z = 0
d ==01	z=0
d ==10	z(new)=z(old)
d ==11	z(new)=z(old)

Casez and Casex Statements

```
casez (expression)

    case_item 1:
        begin
        . statements . .
        end
    case_item 2:
        begin
        . statements. .
        end
    . . . . .
    default:
        begin
        . statements. .
        end
endcase
```

casez:
The case item
can use z or ?

casex:
The case item
can use z, x,
or ?

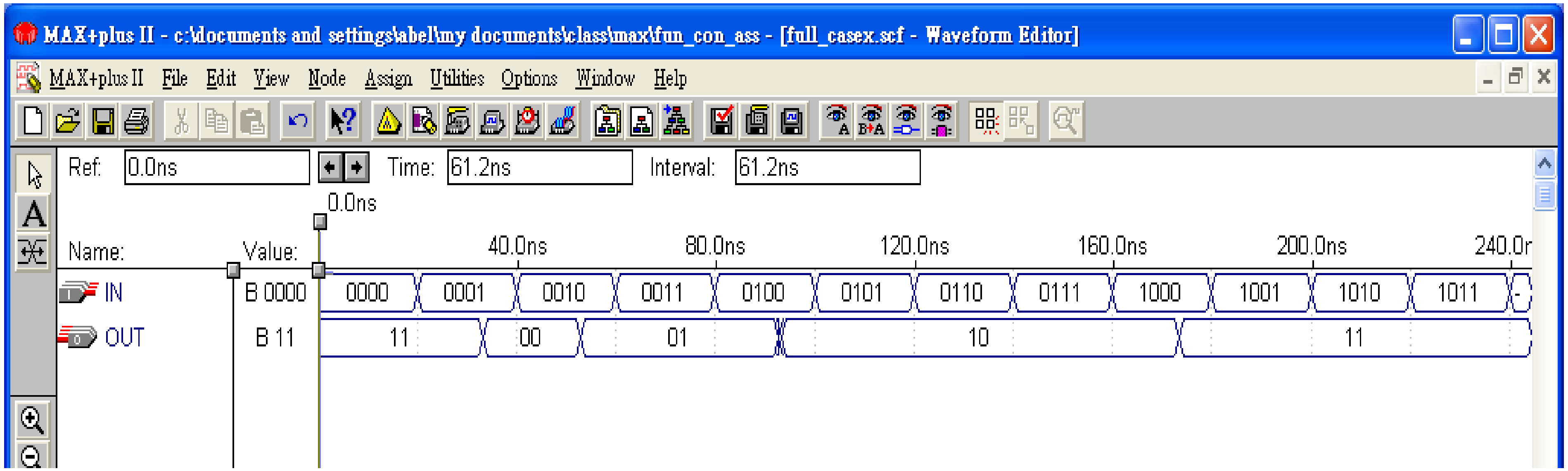
```
casex (expression)

    case_item 1:
        begin
        . statements . .
        end
    case_item 2:
        begin
        . statements. .
        end
    . . . . .
    default:
        begin
        . statements. .
        end
endcase
```

casez is one of the conditions in casex

Casex Statement

```
module FULL_CASEX(IN , OUT);
input  [3 : 0] IN;
output [1 : 0] OUT;

reg [1 : 0] OUT;
```

```
always @(IN)
begin
   casex(IN)
   4'b0001: OUT = 2'b00;
   4'b001?: OUT = 2'b01;
   4'b01??: OUT = 2'b10;
   default: OUT = 2'b11;
   endcase
end
endmodule
```



MAX+plus II - c:\documents and settings\abel\my documents\class\max\fun_con_ass - [full_casex.scf - Waveform Editor]

MAX+plus II File Edit View Node Assign Utilities Options Window Help

```
Ref: 0.0ns    Time: 61.2ns    Interval: 61.2ns
               0.0ns
Name:   Value:      40.0ns   80.0ns   120.0ns   160.0ns   200.0ns   240.0n
IN      B 0000  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 -
OUT     B 11    11   00   01   10   11
```

For Loop Statements (1/2)

The for loop repeatedly executes a single statement or block of statements. The repetitions are performed over a range determined by the range expressions assigned to an index. Two range expressions appear in each for loop: low_range and high_range. In the syntax lines that follow, high_range is greater than or equal to low_range. HDL Compiler recognizes incrementing as well as decrementing loops. The statement to be duplicated is surrounded by begin and end statements.

```
for (index = low_range; index < high_range; index = index + step)
for (index = high_range; index > low_range; index = index - step)
for (index = low_range; index <= high_range; index = index + step)
for (index = high_range; index >= low_range; index = index - step)
```

```
for (i = 0; i <= 31; i = i +1)
begin
    s[i] = a[i] ^ b[I] ^ carry;
    carry = (a[i] & b[i]) | (a[i] & carry) | (b[i] & carry);
end
```

For statement: unrolling the logic

For Loop Statements (2/2)

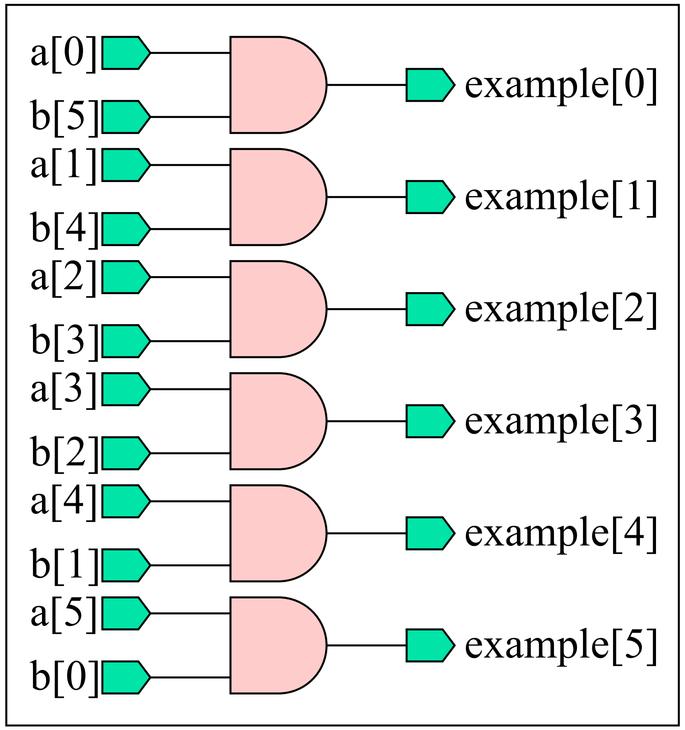
for loop are “unrolled”, and then synthesized.

```
integer i;
always @(a r b)
begin
    for (i = 0; i < 6; i = i+1)
        example[i] <= a[i] & b[5-i];
end
```

Verilog for loop

```
example [0] <= a[0] and b[5];
example [1] <= a[1] and b[4];
example [2] <= a[2] and b[3];
example [3] <= a[3] and b[2];
example [4] <= a[4] and b[1];
example [5] <= a[5] and b[0];
```

for loop unrolled



for loop synthesized to gates

Sorting Problem (1/2)

Bubble Sort: (four inputs)

1. (0)?(1) (compare temp[0] and temp [1], then the bigger value is stored in temp[1])

   (1)?(2) (compare temp[1] and temp [2],..)

   (2)?(3) (compare temp[2] and temp [3],..)

2. (0)?(1), (1)?(2)

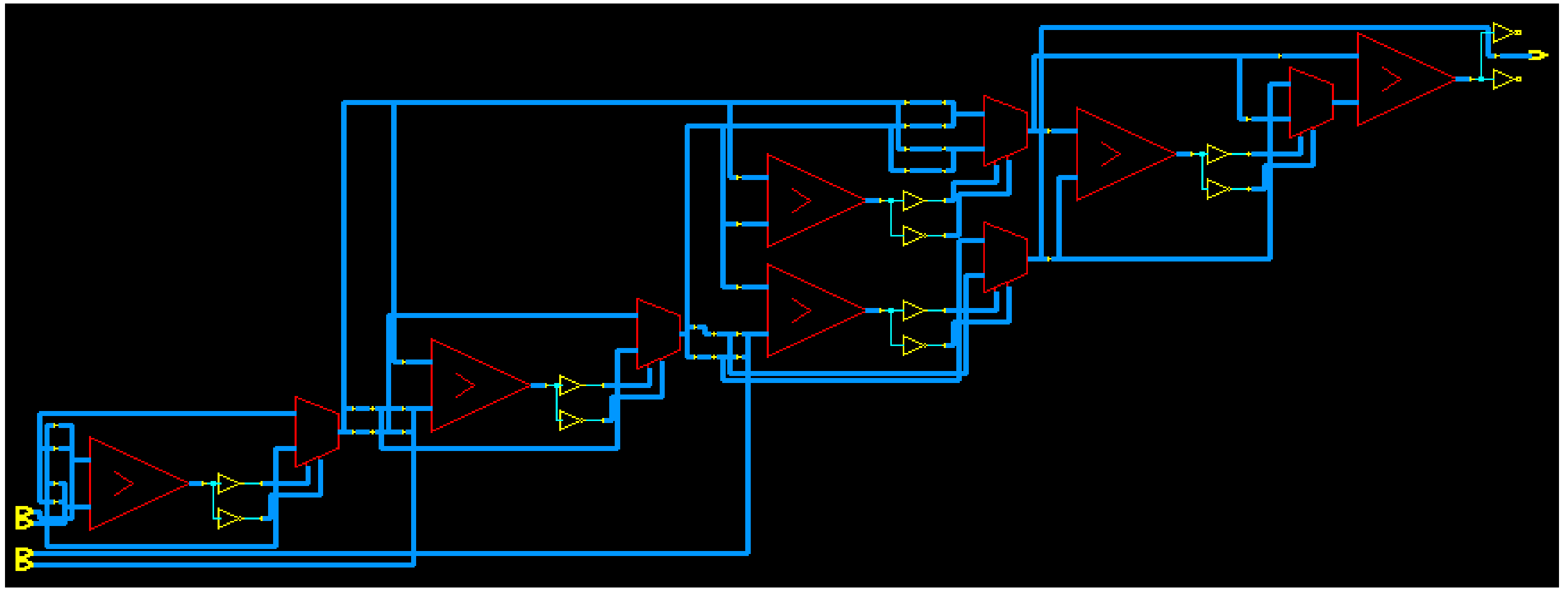
3. (0)?(1)

Totally, six comparators are needed for parallel comparisons of 4 inputs

```
module   for_loop(a,b,c,d,out);
input    [3:0]a,b,c,d;
output   [3:0]out;
reg      [3:0] temp [3:0];
reg      [3:0] buffer,out;
integer  i,j;
```

```
always@(a or b or c or d)
begin
    temp[0]=a;
    temp[1]=b;
    temp[2]=c;
    temp[3]=d;
    for(i=2;i>=0;i=i-1)
      for(j=0;j<=i;j=j+1)
        if(temp[j]>temp[j+1])
          begin
            buffer=temp[j+1];
            temp[j+1]=temp[j];
            temp[j]=buffer;
          end
    out=temp[3];
end
endmodule
```

Sorting Problem (2/2)



Bubble Sort: (four inputs)

(0)?(1), (1)?(2), (2)?(3)

(0)?(1), (1)?(2)

(0)?(1) critical path=6.01 ns

Totally, 6 comparators are needed

for parallel comparisons

Bubble Sort: (five inputs)

(0)?(1), (1)?(2), (2)?(3), (3)?(4)

(0)?(1), (1)?(2), (2)?(3)

(0)?(1), (1)?(2)

(0)?(1) critical path=8.05 ns

Totally, 10 comparators are needed

The more inputs, the longer delay $\Rightarrow$ use one comparator and a suitable FSM

Always Block (1/3)

An always block can imply latches or flip-flops, or it can specify purely combinational logic. An always block can contain logic triggered in response to a change in a level (asynchronous triggers) or the rising or falling edge of a signal (synchronous triggers).
The syntax of an always block is

```
always @  (event-expression )
begin
  . . . statements . . .
end (combinational circuit)
```

Completely specify sensitivity lists to avoid error

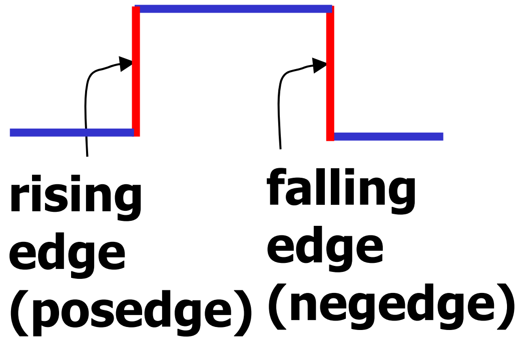
Asynchronous triggers

```
always@ (a or b or c)
begin
x=a | b | c;
end
```

x is recalculated as soon as any input (a or b or c) has a level transition (0 to 1 or 1 to 0).

Always Block (2/3)

```
always @  ( [posedge or negedge] event)
begin
  . . . statements . . .
end (sequential circuit)
```



Rising edge or positive edge (posedge)
Falling edge or negative edge (negedge)

x is recalculated as soon as c changes from 0 to 1 or d changes from 1 to 0.

Synchronous triggers

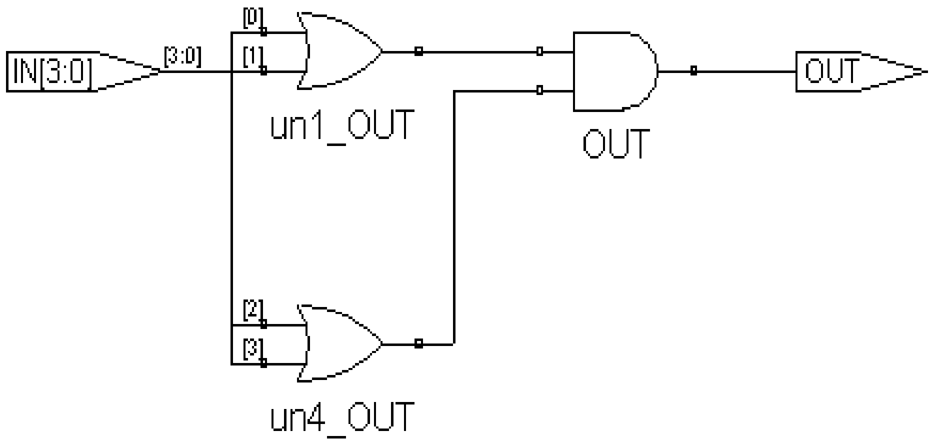
```
always@ (posedge c or negedge d)
begin
x=a +b;
end
```

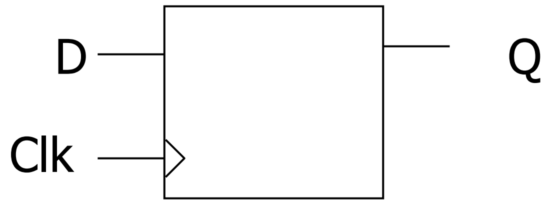
storage unit

Always Block (3/3)

```
module ALWAYS_BLOCK(IN , OUT);
input [3 : 0]IN; output OUT;  reg OUT;

always @(IN)
begin
        OUT = (IN[0] | IN[1]) & (IN[2] | IN[3]);
end
endmodule
```

```
module D_FF(Clk, D, Q);
input  Clk, D;
output Q;
Reg    Q;

always @(posedge Clk)
begin
            Q=D;
end
endmodule
```





At every positive edge of signal Clk, Q is set as D.

Variables

Nets:

```
wire    a;          // 1-bit wire
wire  [3:0]  b;   // 4-bit wire
```

Three types of common variables in Verilog:

(1) register (default width is 1 bit)

(2) integer (default width is 32 bit)

(3) parameter (default width is 1 bit)

```
parameter [range] identifier=expression,
                  identifier=expression
```

```
reg   a;          // 1-bit register
reg  [3:0]  b;  // 4-bit register
integer c;      // single 32-bit
integer
parameter d=4, e=6;
```

Three variables (register, integer and parameter) are declared globally at the module level, or locally at the function level and begin-end block.

Verilog allows you to assign the value of a reg variable only within a function or an always block.

Function

```
function [range] name_of_function;
        function declaration
        statement
endfunction
```

1. Begin with function and end with endfunction
2. [Range] defines the width of the return value of the function (default is 1 bit)
   Contain one or more statements (enclosed inside a begin-end pair)
3. You can call function in a continuous assignment, always block or other functions

Function declaration:

Input declaration: specify the input signals for a function

Output: The output from a function is assigned to the function name.

Use concatenation operation to bundle several values for multi-outputs

Function Statements (1/4)

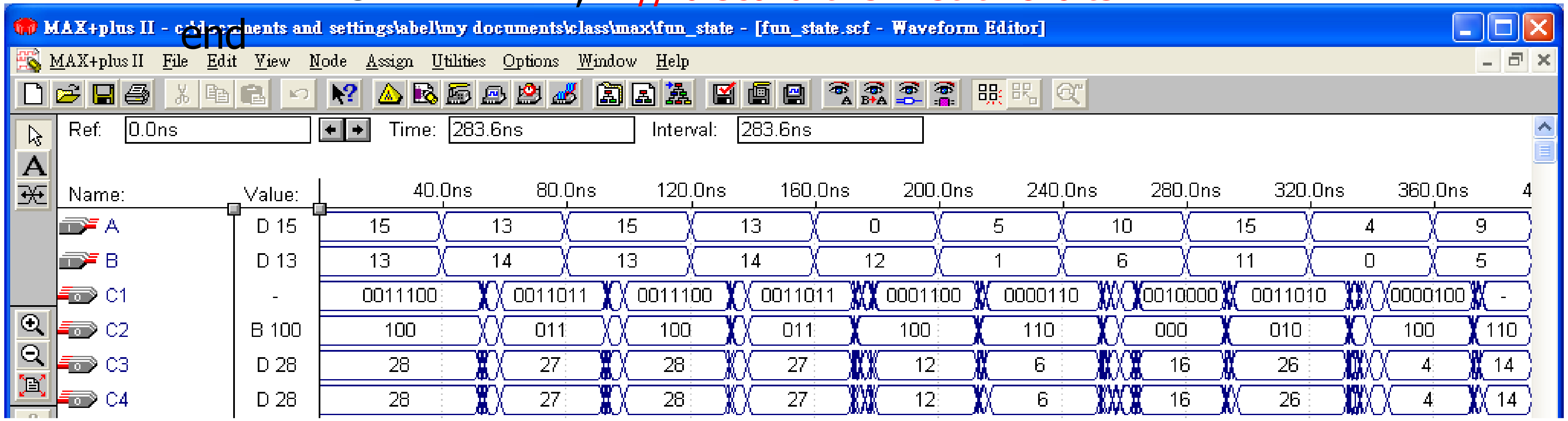
Procedure assignments are assignment statements used inside a function. (It is similar to C language,   Note: it cannot be used in module)
They are similar to the continuous assignments, except that the left side of a procedural assignment can contain only reg and integer variables.

```
module FUN_STATE(A , B , C1 , C2 , C3 , C4 , C5);
input [3 : 0]A;   input [3 : 0]B;
output [6 : 0]C1;                    //0
output [2 : 0]C2;                    //discard
output [4 : 0]C3;                    //always
output [4 : 0]C4;                    //assign
reg [6 : 0]C1;   reg [2 : 0]C2;
reg [4 : 0]C3;   reg [4 : 0]C4;

function [4 : 0]Fn1;
input [3 : 0]A;  input [3 : 0]B;
        Fn1 = A + B;        // like C
endfunction

always @(A or B)
begin
     C1 = Fn1(A , B);
end
always @(A or B)
begin
     C2 = Fn1(A , B);
end
always @(A or B)
begin
     C3 = A + B;
end

assign C4 = A + B;

endmodule
```

Three different ways to implement addition

Function Statements (2/4)

```
input [3 : 0]A;      input [3 : 0]B;          // A and B are 4-bit
values
output [6 : 0]C1;                            // C1 is 7-bit values
output [2 : 0]C2;                             // C2 is 3-bit values

always @(A or B)
begin
          C1 = A +B;        // insert "0" in the first two bits
end
begin
          C2 = A + B  ;     // discard the first two bits
end
```



MAX+plus II - c:\documents and settings\abel\my documents\class\max\fun_state - [fun_state.scf - Waveform Editor]

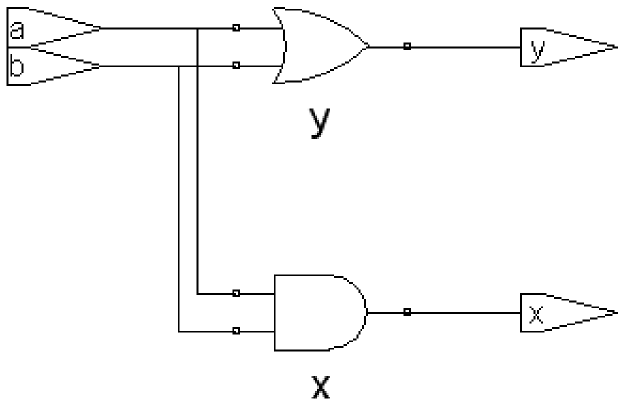
Ref: 0.0ns   Time: 283.6ns   Interval: 283.6ns

```
Name:  Value:    40.0ns    80.0ns    120.0ns    160.0ns    200.0ns    240.0ns    280.0ns    320.0ns    360.0ns
A      D 15      15        13        15         13         0          5          10         15         4          9
B      D 13      13        14        13         14         12         1          6          11         0          5
C1     -         0011100   0011011   0011100    0011011    0001100    0000110    0010000    0011010    0000100    -
C2     B 100     100       011       100        011        100        110        000        010        100        110
C3     D 28      28        27        28         27         12         6          16         26         4          14
C4     D 28      28        27        28         27         12         6          16         26         4          14
```

Function Statements (3/4)

```
module test_n(a, b, x, y);
input      a, b; output  x, y;
reg        x, y;
function Fn1;
   input a, b;
   Fn1 = a & b;
   /* begin-end is required
      for more statements */
endfunction
function Fn2;
   input a, b;
   Fn2 = a | b;
endfunction
always @(a or b)
     begin
        x = Fn1(a, b);
        y = Fn2(a, b);
     end
endmodule
```

```
module test_n(a, b, x, y)
input      a, b;
output     x, y;
assign   x=a & b;
assign  y=a | b;
endmodule
```



```
module test_n(a, b, x, y);
input      a, b;
output     x, y;
reg        x, y;
always @(a or b)
begin
     x = a & b;
     y = a | b;
end
endmodule
```

C language

x = a&b;

y = a | b;

Function Statements (4/4)

```
module test_n(a1, a, b, x, y);
input [7:0] a1;
input      a, b;
output     x, y;
reg        x, y;

function Fn1;
input [width-1:0] a1;
parameter width=8;//error message
    input a, b;
    Fn1 = a & b;
endfunction

always @(a or b)
    begin
      x = Fn1(a1, a, b);
    end
endmodule
```

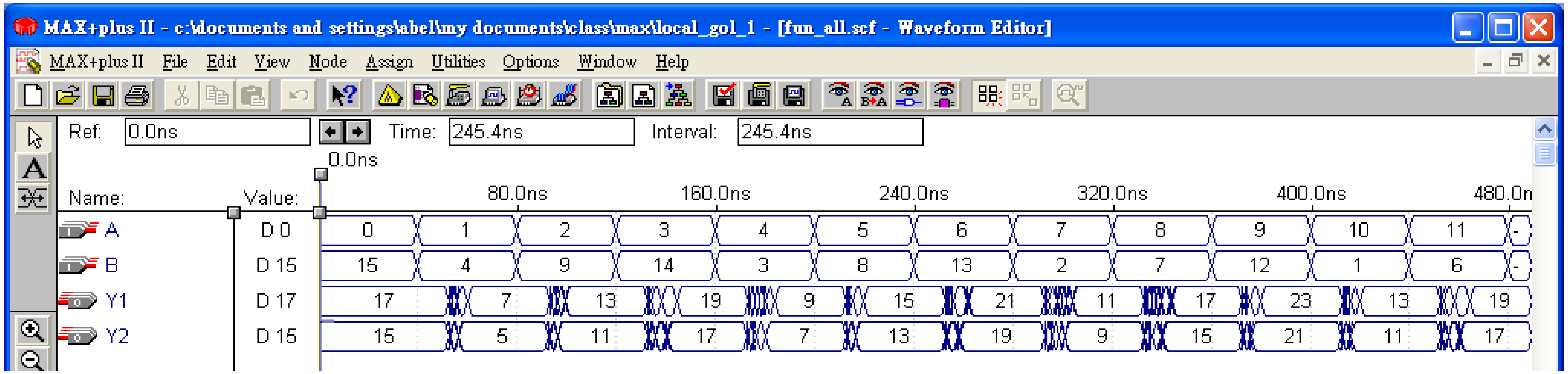
```
module test_n(a1, a, b, x, y);
input [7:0] a1;
input      a, b;
output     x, y;
reg        x, y;

function Fn1;
parameter width=8;
input [width-1:0] a1;   // OK
    input a, b;
    Fn1 = a & b;
endfunction

always @(a or b)
   begin
      x = Fn1(a1, a, b);
   end
endmodule
```

Can we input width with scanf or input ??
Why?

Example-Function

```
module FUN_ALL(A , B , Y1 , Y2);
input [3 : 0] A,B;output [4 : 0] Y1, Y2;
reg [4 : 0] Y1, Y2;

function [4 : 0] Fn2;
input [3 : 0] F1 , F2;
begin
    Fn2 = F1 + F2;
end
endfunction

function [4 : 0] Fn1;
input [3 : 0]F1 , F2;
begin
Fn1 = Fn2(F1 , F2) + 2;
end
endfunction

always @(A or B)
begin
    Y1 = Fn1(A , B);
    Y2 = Fn2(A , B);
end
endmodule
```

Y1=A+B+2
Y2=A+B


MAX+plus II - c:\documents and settings\abel\my documents\class\max\local_gol_1 - [fun_all.scf - Waveform Editor]
MAX+plus II File Edit View Node Assign Utilities Options Window Help

Ref: 0.0ns Time: 245.4ns Interval: 245.4ns

```
                0.0ns
Name:  Value:        80.0ns     160.0ns     240.0ns     320.0ns     400.0ns     480.0n
A      D 0      0    1    2    3    4    5    6    7    8    9    10   11   -
B      D 15     15   4    9    14   3    8    13   2    7    12   1    6    -
Y1     D 17     17   7    13   19   9    15   21   11   17   23   13   19
Y2     D 15     15   5    11   17   7    13   19   9    15   21   11   17
```