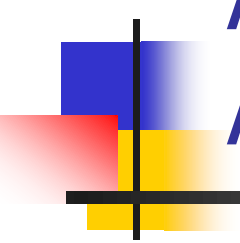


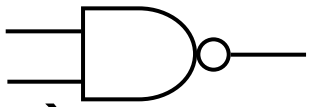
Advanced Computer Organization II
Advanced Computer Architecture II

Introduction to Verilog HDL

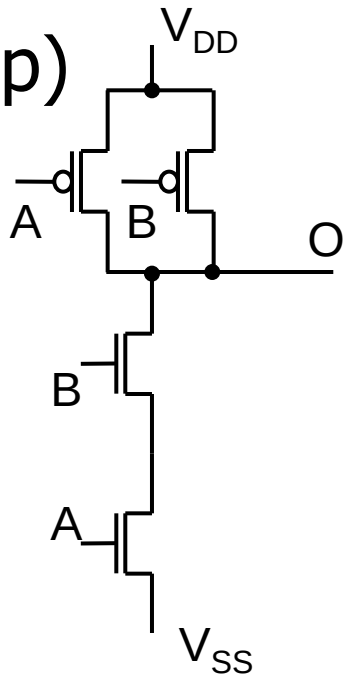
Reference

- D. E. Thomas and P. R. Moorby: The Verilog Hardware Description Language, Kluwer Academic Pub., 1995.

Why do we use HDL?



- History of design methodology
  - 1980's: 10kTr./chip (≒ 2.5k NAND2/chip)
    - Gate level design (schematic entry)
  - 1990's: 1,MTr./chip (≒ 250k NAND2/chip)
    - Register Transfer Level (RTL) design
      (Hardware Description Language)
  - 2000's～: over 100MTr./chip
    (≒25M NAND2/chip)
    - System level design
    - Platform based design
    - IP based Design (IP reuse)
    - High level synthesis (Behavioral design, C like language)



Why Hardware Description Language?

- Standardized HDL
  - Used for digital circuit design in world wide
  - Verilog HDL: Gateway design from 1984, IEEE Standard 1364
  - VHDL: DARPA (Defense Advanced Research Projects Agency,USA) from 1981, IEEE Standard 1076
- Support many design methodology and technology
  - Wired Logic vs. Microprogram Machine, …
- Independent from device technology and process
  - e.g. CMOS, TTL, ECL, …
- Widely description level
  - Architecture, Behavior, register transfer and gate level
- Support large scale design and design re-use
  - Library, IP (Intellectual Property)

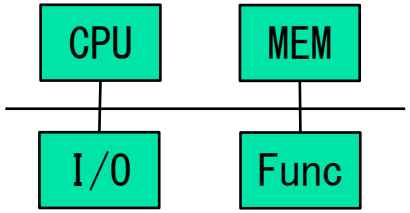
What is Verilog HDL?

- Verilog = Verify + Logic (coined word)
- History
  - Developed as description language for logic simulator by Gateway Design Inc. in 1984
  - Followed by CADENCE Inc.
  - OVI (Open Verilog International)

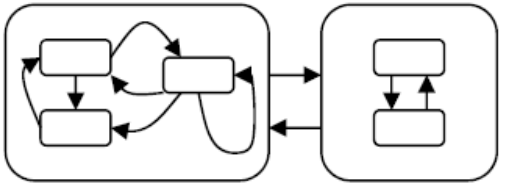
    http://www.ovi.org/
  - Standardized
    - IEEE Standard 1364-1995,2001,2005
    - IEEE Standard 1800-2005,2009,2012 (System Verilog)

Features of Verilog HDL

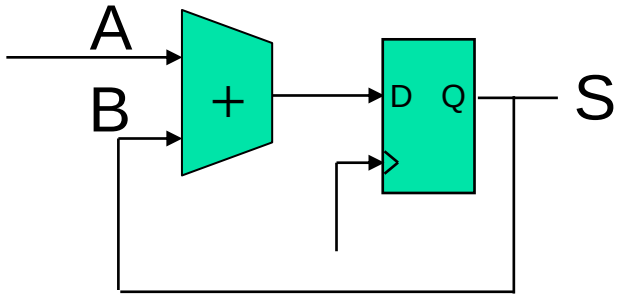
- Widely description level
  - Architecture Level



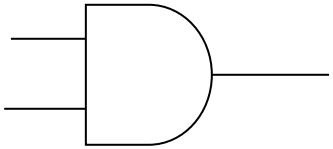
  - Behavior Level



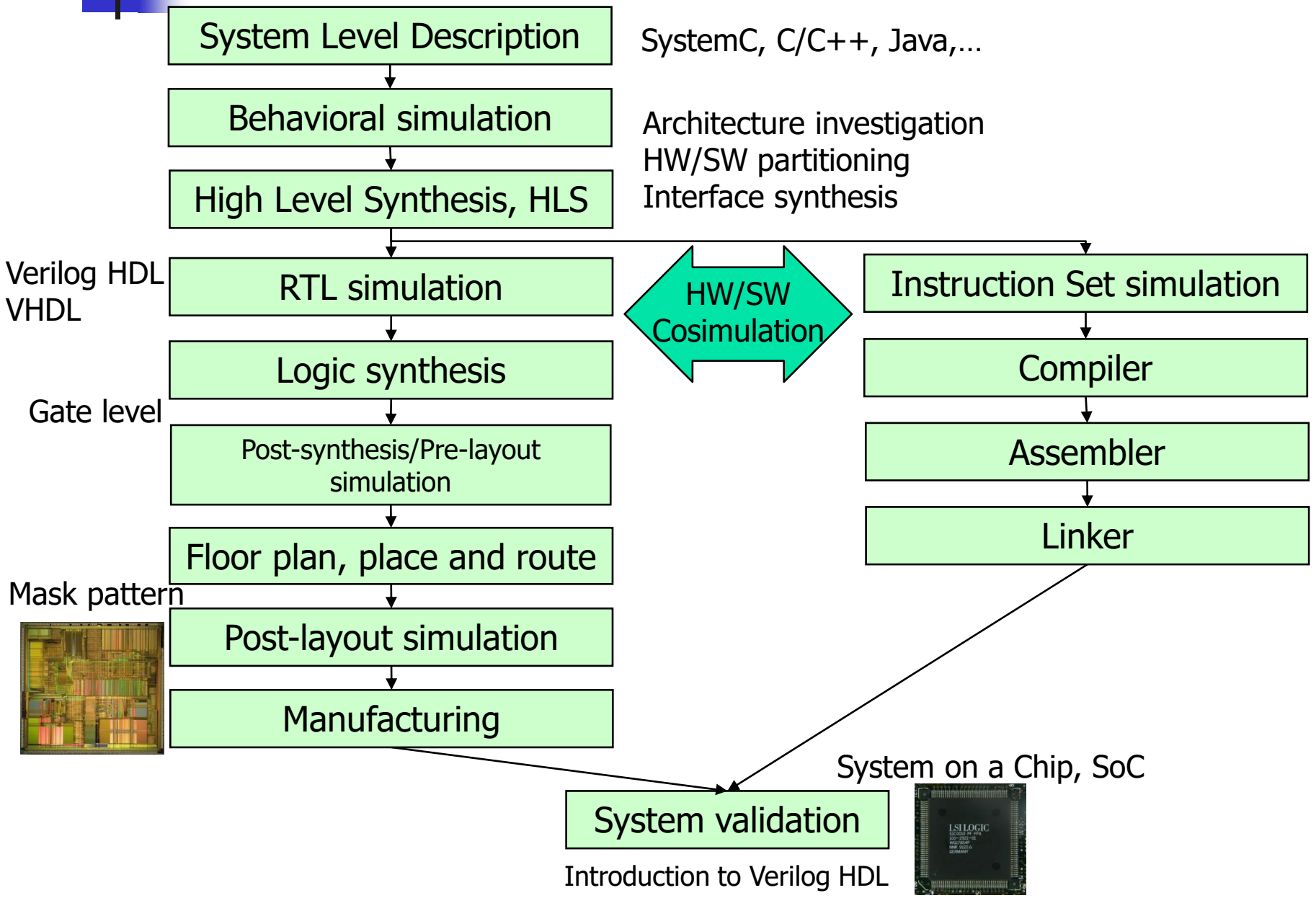
  - Register Transfer Level

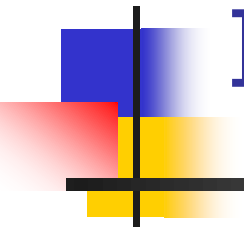


  - Gate Level



SoC design & implementation flow



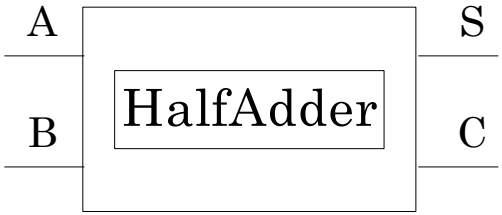


Introduction to Verilog HDL

Gate Level Description

Examples: Half adder and Full adder

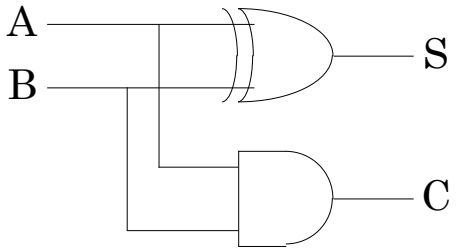
Half Adder (Gate Level)



Symbol

Truth Table

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Logic Circuit

```
module description
  module
    ...
  endmodule
```

Module name (commonly same as file name)

```
module halfadder ( A, B, S, C );
   input  A, B;
   output S, C;
   assign S = A ^ B;
   assign C = A & B;
endmodule
```

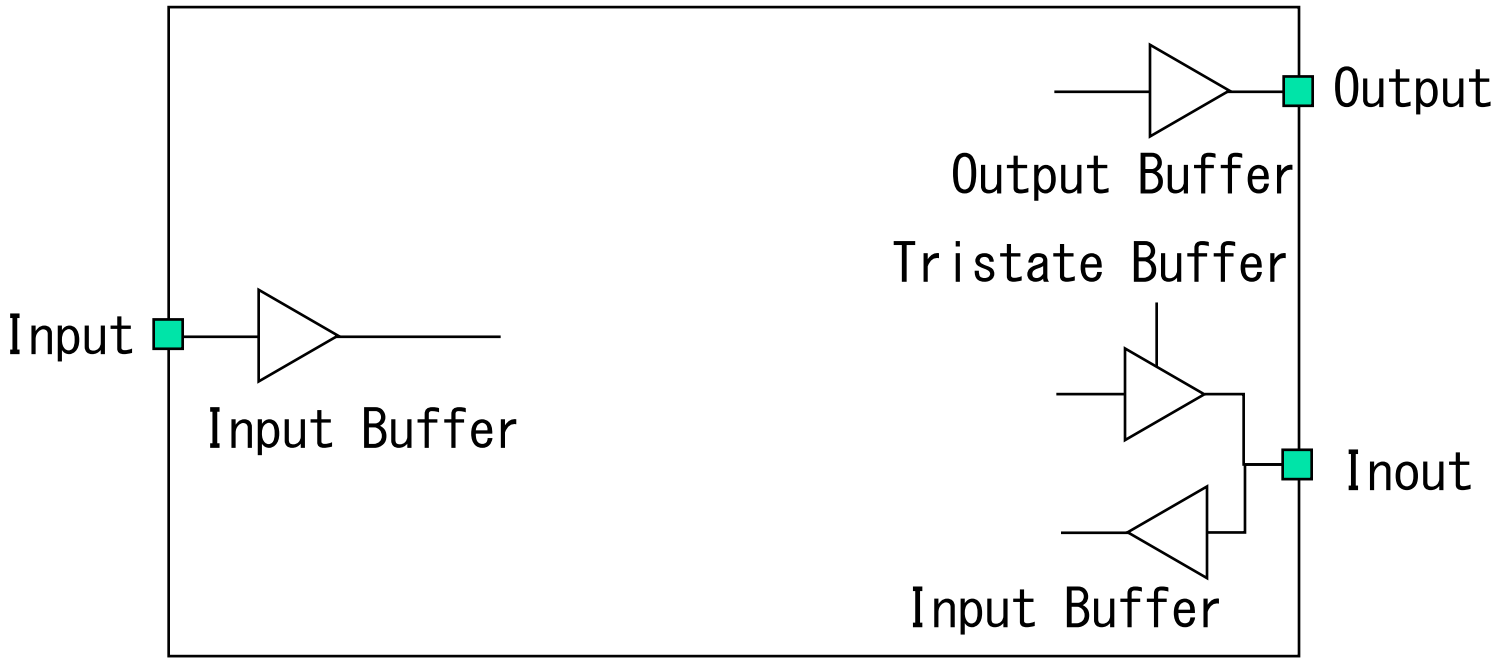
Arguments

Port Directions

assignments

Uppercase and lowercase letters are distinguished

Port Directions



Don't need to describe I/O buffer instances in Verilog HDL because logic synthesizer can automatically insert adequate I/O buffers.
(You can explicitly insert I/O buffer instances.)

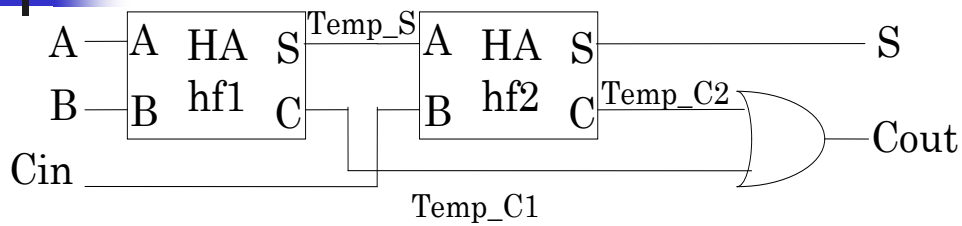
Values in Variable

- 4 types of signal value for digital simulation
  - ‘0’ Low level
  - ‘1’ High level
  - ‘Z’ High impedance
  - ‘X’ Unknown or Don’t Care

- 8 strengths for transistor level simulation
  - supply Power line
  - strong Default signal strength
  - pull Pull-up signal
  - large
  - weak
  - medium
  - small
  - highz High impedance

Don't care “strengths” in digital circuit design

Structured Description (Full Adder)①



Block Diagram of Full Adder

Variable type defined by input or output is "wire".

```
module fulladder ( A, B, Cin, S ,Cout );
  input  A, B, Cin;
  output S, Cout;

  // Internal wires
  wire   Temp_S, Temp_C1, Temp_C2;

  assign Cout = Temp_C2 | Temp_C1;

  halfadder hf1 ( A,      B,    Temp_S ,Temp_C1 );
  halfadder hf2 ( Temp_S, Cin,      S ,Temp_C2 );
endmodule
```

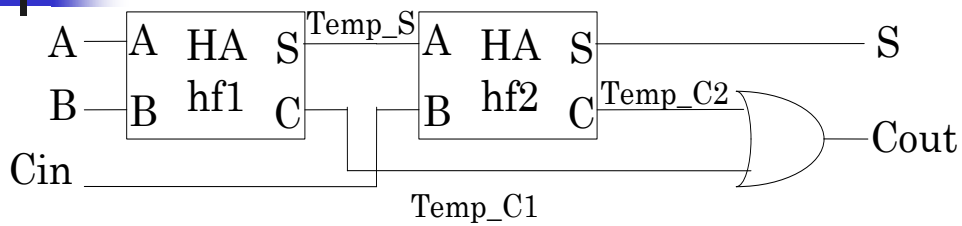
"assign" statement is used for wire type.

The order of arguments match in the lower layer and the upper layer as same as conventional programming languages.

Instance name (for identifying module)

Module name

Structured Description (Full Adder)②



Block Diagram of Full Adder

Variable type defined by input or output is "wire".

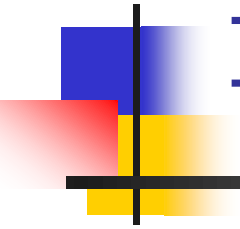
```
module fulladder ( A, B, Cin, S ,Cout );
  input  A, B, Cin;
  output S, Cout;

 // Internal wires
  wire   Temp_S, Temp_C1, Temp_C2;

  assign Cout = Temp_C2 | Temp_C1;

  halfadder hf1 ( .A(A),       .B(B),    .S(Temp_S), .C(Temp_C1) );
  halfadder hf2 ( .A(Temp_S), .B(Cin), .S(S),       .C(Temp_C2) );
endmodule
```

"assign" statement is used for wire type.

Dot notation can explicitly describe correspondence relationship of dummy and actual argument. (Recommend to this notation)

Dummy argument (Lower layer)

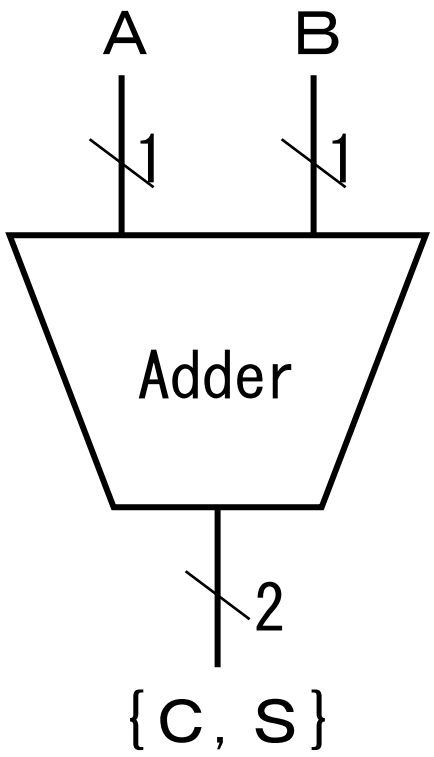
actual argument (this layer)



Introduction to Verilog HDL

Combinational Logic Design with abstracted (behavioral) description than gate level

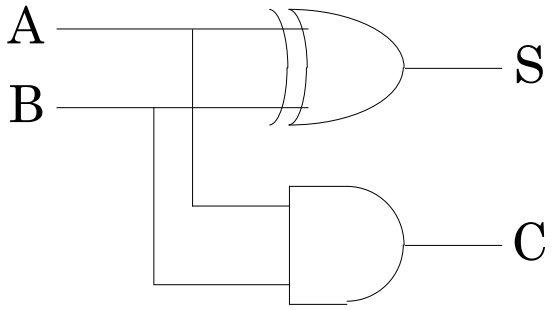
1-bit Adder (behavioral description)



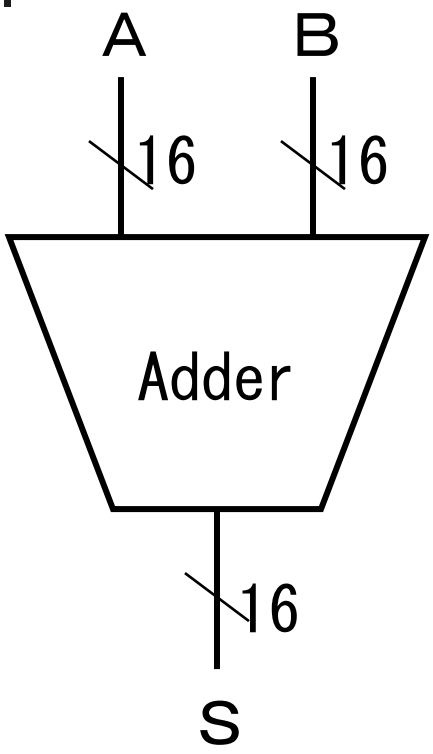

```
module adder ( A, B, S, C )
  input   A, B;
  output  C, S;

  assign  {C,S} = A + B;
endmodule
```

'+' operator for addition

Logic synthesis



16-bit Adder

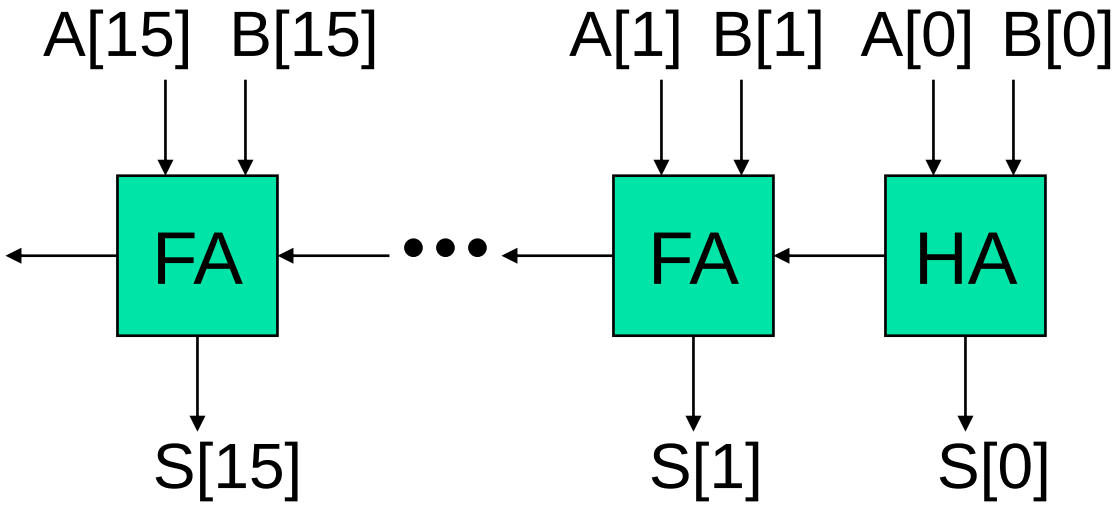


Bus type variable

```
module adder ( A, B, S )
  input   [15:0] A, B;
  output  [15:0] S;

  assign  S = A + B;
endmodule
```

'+' operator for addition

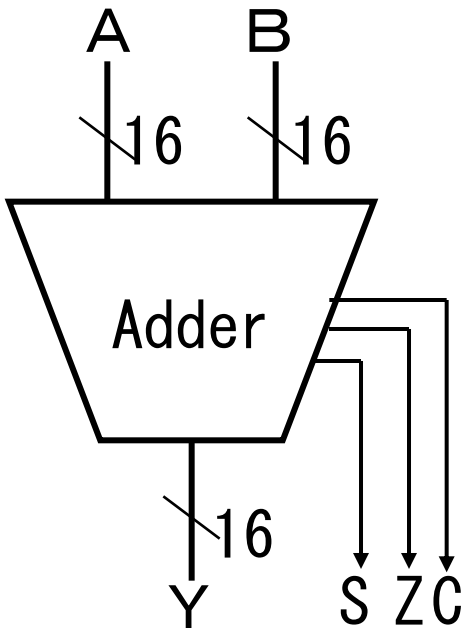
Logic synthesis



Adder algorithm is depend on constraints in logic synthesis.

16-bit Adder with Flags

Example of getting sign, carry and zero flags.



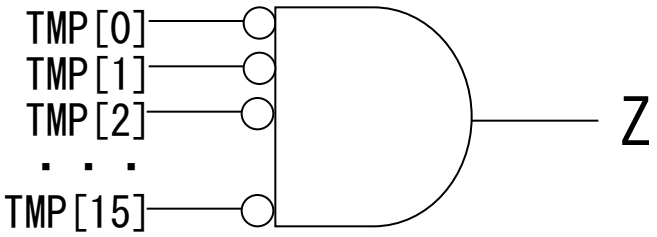

```
......
wire [15:0] A, B, Y;
wire [16:0] TMP;
wire        S, Z, C;
......
assign  TMP = { 1'b0, A } + { 1'b0, B };
assign  Y   = TMP[15:0];
assign  S   = TMP[15];
assign  Z   = ~| TMP[15:0];
assign  C   = TMP[16];
......
```

Concatenation
{ … , … }

bits select

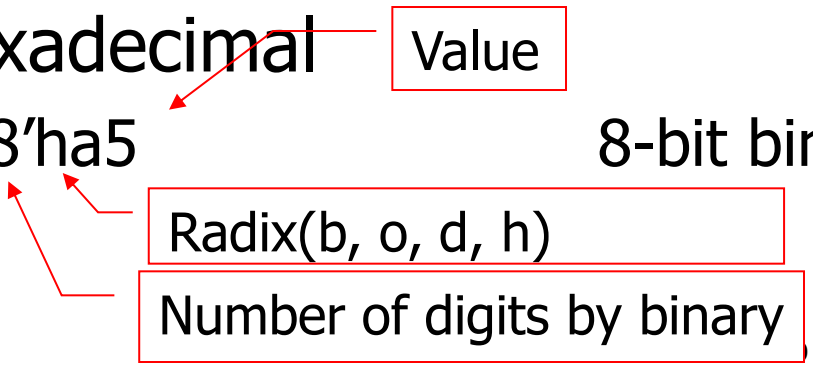
a bit selects

Reduction operator (NOR)



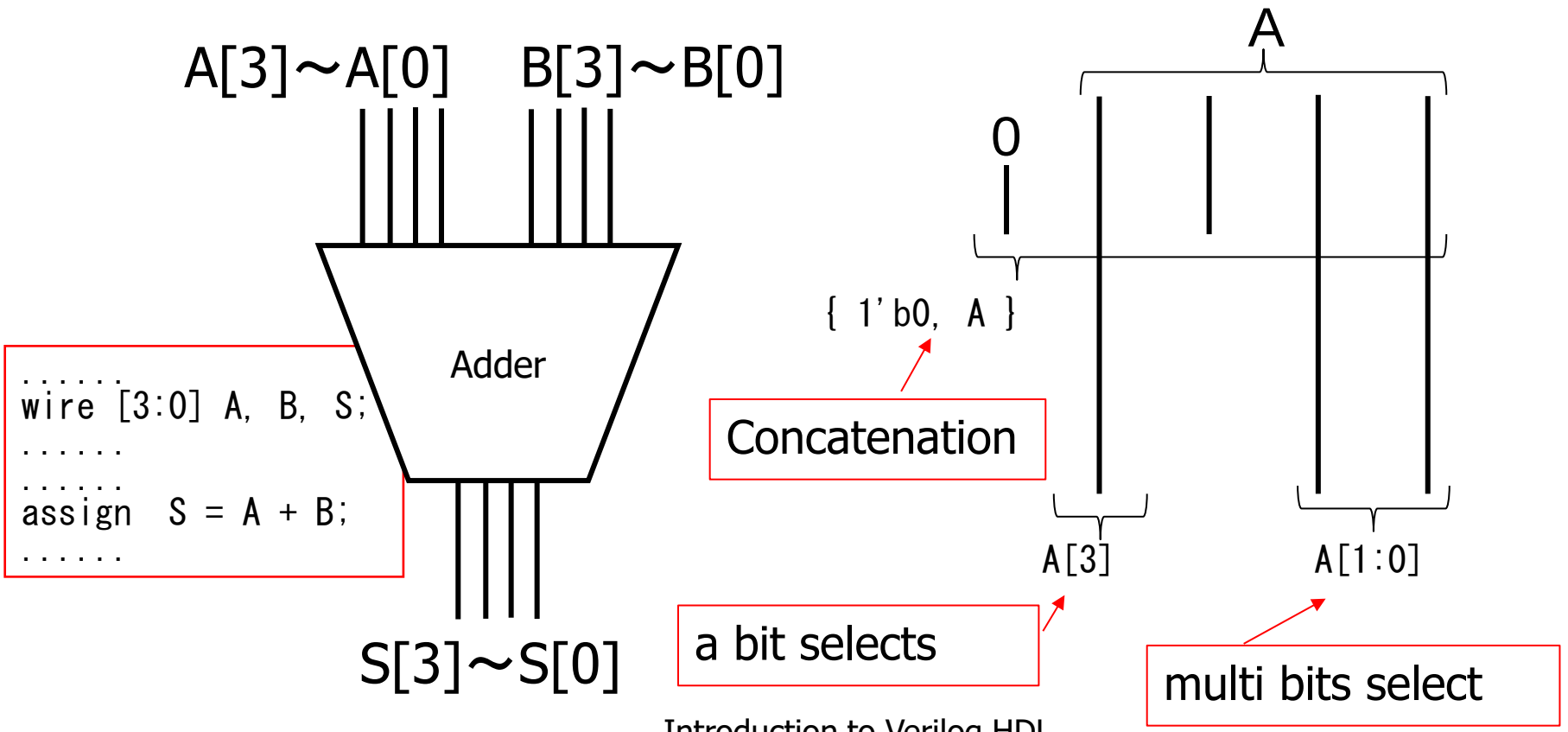
Constant Expression

- Binary
  - 8'b01010101 — 8-bit binary "01010101"
  - 8'b0101_0101 — Use "_" for readability
- Octal
  - 6'o65 — 6-bit binary "110101"
- Decimal
  - 4'd10 — 4-bit binary "1010"
  - 10 — 32-bit binary "0…01010"
- Hexadecimal
  - 8'ha5 — 8-bit binary "10100101"



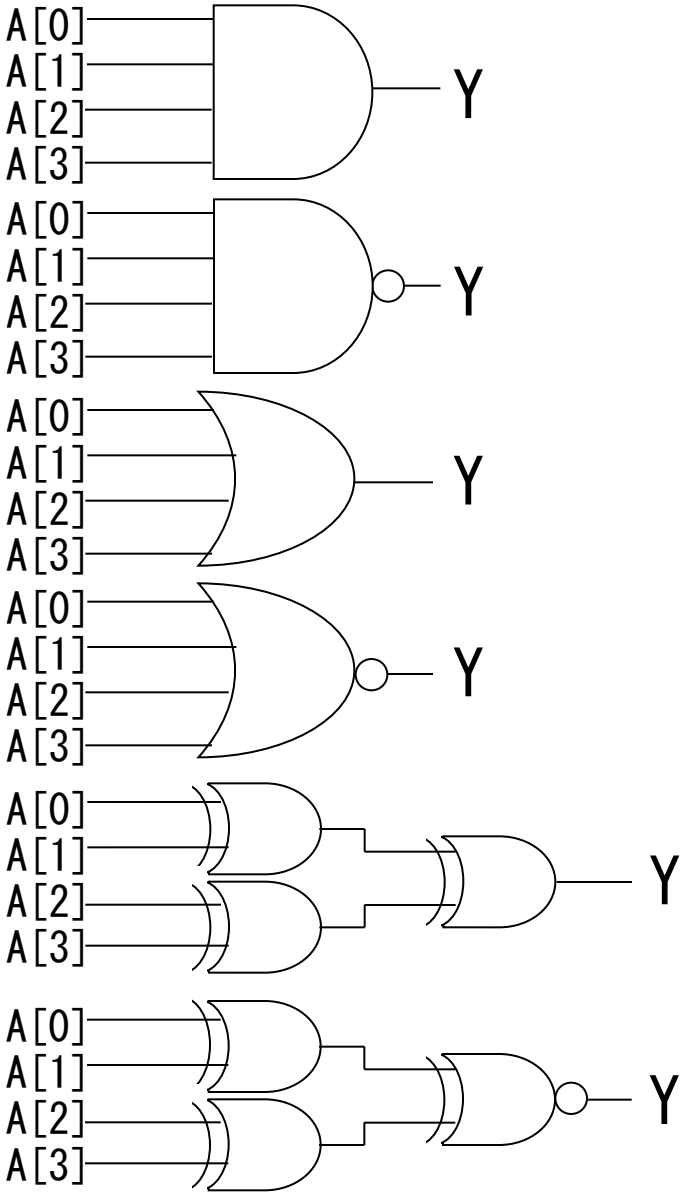
Bus Wire

- Define 4 lines, such as A[3], A[2], A[1], A[0] by "wire [3:0] A"
- 4-bit Adder is described simply by "S = A + B"

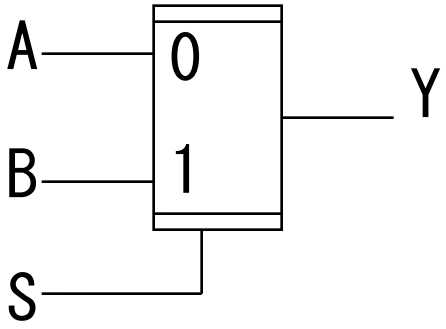


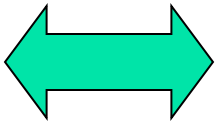
Reduction Operation (wire [3:0] A)

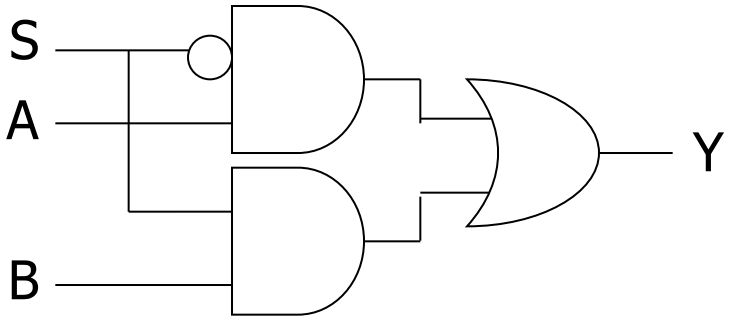
- AND assign Y=&A;
- NAND assign Y=~&A;
- OR assign Y=|A;
- NOR assign Y=~|A;
- ExOR assign Y=^A;
- ExNOR assign Y=~^A;



2-to-1 Multiplexer







assign statement

```
......
wire Y;
......
......
assign Y = ( S == 0 ) ? A : B;
......
......
```

Ternary operator as same as C

"assign" statement is used for wire type.

always statement

```
reg Y;

always @( A or B or S )
begin
  case( S )
  1'b0:    Y <= A;
  1'b1:    Y <= B;
  default: Y <= 1'bX;
  endcase
end
```

Assigned variable must be "reg" type

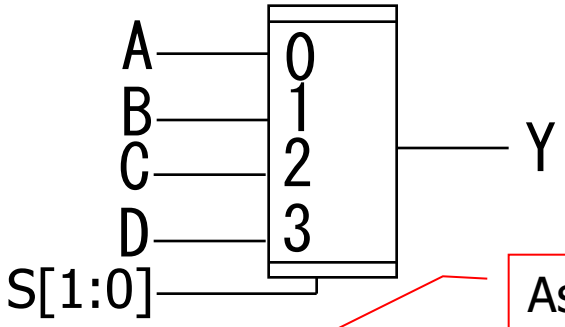
In sensitivity list is enumerated all referenced variable at always block

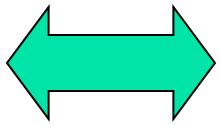
When S has unknown value,
Y is assigned the value in B.
It differs from simulation result and real behavior.

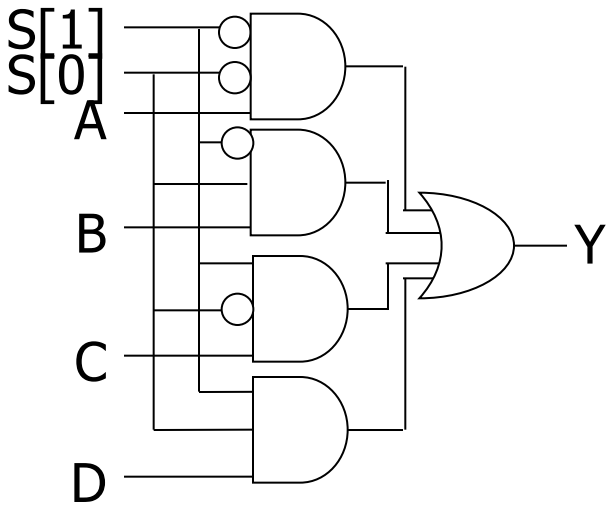
When S has unknown value,
Y is assigned unknown value 'X'.
It is effective for debugging.

"assign" statement is not used for "reg" type variable

4-to-1 Multiplexer








```
reg Y;
always @( A or B or C or D or S )
begin
   case( S )
   2'b00:   Y <= A;
   2'b01:   Y <= B;
   2'b10:   Y <= C;
   2'b11:   Y <= D;
   default: Y <= 1'bX;
   endcase
end
```

Assigned variable must be "reg" type

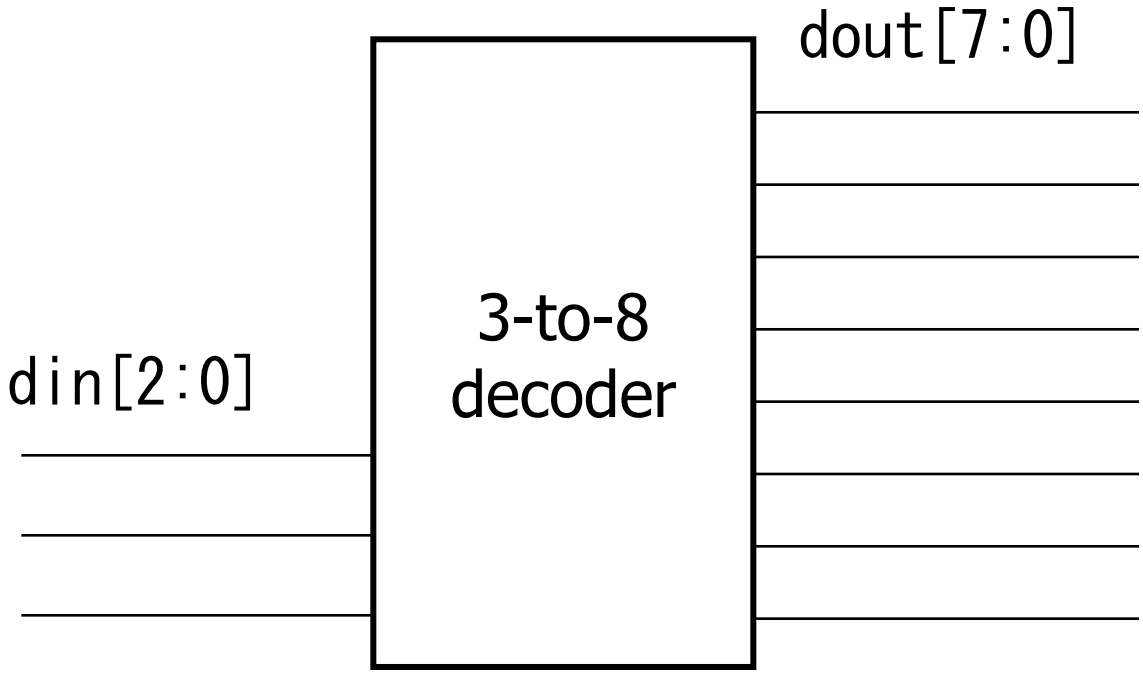
In sensitivity list is enumerated all referenced variable at always block

When S has unknown value, Y is assigned unknown value 'X'. It is effective for debugging.

"assign" statement is not used for "reg" type variable

Decoder

- 3-to-8 Decoder
  - A value '1' is assigned to only one output line in corresponding input value.



3-to-8 Decoder (case and if statements)

case statement

```
module decoder (din, dout);
  input  [2:0] din;
  output [7:0] dout;

  reg    [7:0] dout;

  always @( din )
  begin
    case ( din )
    3'b000: dout <=8'b0000_0001;
    3'b001: dout <=8'b0000_0010;
    3'b010: dout <=8'b0000_0100;
    3'b011: dout <=8'b0000_1000;
    3'b100: dout <=8'b0001_0000;
    3'b101: dout <=8'b0010_0000;
    3'b110: dout <=8'b0100_0000;
    3'b111: dout <=8'b1000_0000;
    default:dout <=8'bXXXX_XXXX;
    endcase
  end

endmodule
```

Assigned variable must be "reg" type in "always" block.

if statement

```
module decoder (din, dout);
 input  [2:0] din;
 output [7:0] dout;

 reg    [7:0] dout;

 always @( din )
 begin

  if(dout==3'b000)dout<=8'b0000_0001;else
  if(dout==3'b001)dout<=8'b0000_0010;else
  if(dout==3'b010)dout<=8'b0000_0100;else
  if(dout==3'b011)dout<=8'b0000_1000;else
  if(dout==3'b100)dout<=8'b0001_0000;else
  if(dout==3'b101)dout<=8'b0010_0000;else
  if(dout==3'b110)dout<=8'b0100_0000;else
  if(dout==3'b111)dout<=8'b1000_0000;else
                  dout<=8'bXXXX_XXXX;

 end

endmodule
```

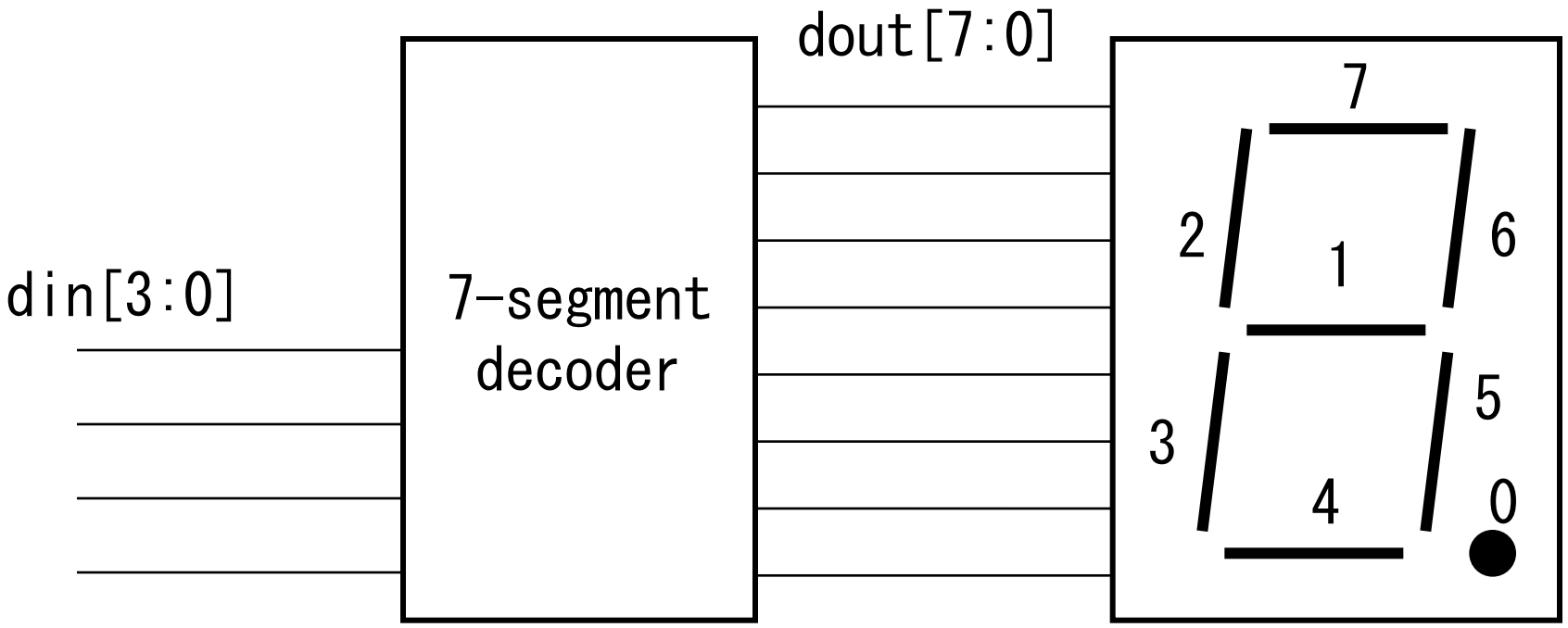
Assigned variable must be "reg" type

In sensitivity list is enumerated all referenced variable at always block

When "din" has unknown value, "dout" is assigned unknown value "X".
It is effective for debugging.

7-segment Decoder

- 7-segment decorder



7-segment Decoder

```
module seven_seg(din,dout);
 input  [3:0] din;
 output [7:0] dout;

 reg    [7:0] dout;

 always @( din )
 begin
  case ( din )
   4'b0000:dout<=8'b11111100;
   4'b0001:dout<=8'b01100000;
   ......
   ......
   4'b1110:dout<=8'b10011110;
   4'b1111:dout<=8'b10001110;
   default:dout<=8'bXXXXXXXX;
  endcase
 end
endmodule
```

Assigned variable must be "reg" type in "always" block.

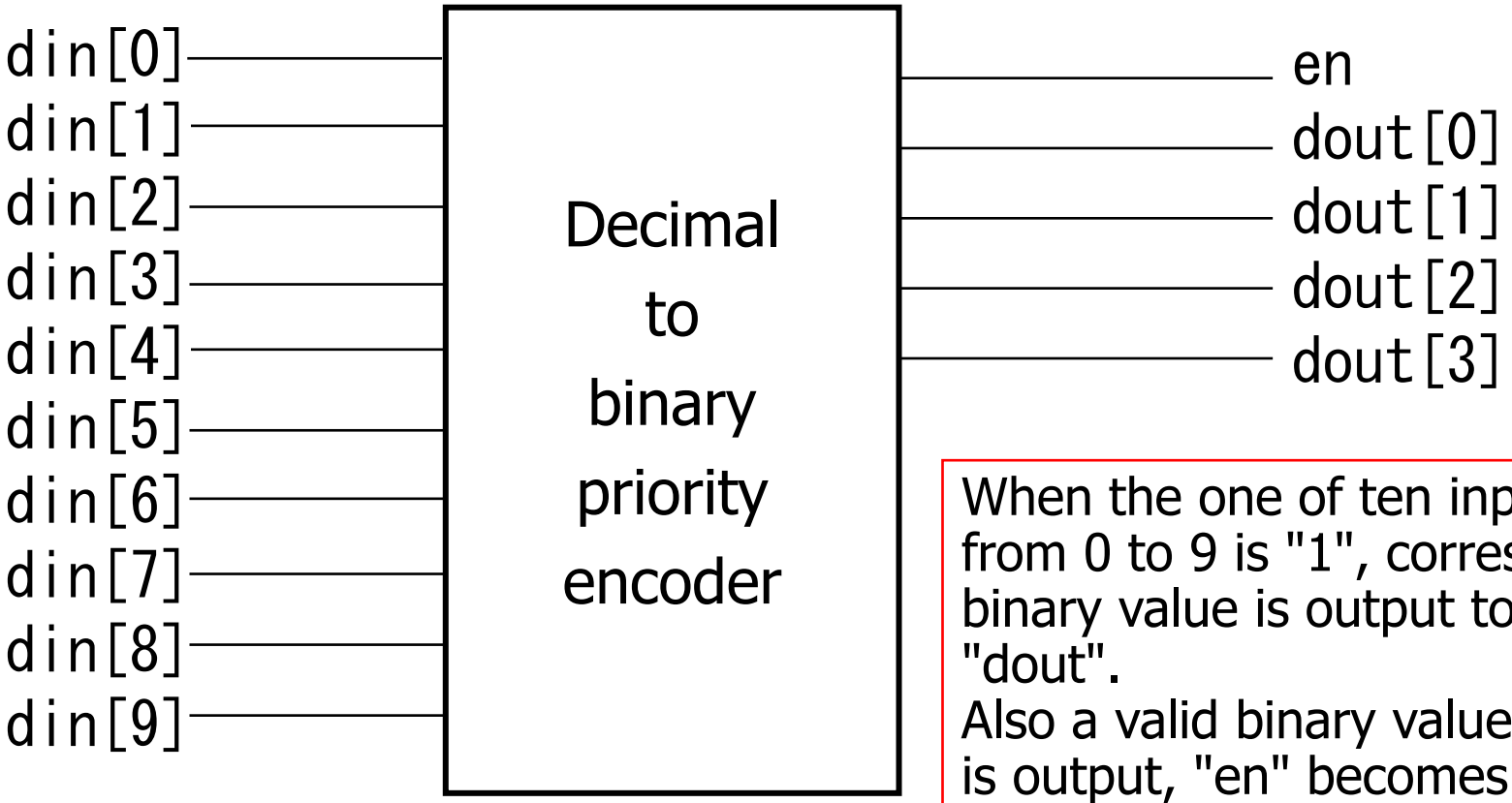
In sensitivity list is enumerated all referenced variable at always block

When "din" have unknown value, "dout" is assigned unknown value "X". It is effective for debugging.

Assigned variable must be "reg" type in "always" block.

Priority Encoder

- Decimal to binary priority encoder



When the one of ten inputs "din" from 0 to 9 is "1", corresponding binary value is output to the "dout".
Also a valid binary value to "dout" is output, "en" becomes "1".

Priority Encoder

You should use "casex" statement to make a prioritized circuit.

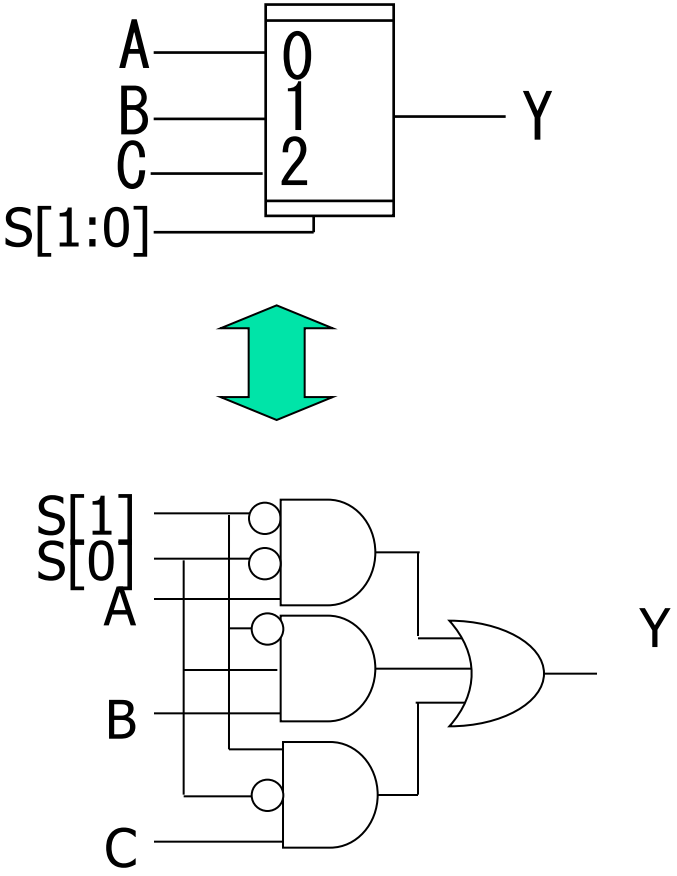
```
module priority_encoder (din, dout, en);
  input  [9:0] din;
  output [3:0] dout;
  output       en;
  reg    [3:0] dout;
  reg          en;
  always @( din )
  begin
    casex ( din )
    10'b?????_????1: { en, dout } <=5'b1_0000;
    10'b?????_???10: { en, dout } <=5'b1_0001;
    10'b?????_??100: { en, dout } <=5'b1_0010;
    10'b?????_?1000: { en, dout } <=5'b1_0011;
    10'b?????_10000: { en, dout } <=5'b1_0100;
    10'b????1_00000: { en, dout } <=5'b1_0101;
    10'b???10_00000: { en, dout } <=5'b1_0110;
    10'b??100_00000: { en, dout } <=5'b1_0111;
    10'b?1000_00000: { en, dout } <=5'b1_1000;
    10'b10000_00000: { en, dout } <=5'b1_1001;
    default:         { en, dout } <=5'b0_XXXX;
    endcase
  end
endmodule
```

Assigned variable must be "reg" type in "always" block.

In sensitivity list is enumerated all referenced variable at always block

3-to-1 Multiplexer ①

- Correctness ①



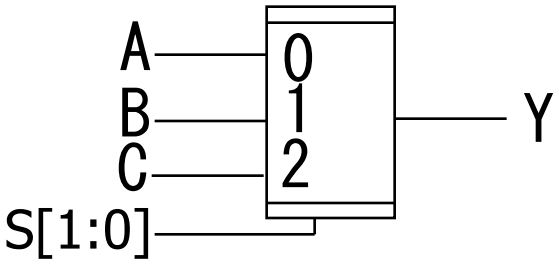
Y<=0 when S == 2'b11

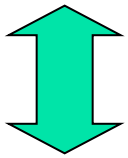
```
reg Y;
always @( A or B or C or S )
begin
  case( S )
  2'b00:   Y <= A;
  2'b01:   Y <= B;
  2'b10:   Y <= C;
  default: Y <= 1'b0;
  endcase
end
```

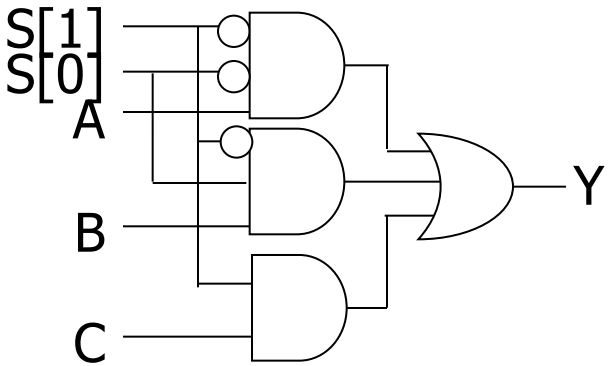
In this case, synthesized logic circuit is in the left side figure.

3-to-1 Multiplexer ②

- Correctness ②







Don't care

```
reg Y;
always @( A or B or C or S )
begin
  case( S )
  2'b00:   Y <= A;
  2'b01:   Y <= B;
  2'b10:   Y <= C;
  default: Y <= 1'bX;
  endcase
end
```

"Y" becomes X, when "S=2'b11" by default.

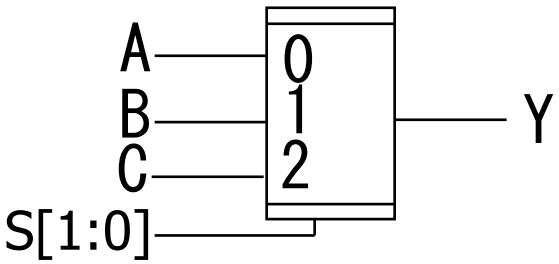
default

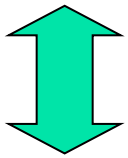
```
reg Y;
always @( A or B or C or S )
begin
  case( S )
  2'b00:   Y <= A;
  2'b01:   Y <= B;
  default: Y <= C;
  endcase
end
```

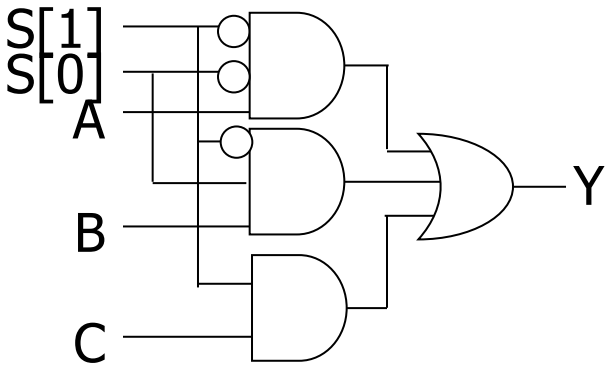
"Y" becomes C, when "S=2'b1X" by default.

3-to-1 Multiplexer ③

- Correctness ③





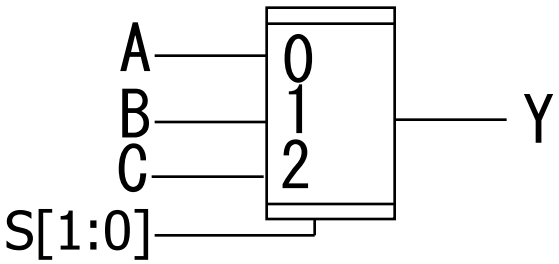


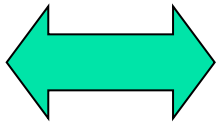
casex statement

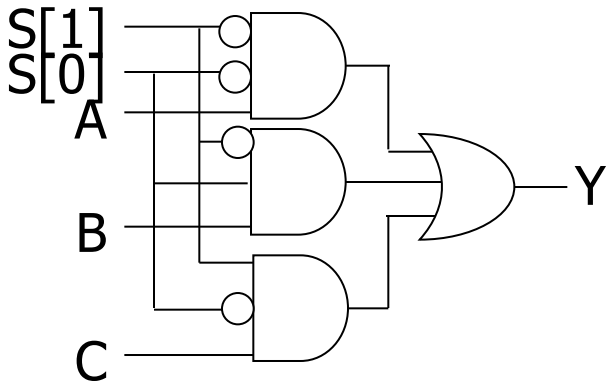
```
reg Y;
always @( A or B or C or S )
begin
  casex( S )
  2'b00:    Y <= A;
  2'b01:    Y <= B;
  2'b1?:    Y <= C;
  default: Y <= 1'bX;
  endcase
end
```

S[0] is treated as "Don't Care" by 2'b1?.
Also "Y" becomes X, when "Y<=1'bX" by default.

3-to-1 Multiplexer ④







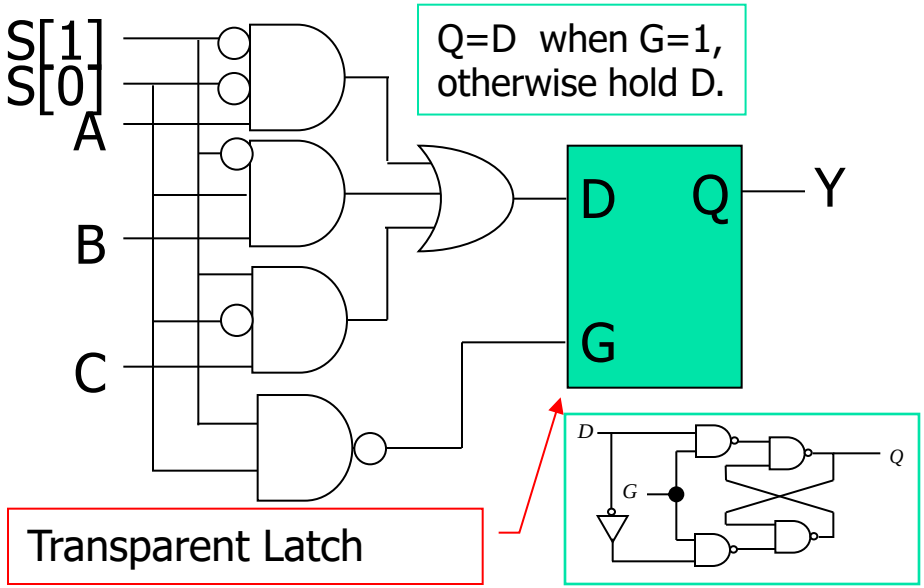
Expected circuit

Mis-description

```
reg Y;
always @( A or B or C or S )
begin
   case( S )
   2'b00: Y <= A;
   2'b01: Y <= B;
   2'b10: Y <= C;

   endcase
end
```

without default

Transparent latch is synthesized because of holding a just before value when S becomes 2'b11.



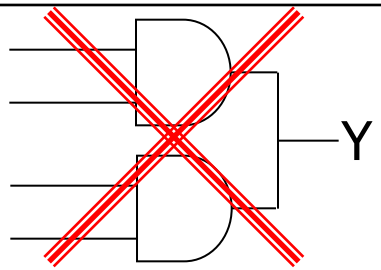
Synthesized circuit from mis-description

Fallible Description (inter always assignment)

Error

```
reg Y;

always @( ... )
begin
  Y <= ...;
end

always @( ... )
begin
  Y <= ...;
end
```

Assign to same Y from multiple always blocks

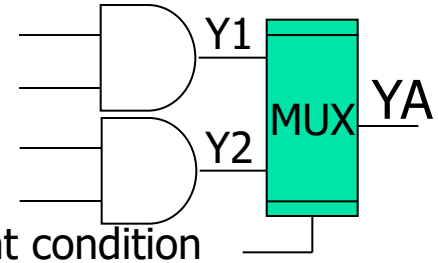


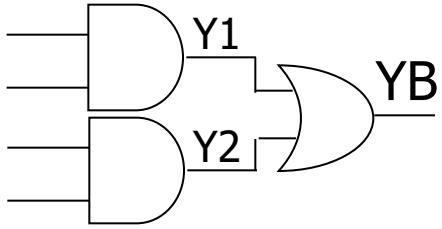
It makes short circuit.
It is correct in syntax and can be simulated.
However, this description is not synthesizable.

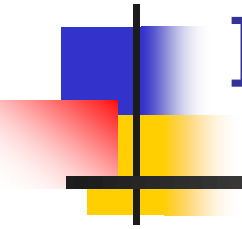
Correctness

```
wire YA, YB;
reg  Y1, Y2;

always @( ... )
begin
  Y1 <= ...;
end

always @( ... )
begin
  Y2 <= ...;
end

assign YA = ( 代入条件 ) ? Y1 : Y2;
assign YB = Y1 | Y2;
```

(1) Assign to differ variable (YA, YB) in always block, and select either one by assignment condition.
(2) Inclusive OR is acceptable if no problem its behavior.







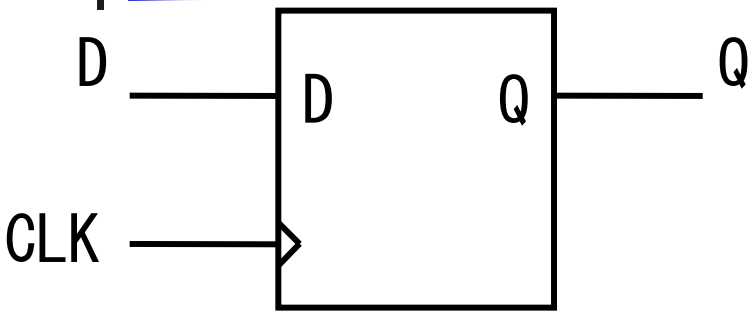
Introduction to Verilog HDL

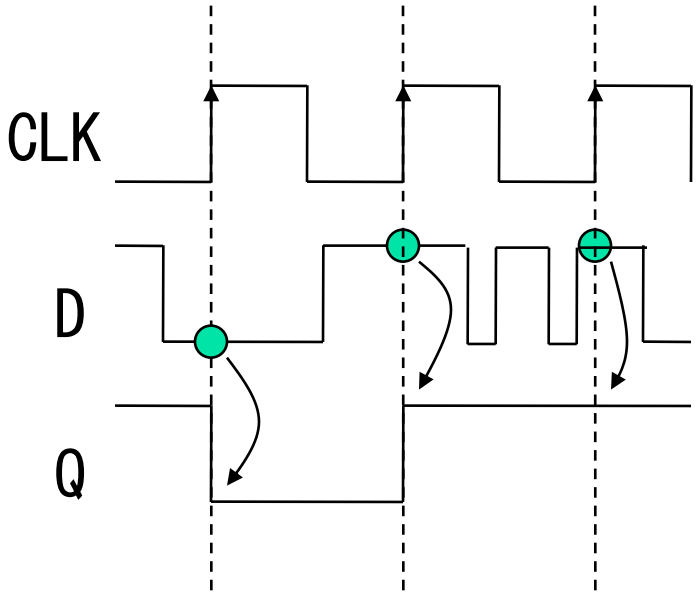
Flip-Flop

Register

Counter

D-Flip Flop (1)



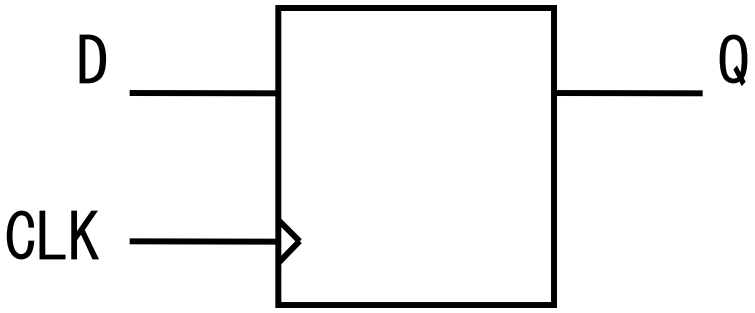


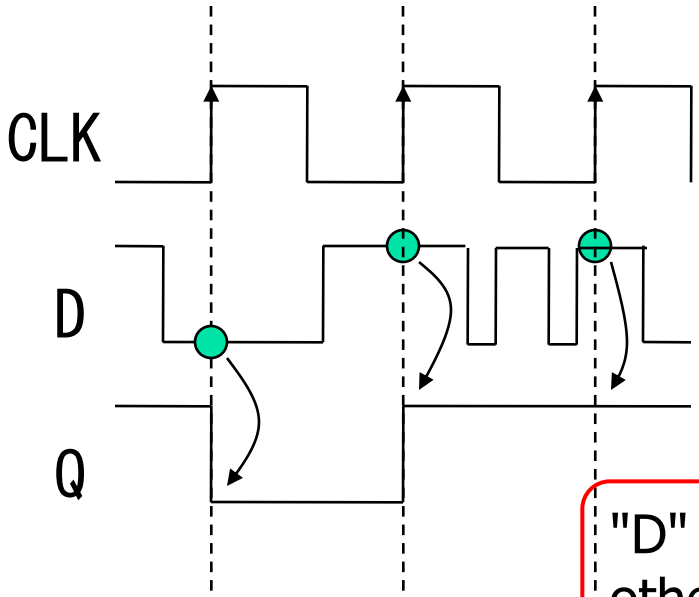

```
module dff ( D, Q, CLK );
  input  D;
  input  CLK;
  output Q;

  reg    tmp; // for Register

  always @( posedge CLK )
    begin
      tmp <= D;
    end

  assign Q = tmp;

endmodule
```

Positive edge, "negedge" for negative edge

"D" is assigned to "tmp" when positive edge clock, otherwise "tmp" never changes.

D-Flip Flop (2)

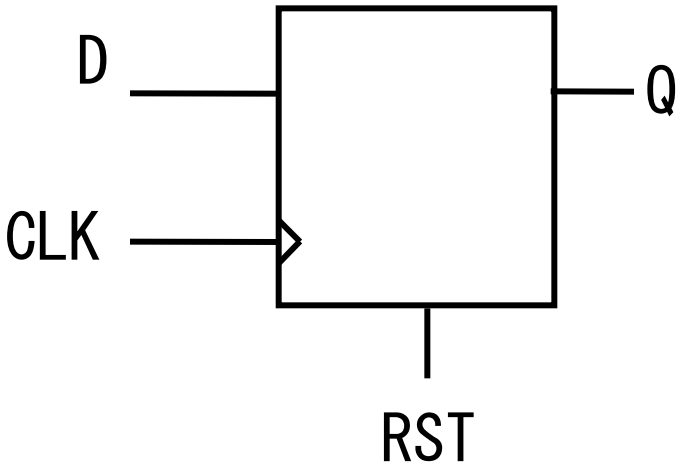


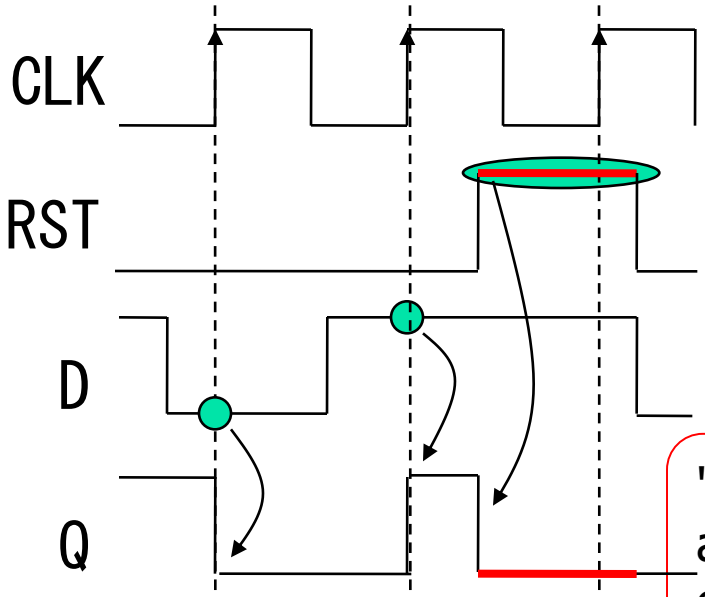



```
module dff ( D, Q, CLK );
  input  D;
  input  CLK;
  output Q;

  reg    Q;

  always @( posedge CLK )
    begin
      Q <= D;
    end

endmodule
```

"D" is assigned to "Q" when positive edge clock, otherwise "Q" never changes.

D-Flip Flop with Asynchronous Reset



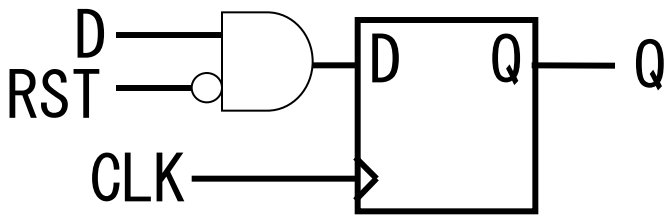


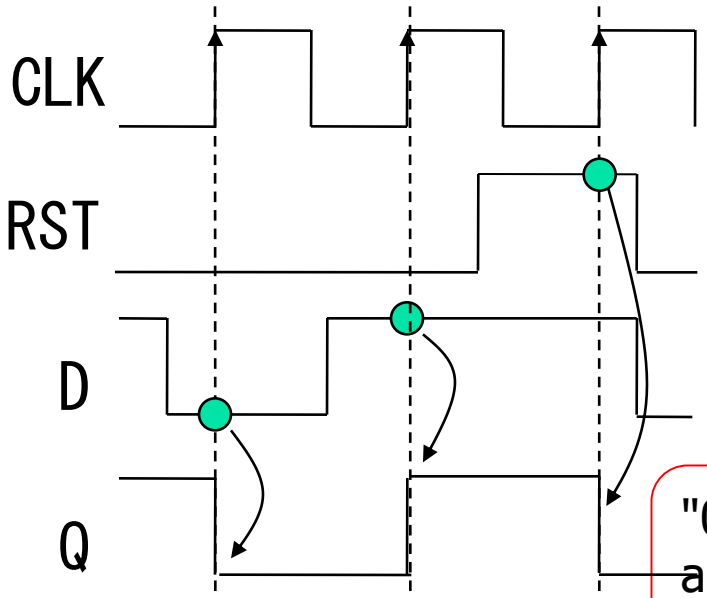

```
module dff ( D, Q, CLK, RST );
  input  D, CLK, RST;
  output Q;

  reg    tmp; // for Register

  always @(posedge RST or posedge CLK)
  begin
    if ( RST ) tmp <= 1'b0; else
               tmp <= D;
  end

  assign Q = tmp;

endmodule
```

Reset condition

"0" is assigned to "tmp" when RST is active, else "D" assigned to "tmp" when positive edge clock, otherwise "tmp" never changes.

D-Flip Flop with Synchronous Reset





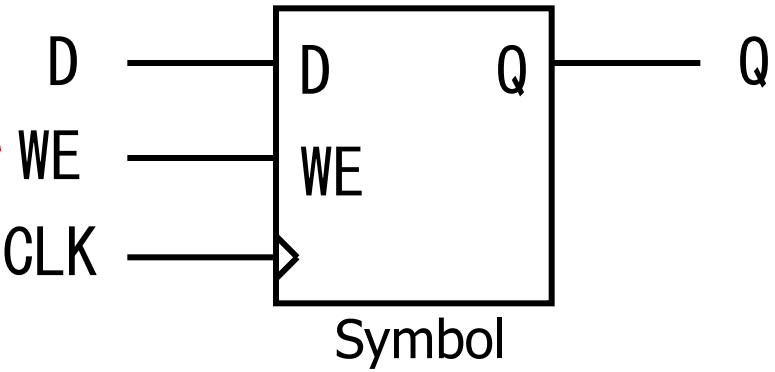

```
module dff ( D, Q, CLK, RST );
  input  D, CLK, RST;
  output Q;

  reg    tmp; // for Register

  always @( posedge CLK )
  begin
    if ( RST ) tmp <= 0; else
               tmp <= D;
  end

  assign Q = tmp;

endmodule
```

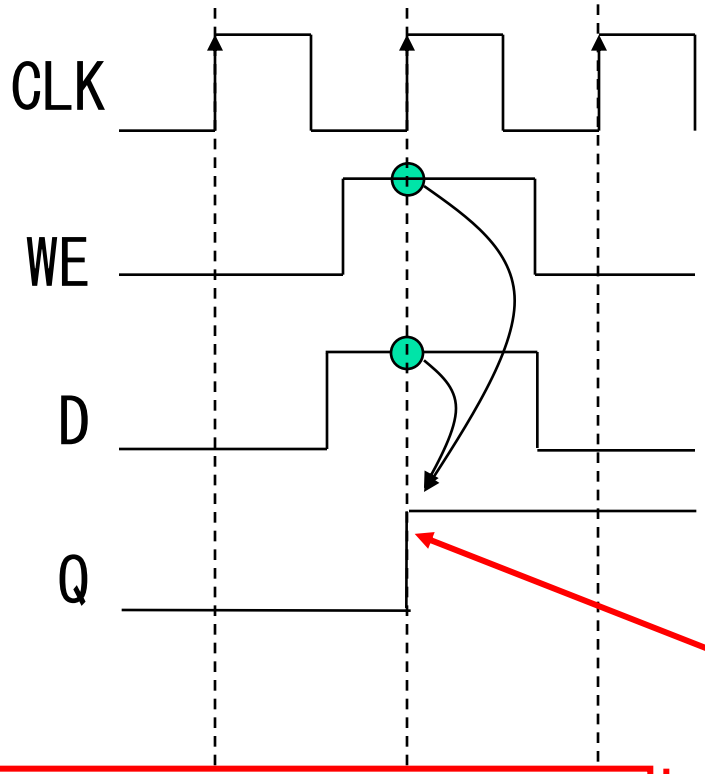
When Reset

"0" is assigned to "tmp" when positive edge clock and active RST, else "D" assigned to "tmp" when positive edge clock, otherwise "tmp" never changes.

D-Flip Flop with Write Enable

WE: Write Enable WE
same as CE (Clock Enable)



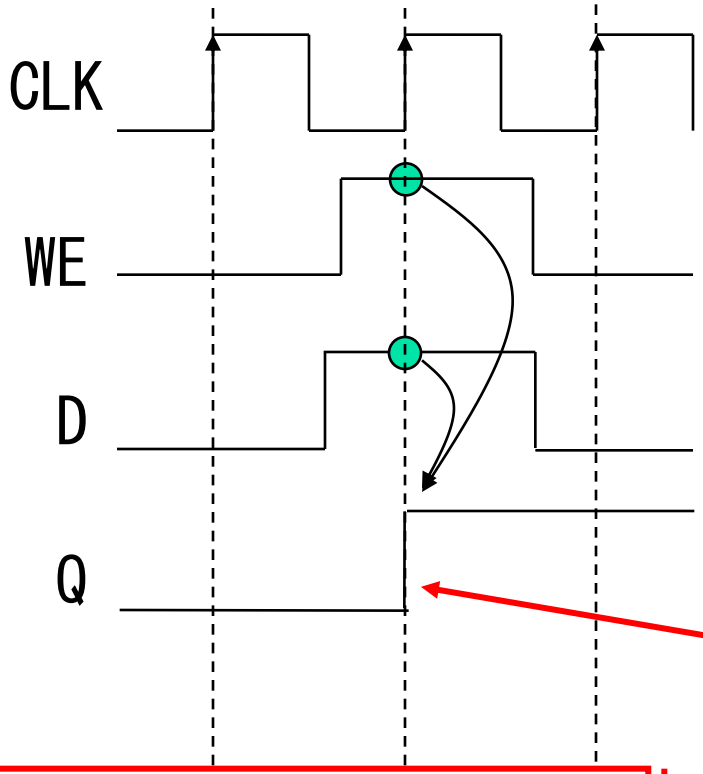


D is held in Q when active
WE and positive edge clock.

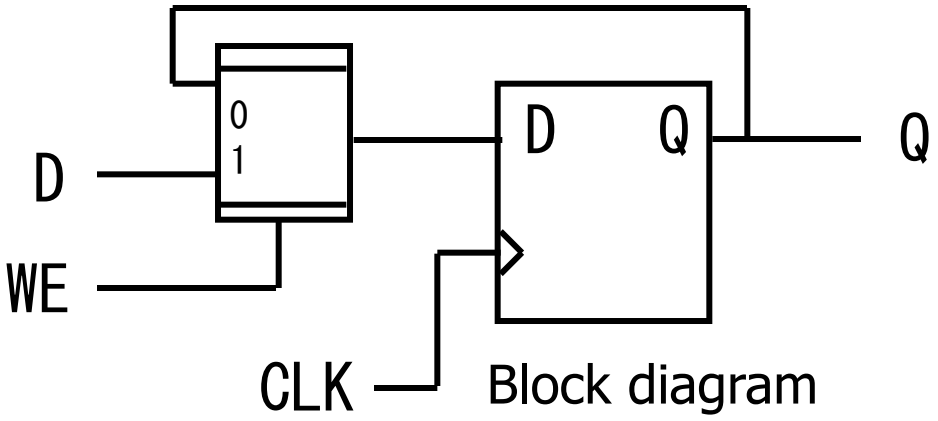
```
module dff_we ( D,WE,Q,CLK );
  input  D;
  input  WE;
  input  CLK;
  output Q;

  reg    tmp;

  always @( posedge CLK )
  begin
    if( WE ) tmp <= D;
  end

  assign Q = tmp;
endmodule
```

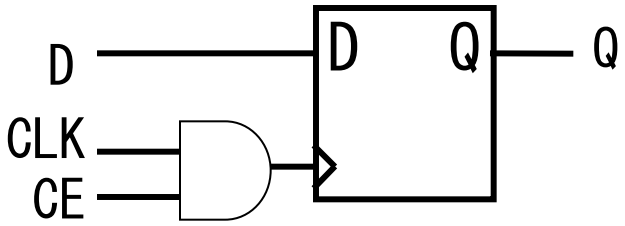
Don't include WE to sensitivity list

D-Flip Flop with Write Enable



D is held in Q when active WE and positive edge clock.

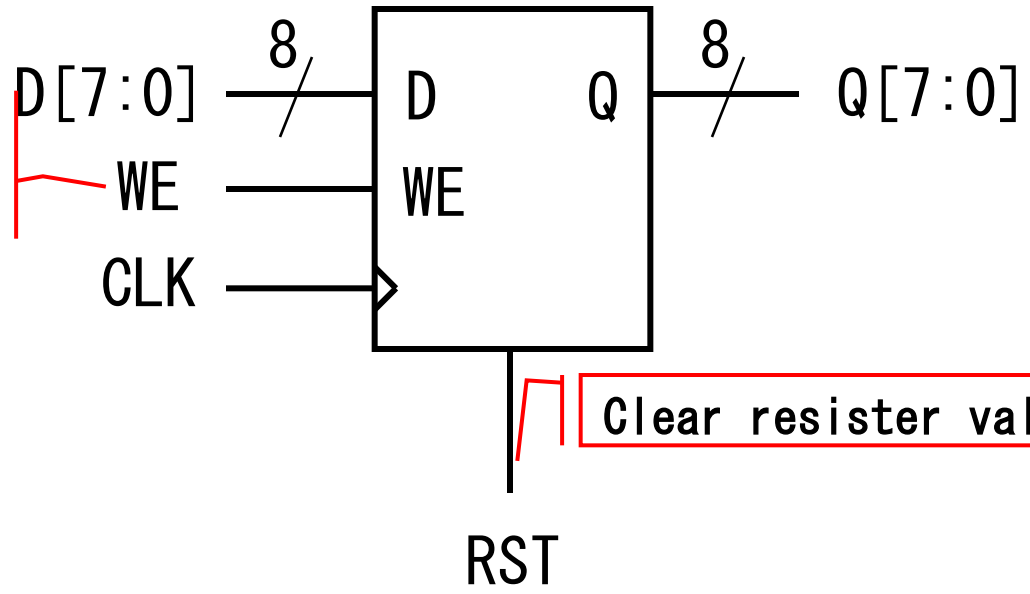


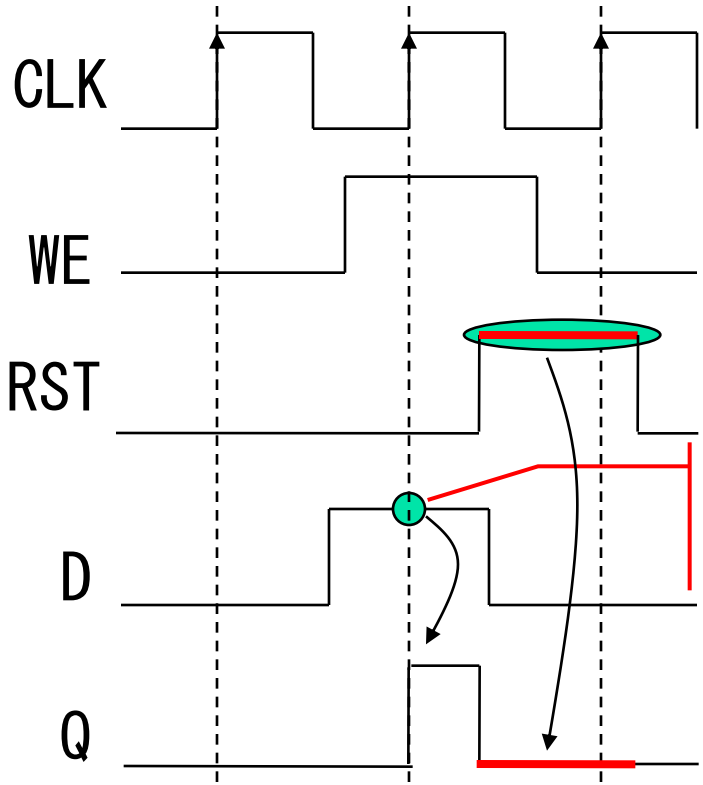


Generally, gated clock circuit is not used in synchronous circuit design, although it used aggressive low-power consumption circuit design.

8-bit Register

WE: Write Enable WE
same as CE (Clock Enable)





D is held in Q when active
WE and positive edge clock.

8-bit Register

8-bit register with asynchronous reset

```
module reg8_ar ( D, WE, RST, Q, CLK );
  input  [7:0] D;
  input        WE;
  input        RST;
  input        CLK;
  output [7:0] Q;

  reg    [7:0] tmp;

  always @(posedge RST or posedge CLK)
  begin
    if ( RST ) tmp <= 8'h00; else
    if ( WE  ) tmp <= D;
  end

  assign Q = tmp;
endmodule
```

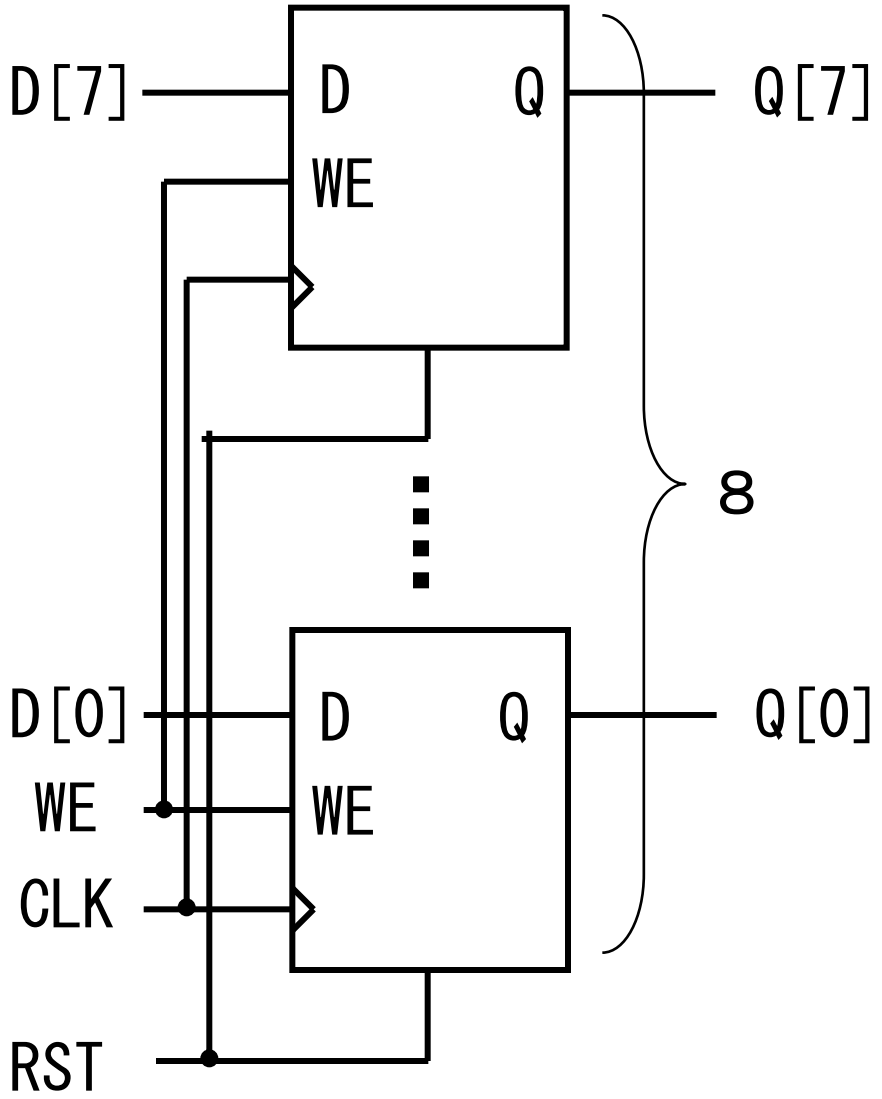
8-bit register with synchronous reset

```
module reg8_sr ( D,WE,RST,Q,CLK );
  input  [7:0] D;
  input        WE;
  input        RST;
  input        CLK;
  output [7:0] Q;

  reg    [7:0] tmp;

  always @(posedge CLK)
  begin
    if( RST ) tmp<=8'h00; else
    if( WE  ) tmp<=D;
  end

  assign Q = tmp;
endmodule
```

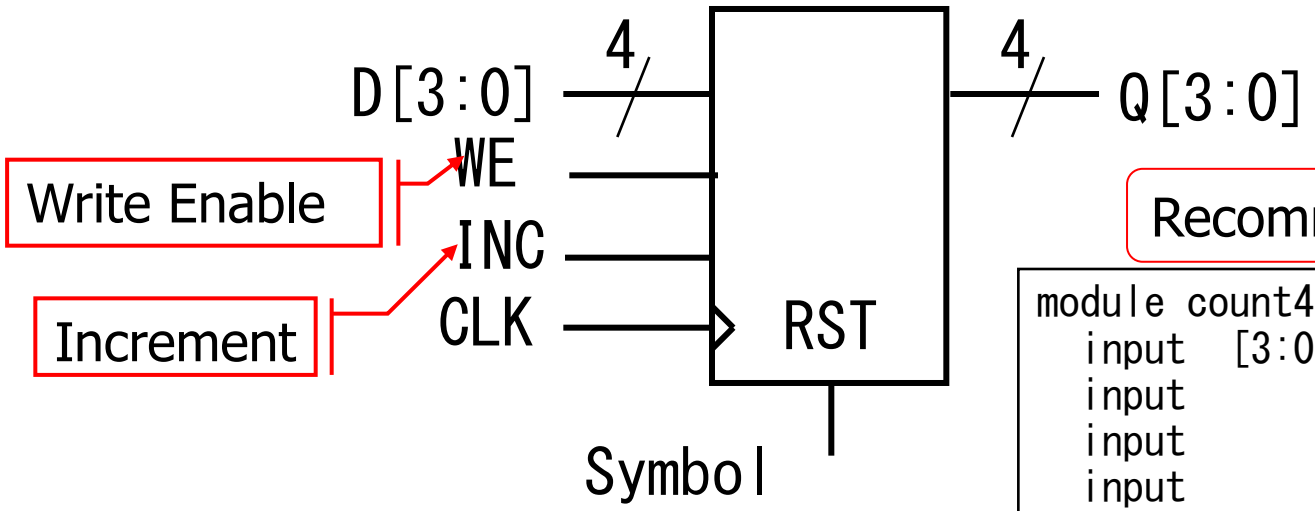
D is held in Q when active WE and positive edge clock.

Schematic of 8-bit Register

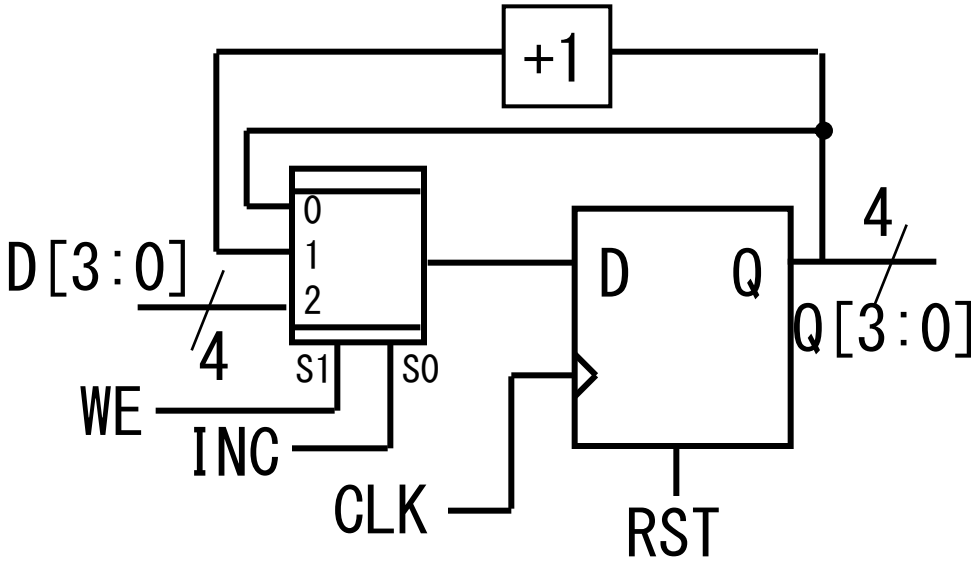
8-bit register with asynchronous reset



Pre-settable 4-bit Binary Counter (1)



Symbol

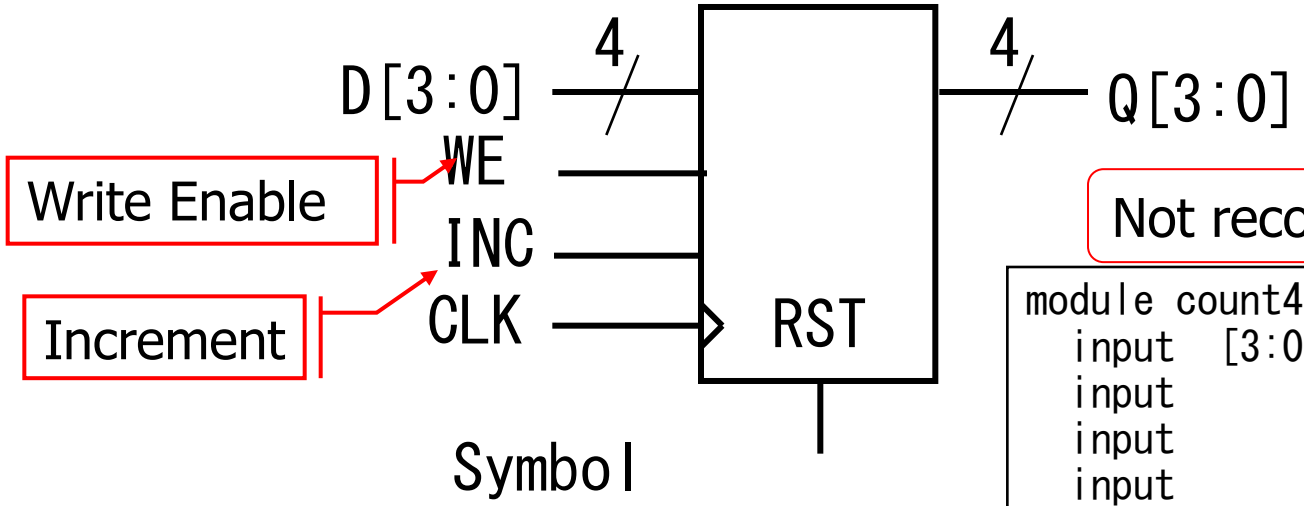


Block diagram

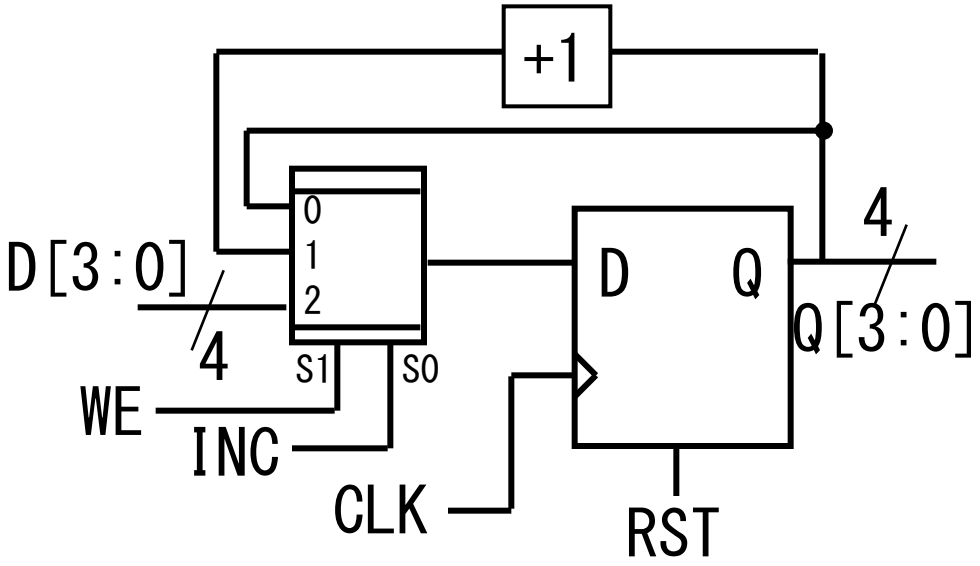
Recommended this style

```
module count4 ( D,WE,INC,RST,Q,CLK );
  input  [3:0] D;
  input        WE;
  input        INC;
  input        RST;
  input        CLK;
  output [3:0] Q;

  reg    [3:0] tmp;  // for Register

  always @(posedge RST or posedge CLK)
  begin
    if ( RST == 1 ) tmp <= 4'h0; else
    if ( WE  == 1 ) tmp <= D;    else
    if ( INC == 1 ) tmp <= tmp + 1;
  end
  assign Q = tmp;
endmodule
```

Pre-settable 4-bit Binary Counter (2)



Symbol



Block diagram

Not recommended this style

```
module count4 ( D,WE,INC,RST,Q,CLK );
  input  [3:0] D;
  input        WE;
  input        INC;
  input        RST;
  input        CLK;
  output [3:0] Q;

  reg    [3:0] Q;  // for Register

  always @(posedge RST or posedge CLK)
  begin
    if ( RST == 1 ) Q <= 4'h0; else
    if ( WE  == 1 ) Q <= D;    else
    if ( INC == 1 ) Q <= Q + 1;
  end

endmodule
```

Non-blocking and Blocking assignment

- Non-blocking assignment
  - Use "<=" sign
  - Assignment don't take place immediately
  - Assignment takes place after a Δ delay
  - Δ delay: infinitesimal time for accuracy simulation
- Blocking assignment
  - Use "=" sign
  - Assignment takes place immediately w/o Δ delay
  - Following assignment can read the previous assignment.

Note that non-blocking and blocking assignments cannot be mixed in one always statement.

Ring Counter

Non-blocking assignment

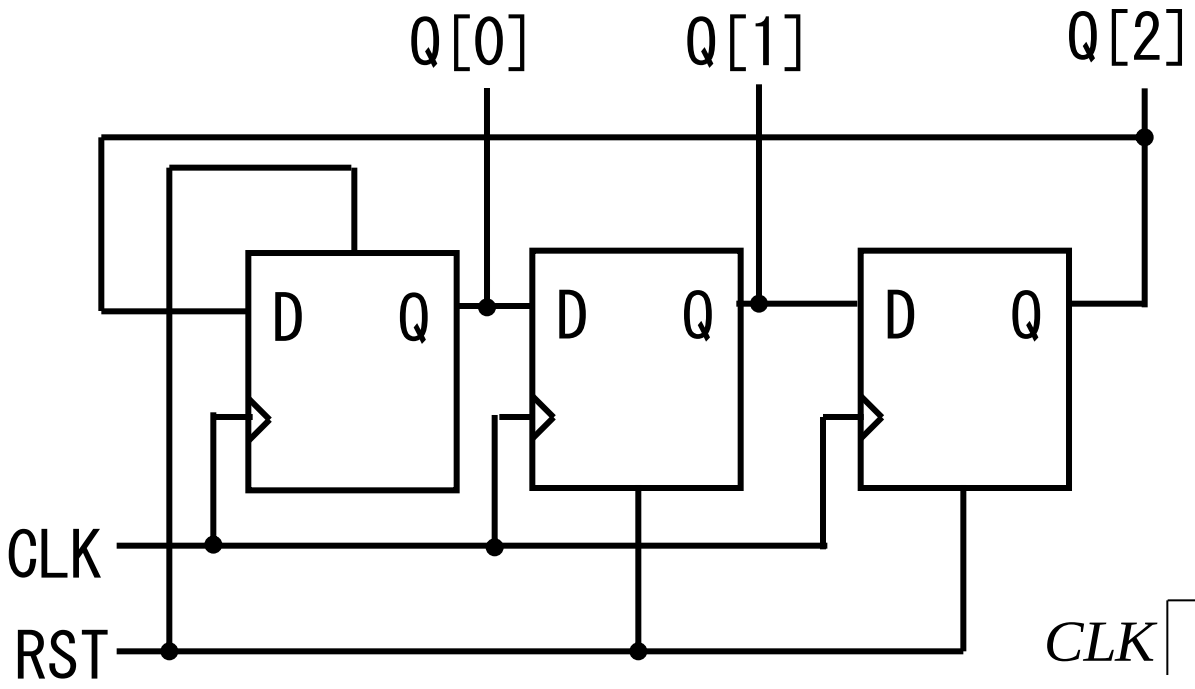
```
module nonblocking ( CLK, RST, Q );

input        CLK, RST;
output [2:0] Q;

reg    [2:0] tmp;

  always @( posedge CLK or posedge RST)
  begin
    if( RST ) tmp <= 3'b001;
    else
      begin
        tmp[1] <= tmp[0];
        tmp[2] <= tmp[1];
        tmp[0] <= tmp[2];
      end
  end

  assign Q = tmp;
endmodule
```

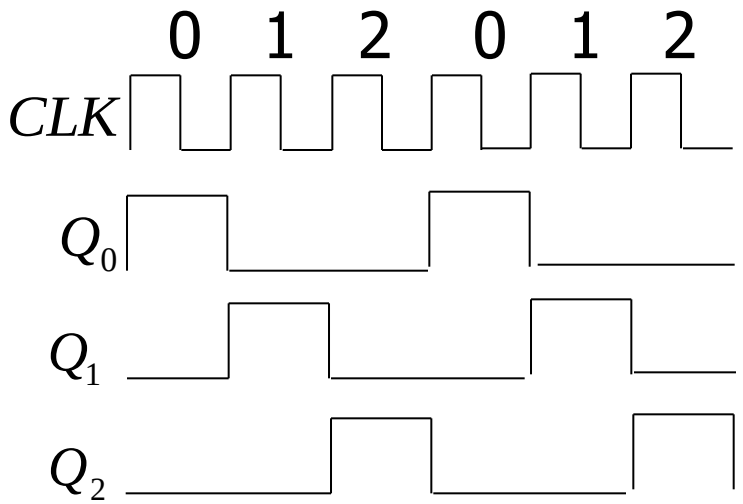
Blocking assignment

```
module blocking ( CLK, RST, Q );

input        CLK, RST;
output [2:0] Q;

reg    [2:0] tmp;

  always @( posedge CLK or posedge RST)
  begin
    if( RST ) tmp = 3'b001;
    else
      begin
        tmp[1] = tmp[0];
        tmp[2] = tmp[1];
        tmp[0] = tmp[2];
      end
  end

  assign Q = tmp;
endmodule
```

- Ring counter is correctly synthesized by non-blocking assignment.
- Assignments are taken place simultaneously at the end of always block.
- In blocking assignment, tmp[0] value is immediately assigned to tmp[1] and tmp[2]. In this result, the synthesized circuit can't behave as a ring counter.
- Non-blocking assignment is recommended in "always block" to prevent a mistaking design.
- Note the use of blocking assignment although there are effective cases.

Schematic of Ring Counter





BCD Adder

Non-blocking assignment

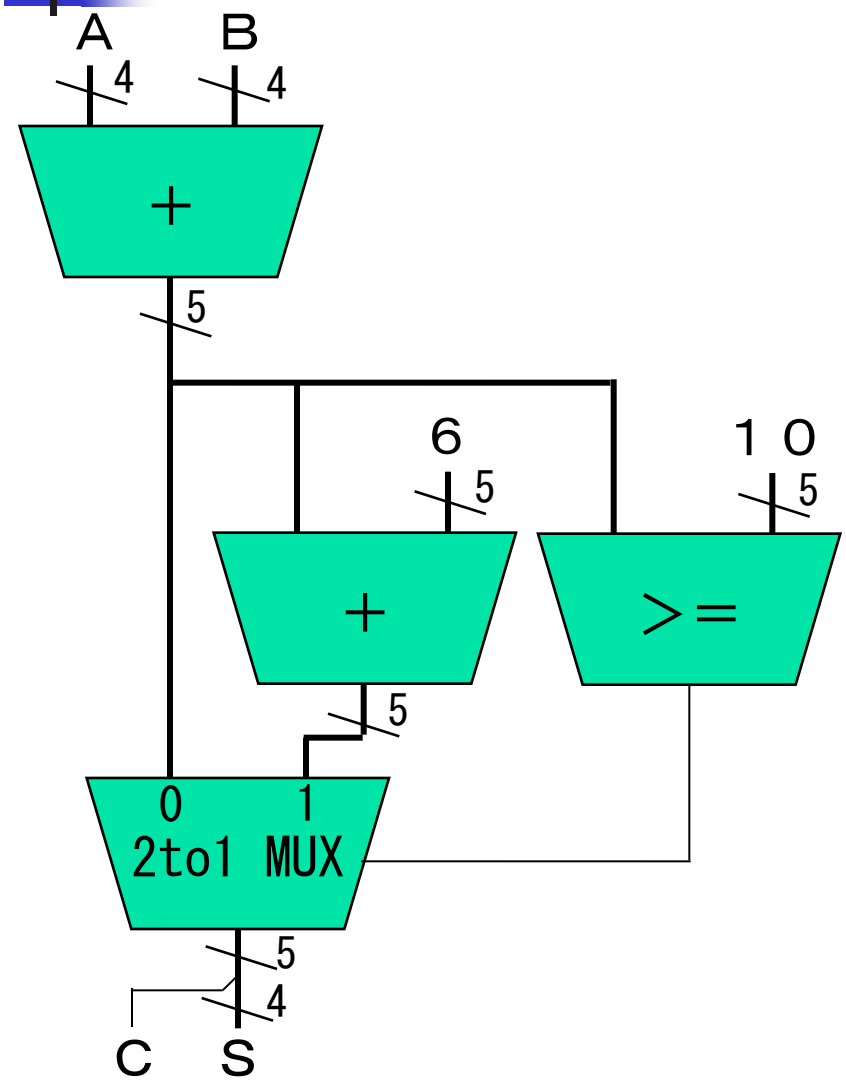
```
module bcd_addr ( A, B, S, C );
  input  [3:0] A, B;
  output [3:0] S;
  output       C; // Carry
  reg    [4:0] tmp;
  always @( A or B )
  begin
    tmp <= { 1'b0, A } + { 1'b0, B };
    if( tmp >= 5'b01010 )
        tmp <= tmp + 5'b00110;
  end
  assign S = tmp[3:0];
  assign C = tmp[4];
endmodule
```

Blocking assignment

```
module bcd_addr ( A, B, S, C );
  input  [3:0] A, B;
  output [3:0] S;
  output       C; // Carry
  reg    [4:0] tmp;
  always @( A or B )
  begin
    tmp = { 1'b0, A } + { 1'b0, B };
    if( tmp >= 5'b01010 )
        tmp = tmp + 5'b00110;
  end
  assign S = tmp[3:0];
  assign C = tmp[4];
endmodule
```

- BCD adder is correctly synthesized by blocking assignment.
- In blocking assignment, tmp value is immediately assigned. You can see properly immediately after.
- In non-blocking assignment, not correctly behaves because assignments are taken place simultaneously at the end of always block.
- This case is only described by blocking assignment.

Block Diagram of BCD Adder



Block diagram

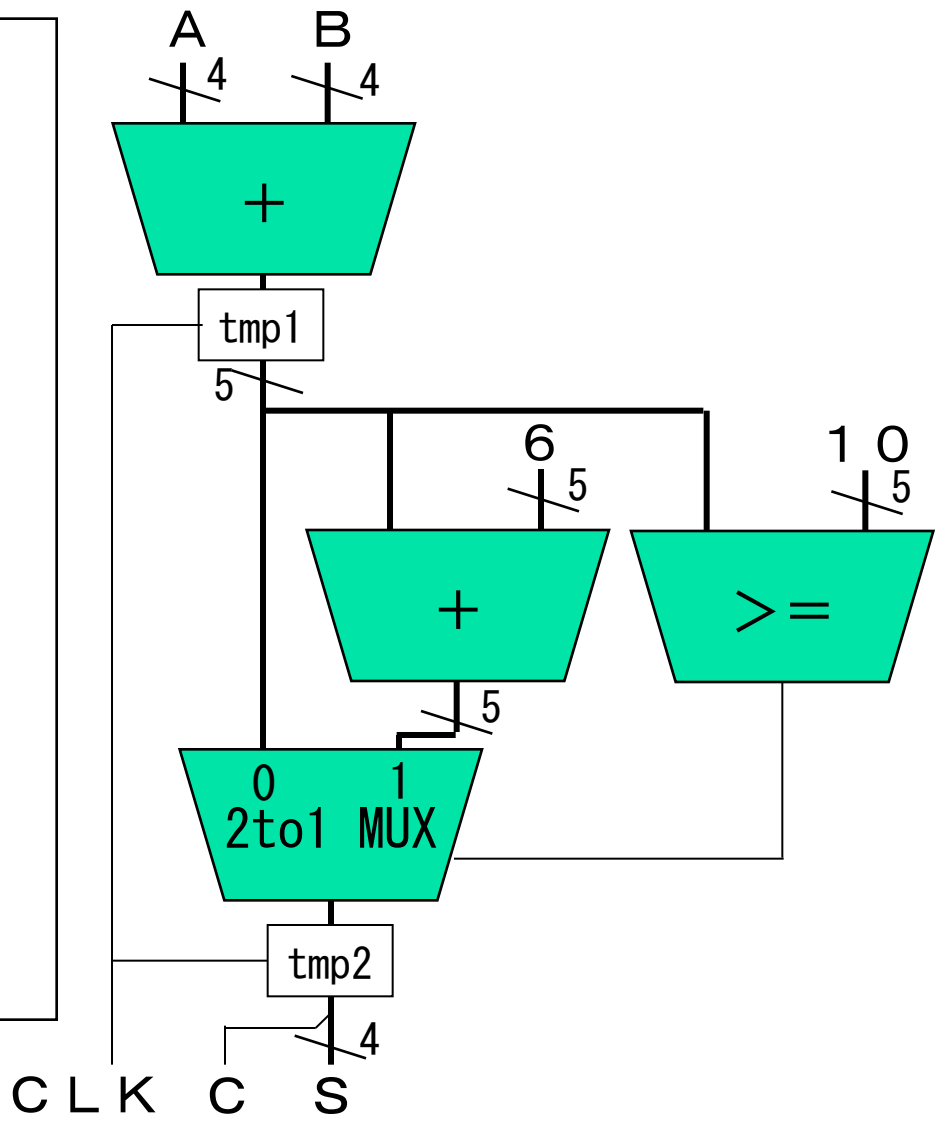
BCD adder adds 2 values of binary coded decimal number. First, A and B are added by binary adder. If the result is from 0 to 9, it is correct. However, when the result is $(A)_{16}$~$(F)_{16}$, the result has to be correction so that the result becomes to BCD value. For correction, constant 6 adds to the result. For example, the result is $(A)_{16}$ when A adds B, then the result of adding 6 is $(10)_{16}$ or $(10)_{10}$ of BCD.

Similarly, when the result is $(F)_{16}$, then the result of adding 6 is $(15)_{16}$ or $(15)_{10}$ of BCD.

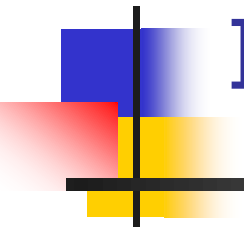
Pipelined BCD Adder

```
module bcd_pipe_addr ( CLK, A, B, S, C );
  input        CLK;
  input  [3:0] A, B;
  output [3:0] S;
  output       C; // Carry
  reg    [4:0] tmp1, tmp2;

  always @( posedge CLK )
  begin
    tmp1 <= { 1'b0, A } + { 1'b0, B };
    if( tmp1 >= 5'b01010 )
        tmp2 <= tmp1 + 5'b00110;
    else
        tmp2 <= tmp1;
  end

  assign S = tmp2[3:0];
  assign C = tmp2[4];
endmodule
```

Pipelined BCD adder is described by blocking assignment.



Block diagram



Introduction to Verilog HDL

Tristate Buffer

and

Bidirectional Input/Output

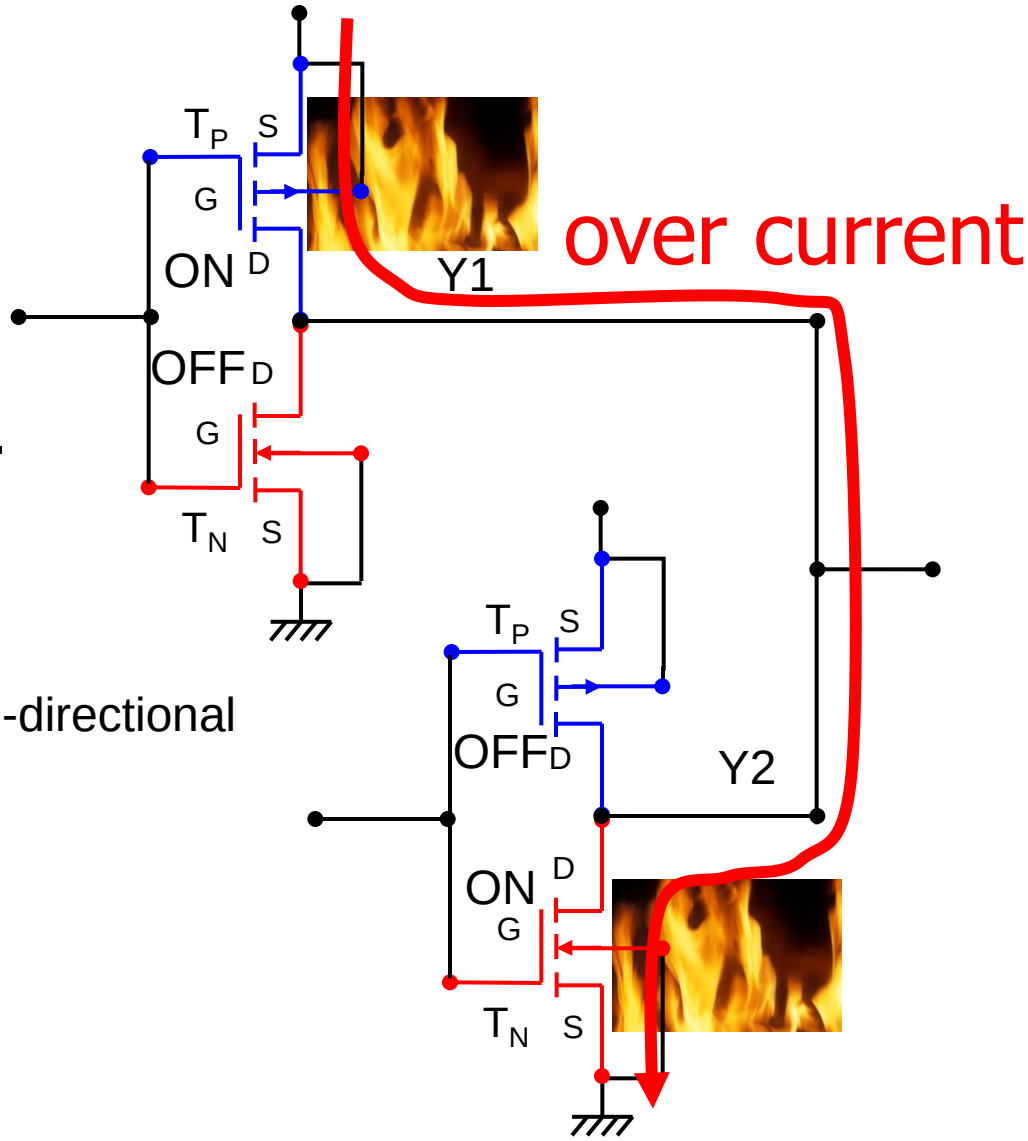
Why Need Tristate Buffer?

Should not connect typical output pins.

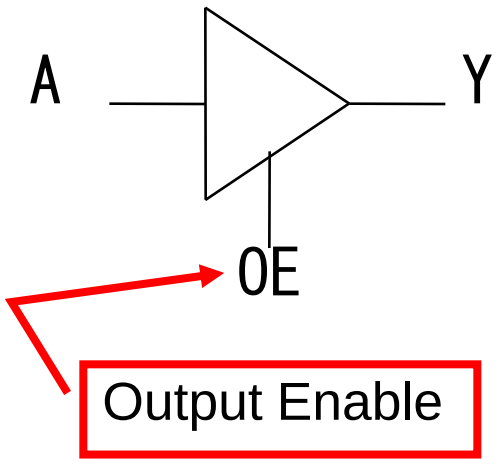
Right figure shows short circuit. Transistors are destroyed because of over current.

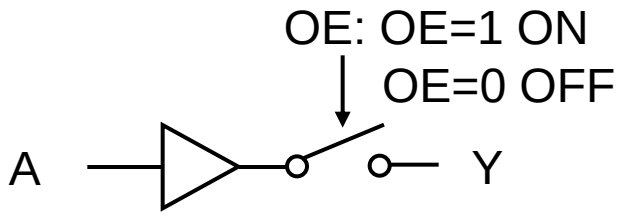
How do we describe bi-directional line or bus line?

Tristate buffer



Tristate Buffer





Equivalent circuit

機能表

OE	Y
0	Z
1	A

Z: High impedance
(unconnected in electrically)

```
wire  [15:0] A, Y;
......
assign Y = ( OE == 1 ) ? A : 16'hZZZZ;
```

always statement can be described at complex output condition.

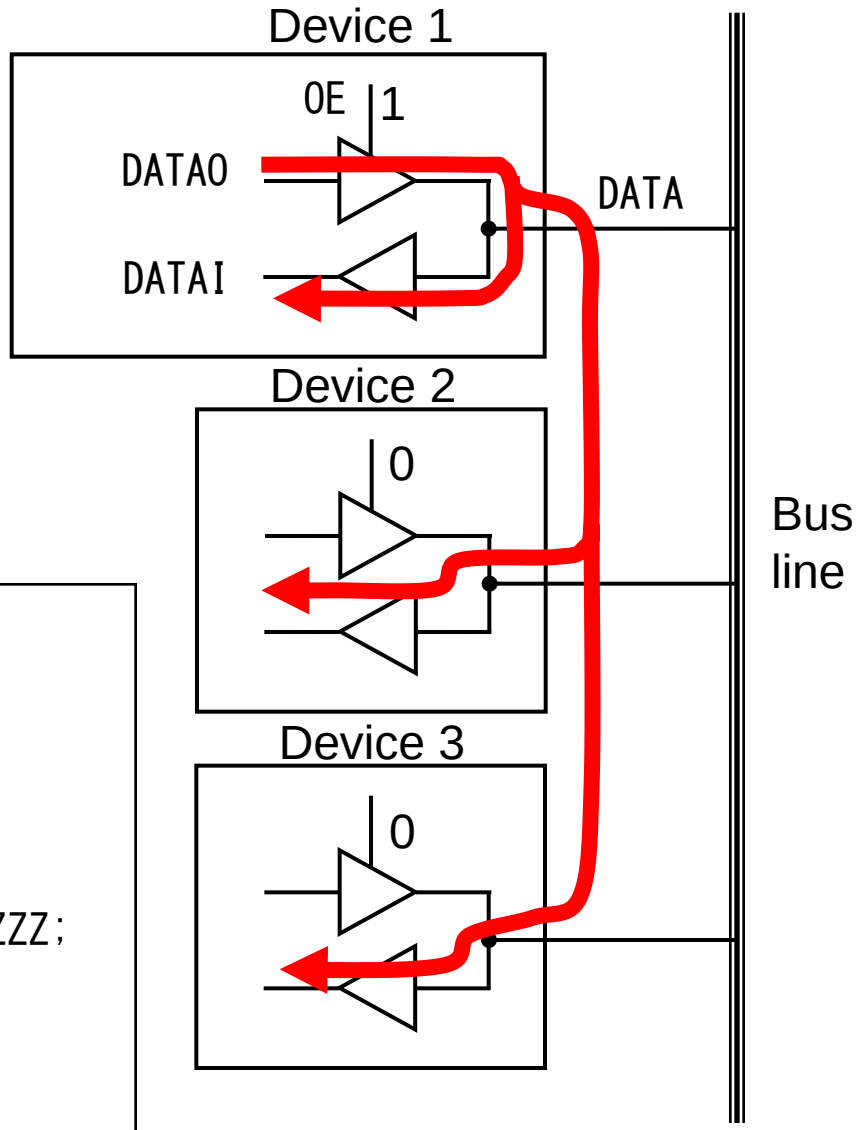
```
reg [15:0] Y;
always @( tmp or OEH or OEL )
begin
   if ( OEH == 1 )      Y <= { tmp[15:8],8'hZZ };
   else if ( OEL == 1 ) Y <= { 8'hZZ, tmp[7:0] };
   else                 Y <= 16'hZZZZ;
end
```

How to use the Tristate Buffer

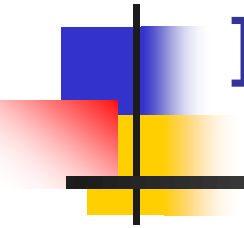
Tristate Buffer utilized for bidirectional buffer.

The one time, only one device can output a value.

All devices can be read the value on the bus line.



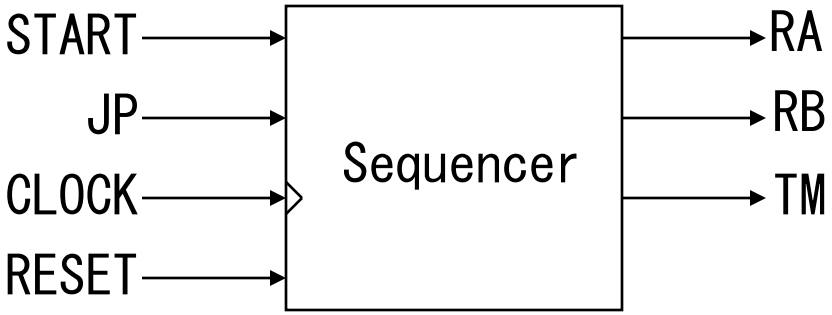

```
module dev1 ( DATA );
  inout  [15:0] DATA;

  wire [15:0] DATAI, DATAO;
  wire        OE;

  assign DATA = ( OE ) ? DATAO : 16'hZZZZ;
    ...
  assign DATAI = DATA;
    ...
endmodule
```

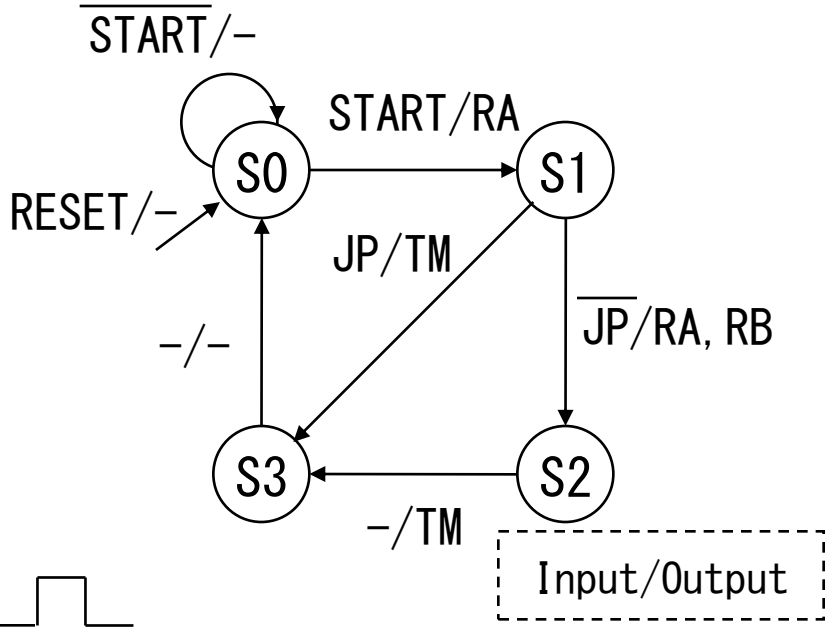


Introduction to Verilog HDL

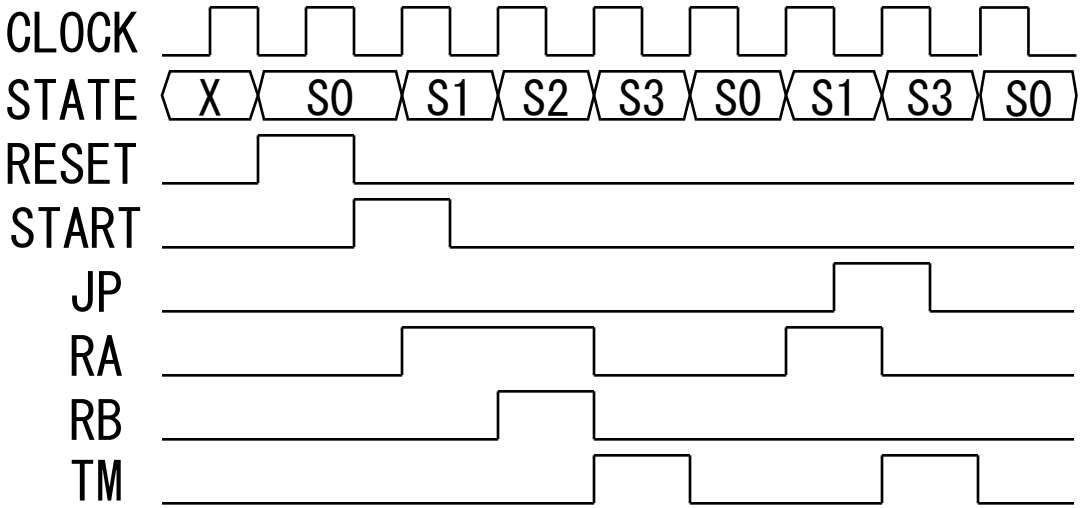
Sequential logic

Sequential Logic





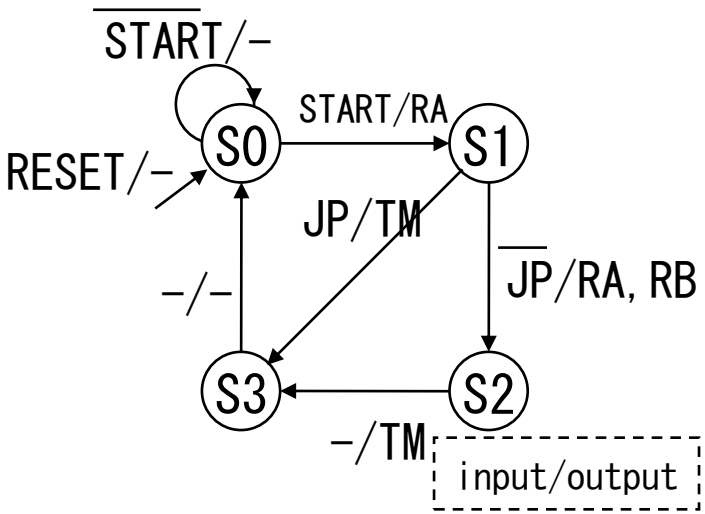
State transition diagram



Wave form

Sequencer (State machine)

```
`define S0  2'b00 // define states
`define S1  2'b01
`define S2  2'b10
`define S3  2'b11

module sequencer ( START, JP, RESET, CLK, RA, RB, TM );
  input  START, JP, RESET, CLK;
  output RA, RB, TM;

  reg    RA, RB, TM;
  reg  [1:0] STATE;   // state variable

  // state machine
  always @( posedge RESET or posedge CLK )
  begin
    if( RESET == 1 ) // asynchronous reset
      STATE <= `S0;
    else
      case ( STATE )
        `S0 : if( START == 1 ) STATE <= `S1; // transition start
        `S1 : if( JP    == 1 ) STATE <= `S3; // jump
              else             STATE <= `S2;
        `S2 :                  STATE <= `S3;
        `S3 :                  STATE <= `S0; // become initial state
      endcase
  end
```



Sequencer (Generate control signals)

Case 1

```
  // control signals
  always @( STATE )
  begin
    case ( STATE )
      `S0 :    begin RA <= 0;  RB <= 0;  TM <= 0; end
      `S1 :    begin RA <= 1;  RB <= 0;  TM <= 0; end
      `S2 :    begin RA <= 1;  RB <= 1;  TM <= 0; end
      `S3 :    begin RA <= 0;  RB <= 0;  TM <= 1; end
      default :begin RA <= 0;  RB <= 0;  TM <= 0; end
    endcase
  end
endmodule
```

Should assign a value to all variables in all cases,
or logic synthesizer makes unexpected transparent latches.

Case 2

```
  // control signals
  always @( STATE )
  begin
    RA <= 0;
    RB <= 0;
    TM <= 0;
    case ( STATE )
      `S1 : begin RA <= 1;                    end
      `S2 : begin RA <= 1; RB <= 1;           end
      `S3 : begin                  TM <= 1; end
    endcase
  end

endmodule
```

default value

assign '1' if necessary

Operators in Verilog HDL

Binary arithmetic
- ·Addition +
- ·Subtraction −
- ·Multiply *
- ·Divide /
- ·Modulo %

Unary arithmetic
- ·Plus +
- ·Minus −

Relational
- ·Equal ==
- ·Not equal !=
- ·Less <
- ·Greater >
- ·Less than <=
- ·Greater than >=
- ·Not !
- ·And &&
- ·Or ||

Binary logical
- ·And &
- ·Or |
- ·Exclusive Or ^

Unary logical
- ·Complement ~

Reduction
- ·And &
- ·Nand ~&
- ·Or |
- ·Nor ~|
- ·Exclusive Or ^
- ·Exclusive Nor ~^

Bit operation
- ·Concatenation { ... , ... }