Chapter 4 Dataflow Modeling

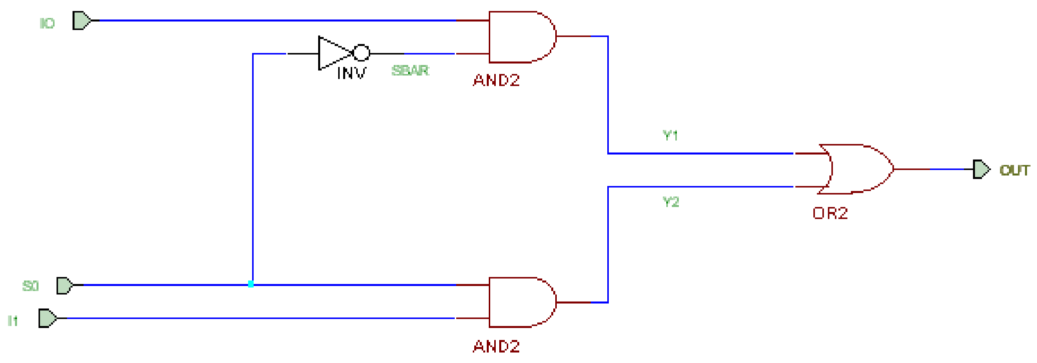
Dataflow modeling provides a powerful way to implement a design. Verilog allows a design processes data rather than instantiation of individual gates. Dataflow modeling has become a popular design approach as ***logic synthesis tools*** have become sophisticated. This approach allows the designer to concentrate on optimizing the circuit in terms of data flow.

4.1 Continuous Assignments

and (Y1, I0, SBAR);　　　(實體化描述)

Y1 is continuous assigned by AND gate.

assign Y1 = I0 & SBAR;　　(布林式描述)



4.1 Continuous Assignments

A **continuous assignment** is the most basic statement in dateflow modeling, used to drive a value onto a net, A continuous assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction. A continuous assignment statement starts with the keyword **assign**.

```
//Syntax of assign statement in the simplest form
< continuous_assign >
  : : = assign < drive_strength > ? < delay > ? < list_of_assignments > ;
```

Continuous assign. out is a net. ***i1*** and ***i2*** are nets .

Concatenation . Left-hand side is a concatenation of a scalar net and a vector net .

Example of Continuous Assignment

```
assign   out = i1 & i2 ;

assign   { c_out , sum[ 3 : 0 ] } = a [ 3 : 0 ] + b [ 3 : 0 ]
         + c_in ;
```

4.1.1 Implicit Continuous Assignment

Instead of declaring a net and then writing a continuous assignment on the net. Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared. There can be only one implicit declaration assignment per net because a net is declared only once.

Regular continuous assignment

Same effect is achieved by an implicit continuous assignment

Example

```
wire out ;
assign   out = in1 & in2 ;

wire     out = in1 & in2 ;
```

4.2 Delays

Delay value control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand-side.

4.2.1 Regular Assignment Delay

The first method is to assign a delay value in a continuous assignment statement. The delay value is specified after the keyword **assign**.

Ex1. Regular Assignment Delay program

1.This example shows inertial delay. The waveform is generated by the assign statement. It shows the delay on signal out. Note the following changes. When signals ***in1*** and ***in2*** go high at time 20, out goes to a high 10 time units later (time = 30).

2.When ***in1*** goes to low at 60, ***out*** changes to low at 70.
3.However, ***in1*** changes to high at 100, but it goes down to low before 10 time units have elapsed.
4.Hence, at the time of recompilation, 10 units after time 100, ***in1*** is 0. Thus, out gets the value 0. A pulse of width less than the specified assignment delay is not propagated to the output.

Verilog Code

Declaration I/O port

Delay in a continuous

```
module regular_delay (out, in1, in2);

output out;
input in1, in2;

assign #10 out = in1 & in2;
endmodule
```

Test Stimulus Code

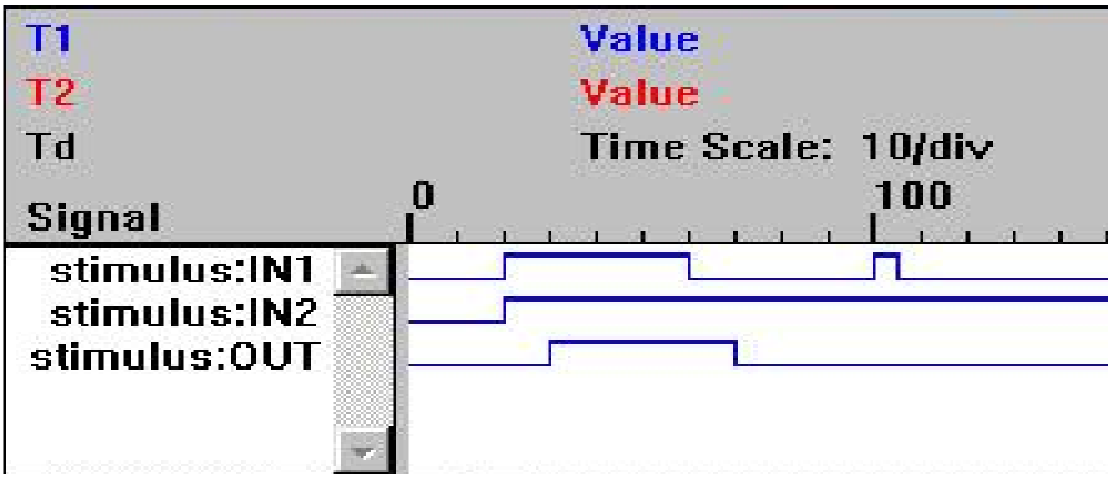
Input test value

Shown the result

Call regular_delay module

```
module stimulus;
wire OUT;
reg IN1, IN2;
initial
begin
      IN1 = 0; IN2= 0;
      #20 IN1=1; IN2= 1;
      #40 IN1 = 0;
      #40 IN1 = 1;
      #5 IN1 = 0;
      #150 $stop;
end

initial
      $monitor("out", OUT, "in1", IN1, "in2",
IN2);

regular_delay rd1(OUT, IN1, IN2);

endmodule
```

Simulation Waveform



4.2.2 Implicit Continuous Assignment Delay

An equivalent method is using an implicit continuous assignment to specify both a delay and an assignment on the net.

Ex2. Implicit Continuous Assignment

Verilog Code

Delay in a continuous assign

```
module implicit_delay (out, in1, in2);

output out;
input in1, in2;

wire #10 out = in1 & in2;

endmodule
```

Test Stimulus Code

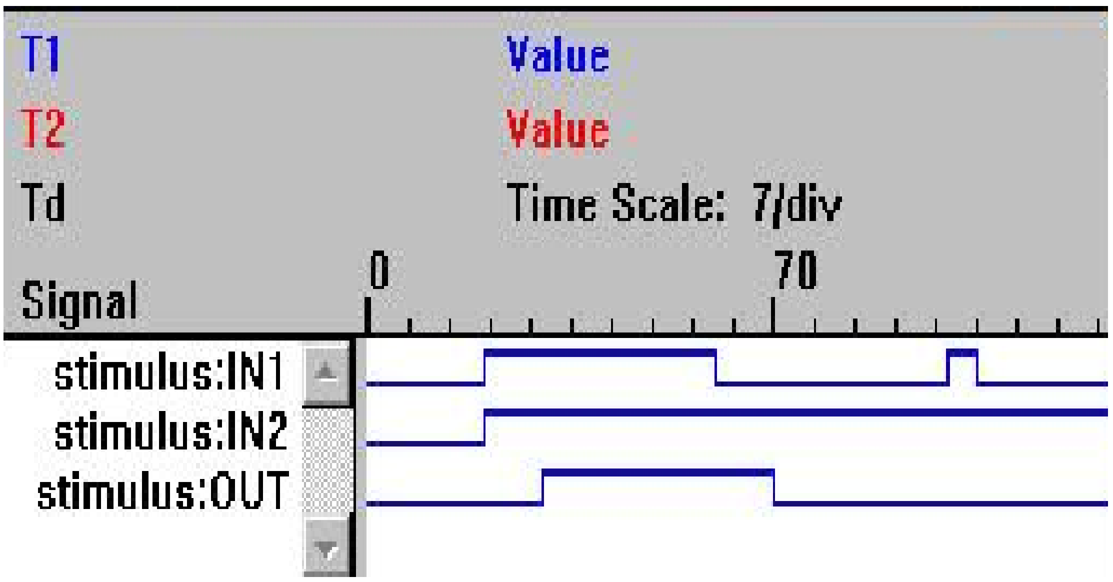
Input test value

Shown the result

Call implicit_delay module

```
module stimulus;

wire OUT;
reg IN1, IN2;

initial
begin
      IN1 = 0; IN2= 0;
      #20 IN1=1; IN2= 1;
      #40 IN1 = 0;
      #40 IN1 = 1;
      #5 IN1 = 0;
      #150 $stop;
end

initial
      $monitor("out", OUT, "in1", IN1, "in2",
IN2);

implicit_delay rd1(OUT, IN1, IN2);

endmodule
```

Simulation Waveform



4.2.3 Net Declaration Delay

A delay can be specified on a net when it is declared without putting a continuous assignment on the net. If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly. Net declaration delays can also be used in gate-level modeling.

Ex3. Net Declaration Delay program

Verilog Code

No delay in a continuous assignment

```
module net_declare (out, in1, in2);

output out;
input in1, in2;

assign out = in1 & in2;

endmodule
```

Test Stimulus Code

Declare the delay on the net.

Input test value

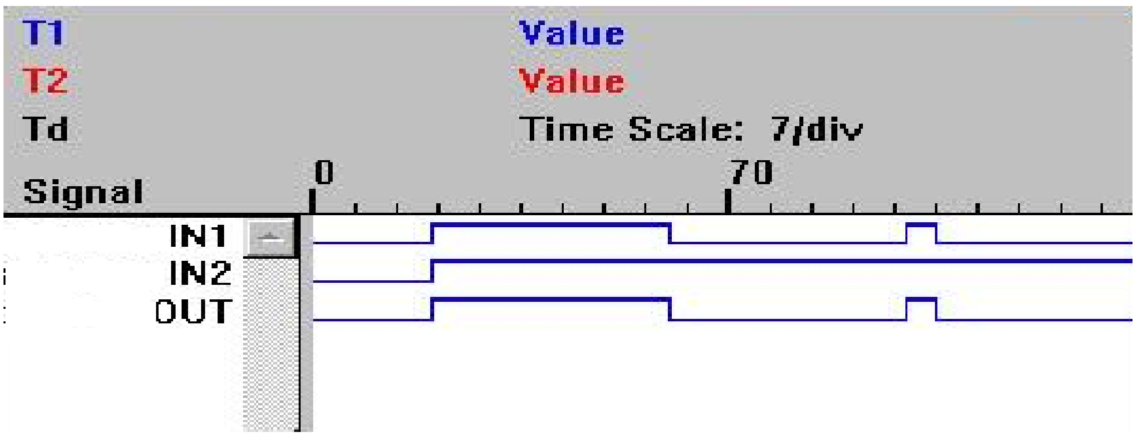
```
module stimulus;

wire    OUT;
reg IN1, IN2;

initial
begin
      IN1 = 0; IN2= 0;
      #20 IN1=1; IN2= 1;
      #40 IN1 = 0;
      #40 IN1 = 1;
```

```
        #5 IN1 = 0;
        #150 $stop;
end

initial
        $monitor("out", OUT, "in1", IN1, "in2",
IN2);

net_declare rd1(OUT, IN1, IN2);

endmodule
```

Show the result → `$monitor`

Call net_declare module → `net_declare rd1`

Simulation Waveform



4.3 Expressions, Operators, and Operands

Dataflow modeling describes the design in terms of expressions instead of primitive gates. **expressions, operators,** and **operands** form the basis of dataflow modeling .

4.3.1 Expressions

Expressions are constructs that combine operators and operands to produce a result.

```
//Examples of expressions . Combine operators and operands
a ^ b
addr1[20:17] + addr2[20:17]
in1 | in2
```

4.3.2 Operands

Operands can be any one of the data types. Some constructs will take only certain types of operands.

count is an integer operand

a and b are real operands

```
integer   count , final_count ;
final_count = count + 1 ;

reg  [127:0] a , b , c ;
c = a – b ;
```

4.3.3 Operators

The operator acts on the operands to produce desired results. Verilog provides various types of operators.

Operator Type	***Operator Symbol***	***Operation Performed***	***Number of Operands***
Arithmetic	*	Multiply	Two
	/	Divide	Two
	+	Add	Two
	-	Subtract	Two
	%	Modulus	Two
Logical	!	Logical negation	One
	&&	Logical and	Two
	| |	Logical or	Two
Relational	>	Greater than	Two
	<	Less than	Two
	>=	Greater than or equal	Two
	<=	Less than or equal	Two
Equality	= =	Equality	Two
	!=	Inequality	Two

	= = =	Case equality	Two
	!= =	Case inequality	Two
Bitwise	~	Bitwise negation	One
	&	Bitwise and	Two
	|	Betwise or	Two
	^	Bitwise xor	Two
	^~ or ~^	Bitwise xnor	Two
Reduction	&	Reduction and	One
	~&	Reduction nand	One
	|	Reduction or	One
	~|	Reduction nor	One
	^	Reduction xor	One
	^~ or ~^	Reduction nxor	One
Shift	>>	Right shift	Two
	<<	Left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	? :	Conditional	three

4.4 Examples

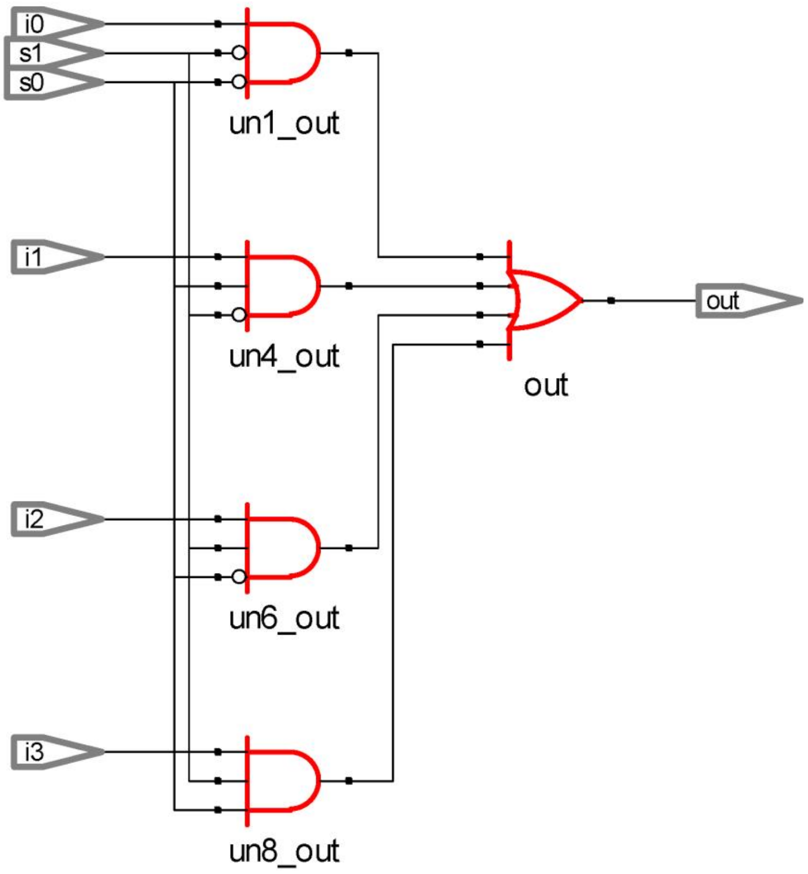
- **Dataflow 4-to-1 Multiplexer (Using Logic Equations)**

Verilog Code

Port declarations from the I/O diagram

Set assignment function

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

output out;
input i0, i1, i2, i3;
input s1, s0;

assign out = (~s1 & ~s0 & i0) |
             (~s1 & s0 & i1) |
             (s1 & ~s0 & i2) |
             (s1 & s0 & i3) ;

endmodule
```


i0
s1
s0
un1_out
i1
un4_out
out
out
i2
un6_out
i3
un8_out

Test Stimulus Code

Define the stimulus module (no ports)

```
module stimulus;
```

Declare variables to be connected to inputs

```
reg IN0, IN1, IN2, IN3;
reg S1, S0;
```

Declare output wire

```
wire OUTPUT;
```

Instantiated the multiplexer

```
mux4_to_1 mymux(OUTPUT, IN0, IN1,
IN2, IN3, S1, S0);
```

Stimulate the inputs

```
initial
begin
```

set input lines

```
    IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
    #1 $display("IN0= %b, IN1= %b,
IN2= %b, IN3= %b\n",IN0,IN1,IN2,IN3);
```

choose IN0

```
    S1 = 0; S0 = 0;
    #1 $display("S1 = %b, S0 = %b,
OUTPUT = %b \n", S1, S0, OUTPUT);
```

choose IN1

```
    S1 = 0; S0 = 1;
    #1 $display("S1 = %b, S0 = %b,
OUTPUT = %b \n", S1, S0, OUTPUT);
```

choose IN2

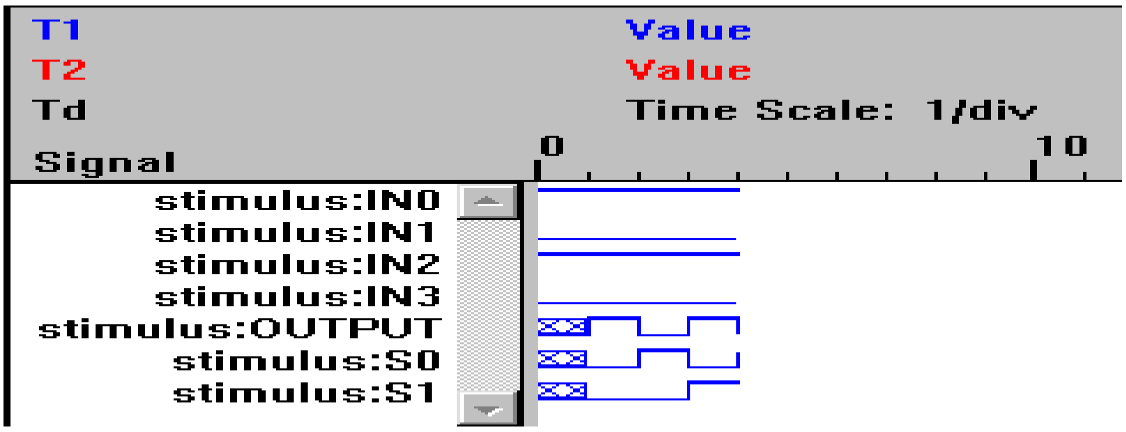
```
    S1 = 1; S0 = 0;
    #1 $display("S1 = %b, S0 = %b,
OUTPUT = %b \n", S1, S0, OUTPUT);
```

choose IN3

```
    S1 = 1; S0 = 1;
    #1 $display("S1 = %b, S0 = %b,
OUTPUT = %b \n", S1, S0, OUTPUT);
```

```
end

endmodule
```

Simulation Waveform



Simulation Result

```
     Simulation stopped at the end of time 0.
Ready: sim
IN0= 1, IN1= 0, IN2= 1, IN3= 0

S1 = 0, S0 = 0, OUTPUT = 1

S1 = 0, S0 = 1, OUTPUT = 0

S1 = 1, S0 = 0, OUTPUT = 1

S1 = 1, S0 = 1, OUTPUT = 0


     16 State changes on observable nets.

     Simulation stopped at the end of time 4.
Ready:
```

- **Dataflow 4-to-1 Multiplexer (Using Conditional Operators)**

Verilog Code

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
```

Port declarations from the I/O diagram

```
output out;
input i0, i1, i2, i3;
input s1, s0;
```

Set mux function

```
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;

endmodule
```

Test Stimulus Code

Define the stimulus module (no ports)

```
module stimulus;
```

Declare variables to be connected to inputs

```
reg IN0, IN1, IN2, IN3;
reg S1, S0;
```

Declare output wire

```
wire OUTPUT;
```

Instantiated the multiplexer

```
mux4_to_1 mymux(OUTPUT, IN0, IN1, IN2,
IN3, S1, S0);
```

Stimulate the inputs

```
initial
begin
```

set input lines

```
      IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
      #1 $display("IN0= %b, IN1= %b, IN2=
%b, IN3= %b\n",IN0,IN1,IN2,IN3);
```

choose IN0

```
      S1 = 0; S0 = 0;
      #1 $display("S1 = %b, S0 = %b, OUTPUT = %b
\n", S1, S0, OUTPUT);
```

choose IN1

```
      S1 = 0; S0 = 1;
      #1 $display("S1 = %b, S0 = %b, OUTPUT = %b
\n", S1, S0, OUTPUT);
```

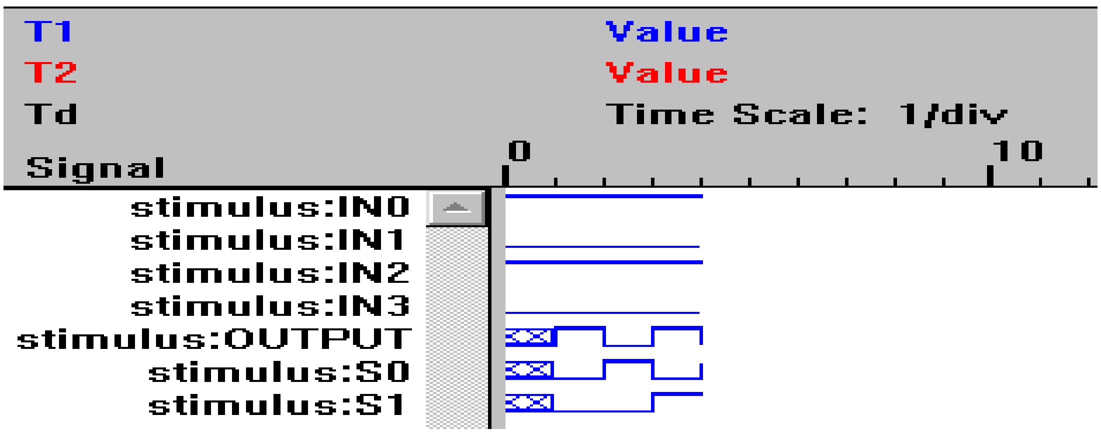
choose IN2

```
        S1 = 1; S0 = 0;
        #1 $display("S1 = %b, S0 = %b,
OUTPUT = %b \n", S1, S0, OUTPUT);
```

choose IN3

```
        S1 = 1; S0 = 1;
        #1 $display("S1 = %b, S0 = %b,
OUTPUT = %b \n", S1, S0, OUTPUT);
end

endmodule
```

Simulation Waveform



Simulation Result

```
      Simulation stopped at the end of time 0.
Ready: sim
IN0= 1, IN1= 0, IN2= 1, IN3= 0

S1 = 0, S0 = 0, OUTPUT = 1

S1 = 0, S0 = 1, OUTPUT = 0

S1 = 1, S0 = 0, OUTPUT = 1

S1 = 1, S0 = 1, OUTPUT = 0


      16 State changes on observable nets.

      Simulation stopped at the end of time 4.
Ready:
```

- **Dataflow 4-bit Full Adder (Using Dataflow Operators)**

Verilog Code

Define a 4-bit full adder

I/O port declarations

Specify the function of a full adder

```
module fulladd4(sum, c_out, a, b, c_in);

output [3:0] sum;
output c_out;
input[3:0] a, b;
input c_in;

assign {c_out, sum} = a + b + c_in;

endmodule
```

Test Stimulus Code

Define the stimulus (top level module)

Set up variables

Instantiated the 4-bit full adder. call it FA1 4

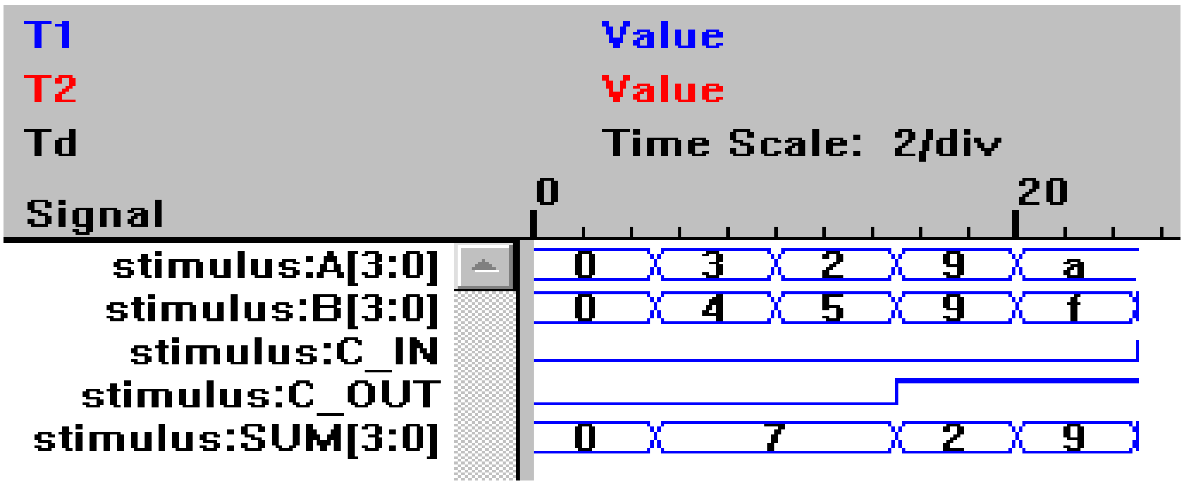
Setup the monitoring for the signal values

Stimulate inputs

```
module stimulus;


reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

fulladd4 FA1_4(SUM, C_OUT, A, B,
C_IN);

initial
begin
     $monitor($time," A= %b, B=%b,
C_IN= %b,, C_OUT= %b, SUM=
%b\n",   A, B, C_IN, C_OUT, SUM);
end

initial
begin
     A = 4'd0; B = 4'd0; C_IN = 1'b0;
     #5 A = 4'd3; B = 4'd4;
     #5 A = 4'd2; B = 4'd5;
```

```
    #5 A = 4'd2; B = 4'd5;
    #5 A = 4'd9; B = 4'd9;
    #5 A = 4'd10; B = 4'd15;
    #5 A = 4'd10; B = 4'd5; C_IN = 1'b1;
end
endmodule
```

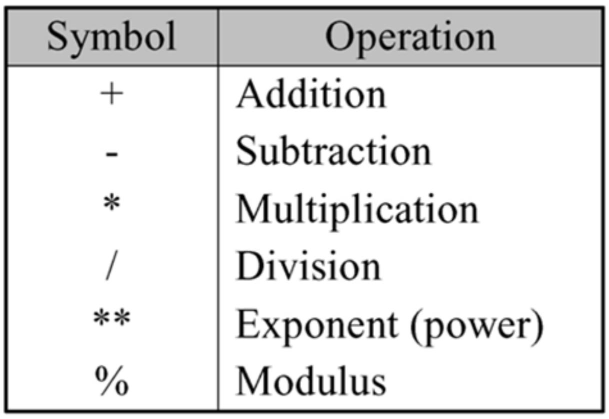
Simulation Waveform



Simulation Result

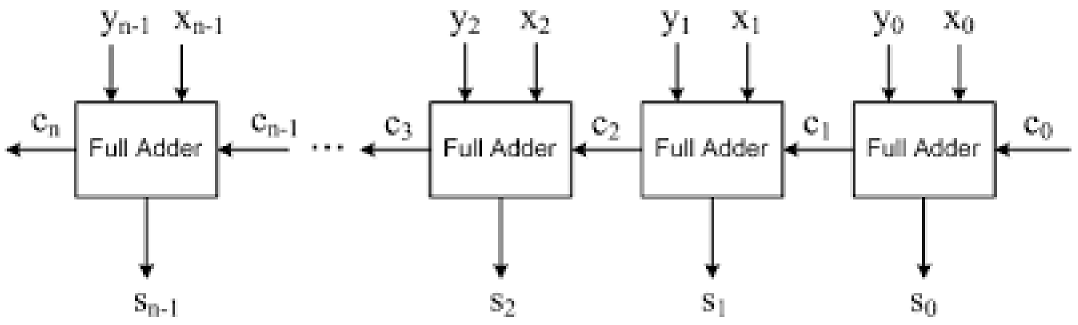
```
    Simulation stopped at the end of time 0.
Ready: sim
       5 A= 0011, B=0100, C_IN= 0,, C_OUT= 0, SUM= 0111

      10 A= 0010, B=0101, C_IN= 0,, C_OUT= 0, SUM= 0111

      15 A= 1001, B=1001, C_IN= 0,, C_OUT= 1, SUM= 0010

      20 A= 1010, B=1111, C_IN= 0,, C_OUT= 1, SUM= 1001

    | 25 A= 1010, B=0101, C_IN= 1,, C_OUT= 1, SUM= 0000


    45 State changes on observable nets.

    Simulation stopped at the end of time 25.
Ready:
```

- The result is x if any operand bit has a value x
- Negative numbers are represented as 2's complement



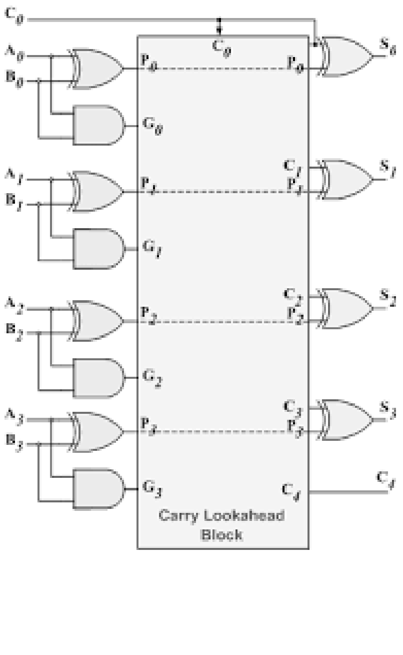
Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponent (power)
%	Modulus

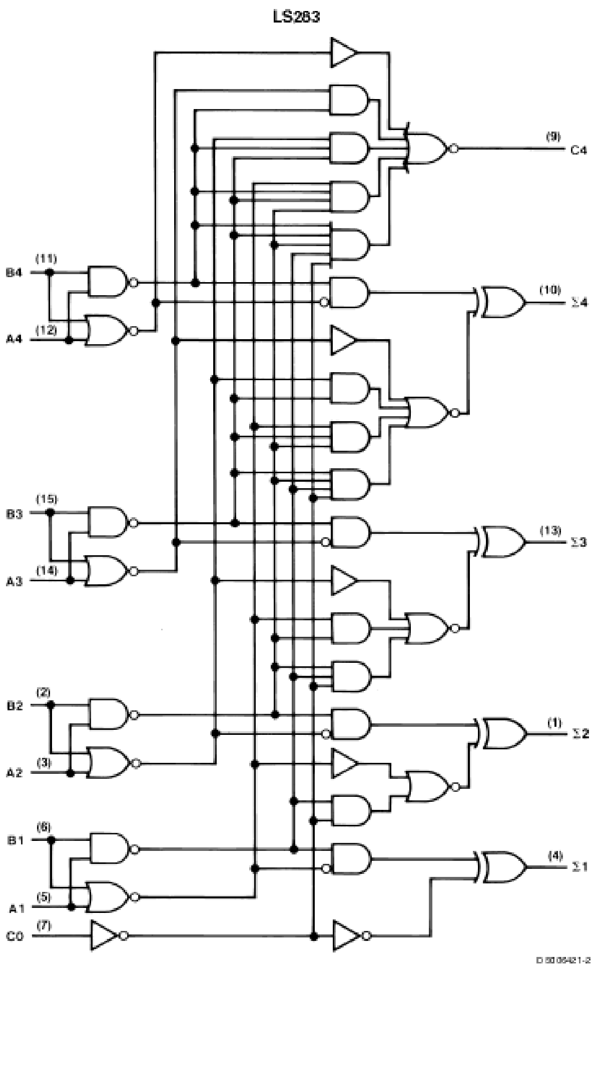
Carry Propagation Adder:



Disadvantage: Long delay for output ***Cn***

Carry Look-Ahead Adder:





- **Dataflow 4-bit Full Adder with Carry Lookahead**

Verilog Code

```
module fulladd4(sum, c_out, a, b, c_in);
```

Inputs and outputs

```
output [3:0] sum;
output c_out;
input [3:0] a,b;
input c_in;
```

Internal wires

```
wire p0,g0, p1,g1, p2,g2, p3,g3;
wire c4, c3, c2, c1;
```

compute the p for each stage

```
assign p0 = a[0] ^ b[0],
             p1 = a[1] ^ b[1],
             p2 = a[2] ^ b[2],
             p3 = a[3] ^ b[3];
```

compute the g for each stage

```
assign g0 = a[0] & b[0],
             g1 = a[1] & b[1],
             g2 = a[2] & b[2],
             g3 = a[3] & b[3];
```

compute the carry for each stage.

```
assign c1 = g0 | (p0 & c_in),
          c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
           c3 = g2 | (p2 & g1) | (p2 & p1 & g0) |
          (p2 & p1 & p0 & c_in),
           c4 = g3 | (p3 & g2) | (p3 & p2 & g1) |
          (p3 & p2 & p1 & g0) | (p3 & p2 & p1 &
          p0 & c_in);
```

Compute Sum

```
assign sum[0] = p0 ^ c_in,
             sum[1] = p1 ^ c1,
             sum[2] = p2 ^ c2,
             sum[3] = p3 ^ c3;
```

Assign carry output

```
assign c_out = c4;

endmodule
```

Test Stimulus Code

Define the stimulus (top level module)

Set up variables

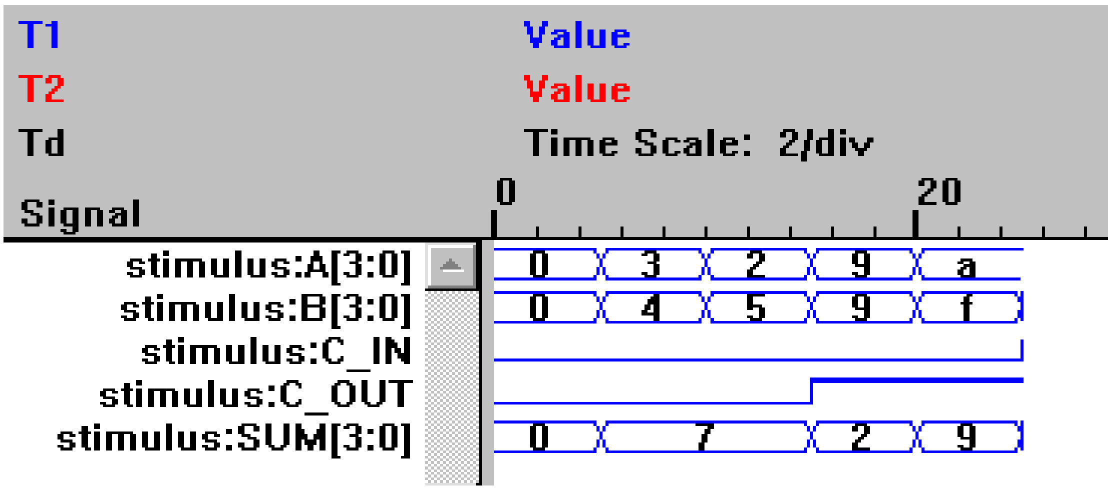
Instantiated the 4-bit full adder. call it FA1_4

Setup the monitoring for the signal values

Stimulate inputs

```
module stimulus;


reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

fulladd4 FA1_4(SUM, C_OUT, A, B, C_IN);

initial
begin
        $monitor($time," A= %b, B=%b,
C_IN= %b,, C_OUT= %b, SUM= %b\n",
        A, B, C_IN, C_OUT, SUM);
end

initial
begin
        A = 4'd0; B = 4'd0; C_IN = 1'b0;
        #5 A = 4'd3; B = 4'd4;
        #5 A = 4'd2; B = 4'd5;
        #5 A = 4'd9; B = 4'd9;
        #5 A = 4'd10; B = 4'd15;
       #5 A = 4'd10; B = 4'd5; C_IN = 1'b1;
end

endmodule
```

Simulation Waveform



Simulation Result

```
    Simulation stopped at the end of time 0.
Ready: sim
      5 A= 0011, B=0100, C_IN= 0,, C_OUT= 0, SUM= 0111

     10 A= 0010, B=0101, C_IN= 0,, C_OUT= 0, SUM= 0111

     15 A= 1001, B=1001, C_IN= 0,, C_OUT= 1, SUM= 0010

     20 A= 1010, B=1111, C_IN= 0,, C_OUT= 1, SUM= 1001

     25 A= 1010, B=0101, C_IN= 1,, C_OUT= 1, SUM= 0000


    71 State changes on observable nets.

    Simulation stopped at the end of time 25.
Ready:
```

- **Dataflow Ripple Counter**

Verilog Code

Edge triggered D flipflop →

Inputs and outputs →

```
module edge_dff(q, qbar, d, clk, clear);

output q,qbar;
input d, clk, clear;
```

Internal variables

```
wire s, sbar, r, rbar,cbar;

// Data flow statements
```

Create a complement of signal clear

```
assign cbar = ~clear;
```

Input latches

```
assign    sbar = ~(rbar & s),
        s = ~(sbar & cbar & ~clk),
        r = ~(rbar & ~clk & s),
        rbar = ~(r & cbar & d);
```

Output latch

```
assign    q = ~(s & qbar),
        qbar = ~(q & r & cbar);

endmodule
---------------------------------------------------
```

Edge triggered T-flipflop. Toggles every clock cycle.

```
module T_ff(q, clk, clear);

output q;
input clk, clear;
```

Instantiated the edge triggered DFF

```
edge_dff ff1(q, ,~q, clk, clear);

endmodule
----------------------------------------------------
```

Ripple counter

```
module counter(Q , clock, clear);
```

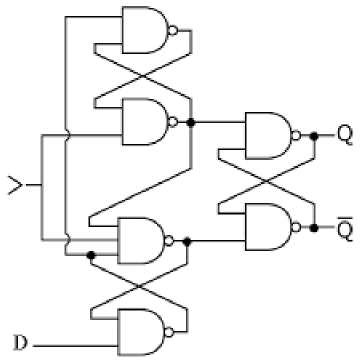
I/O ports

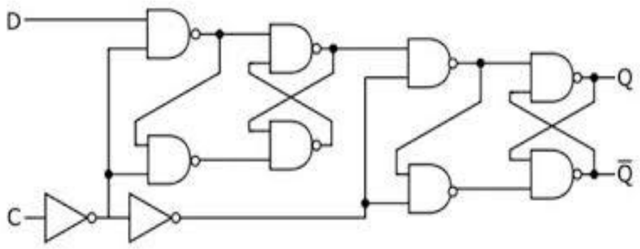
```
output [3:0] Q;
input clock, clear;
```

Instantiated the T flipflops

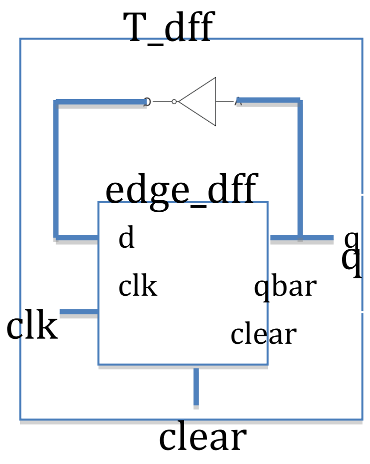
```
T_ff tff0(Q[0], clock, clear);
T_ff tff1(Q[1], Q[0], clear);
T_ff tff2(Q[2], Q[1], clear);
T_ff tff3(Q[3], Q[2], clear);

endmodule
```

The logic diagram of edge trigger Flip Flop

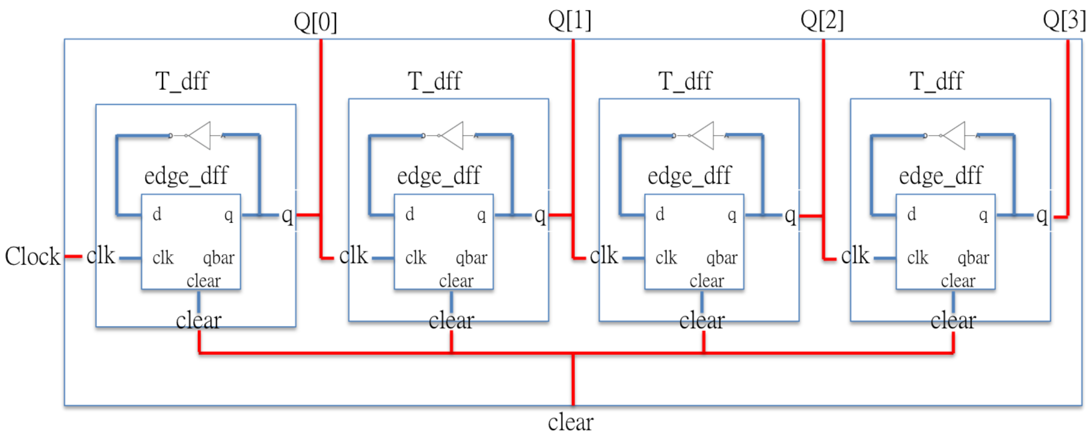




The circuit diagram of T_dff and edge_dff modules:



The circuit diagram of counter:



Test Stimulus Code

Top level stimulus module

Declare variables for stimulating input

Instantiated the design block

Stimulate the Clear Signal

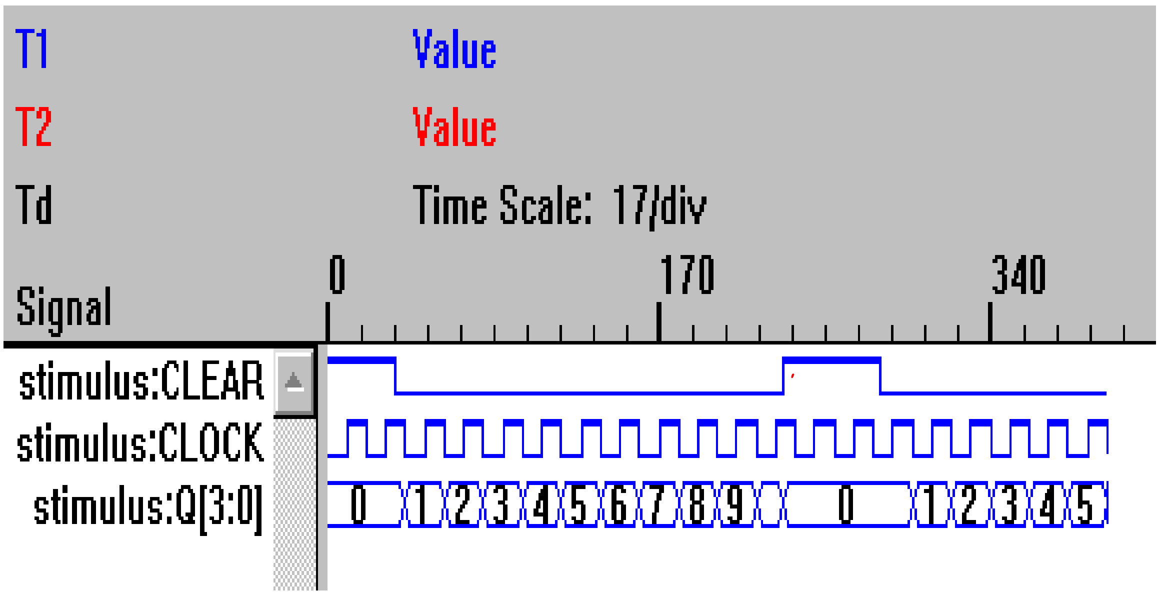
Setup the clock to toggle every 10 time units

Finish the simulation at time 200

```
module stimulus;

reg CLOCK, CLEAR;
wire [3:0] Q;

initial
      $monitor($time, " Count Q = %b Clear=
%b",    Q[3:0],CLEAR);

initial
      $gr_waves(  "clk", CLOCK,
                  "Clear", CLEAR,
                  "Q", Q[3:0],
                  "Q0", Q[0],
                  "Q1", Q[1],
                  "Q2", Q[2],
                  "Q3", Q[3]);

counter c1(Q, CLOCK, CLEAR);

initial
begin
      CLEAR = 1'b1;
      #34 CLEAR = 1'b0;
      #200 CLEAR = 1'b1;
      #50 CLEAR = 1'b0;
end

initial
begin
      CLOCK = 1'b0;
      forever #10 CLOCK = ~CLOCK;
end

initial
begin
```

```
        #400 $finish;
end

endmodule
```

Simulation Waveform



Simulation Result

```
    Simulation stopped at the end of time 0.
Ready: sim
                                  34 Count Q = 0000 Clear= 0
                                  40 Count Q = 0001 Clear= 0
                                  60 Count Q = 0010 Clear= 0
                                  80 Count Q = 0011 Clear= 0
                                 100 Count Q = 0100 Clear= 0
                                 120 Count Q = 0101 Clear= 0
                                 140 Count Q = 0110 Clear= 0
                                 160 Count Q = 0111 Clear= 0
                                 180 Count Q = 1000 Clear= 0
                                 200 Count Q = 1001 Clear= 0
                                 220 Count Q = 1010 Clear= 0
                                 234 Count Q = 0000 Clear= 1
                                 284 Count Q = 0000 Clear= 0
                                 300 Count Q = 0001 Clear= 0
                                 320 Count Q = 0010 Clear= 0
                                 340 Count Q = 0011 Clear= 0
```

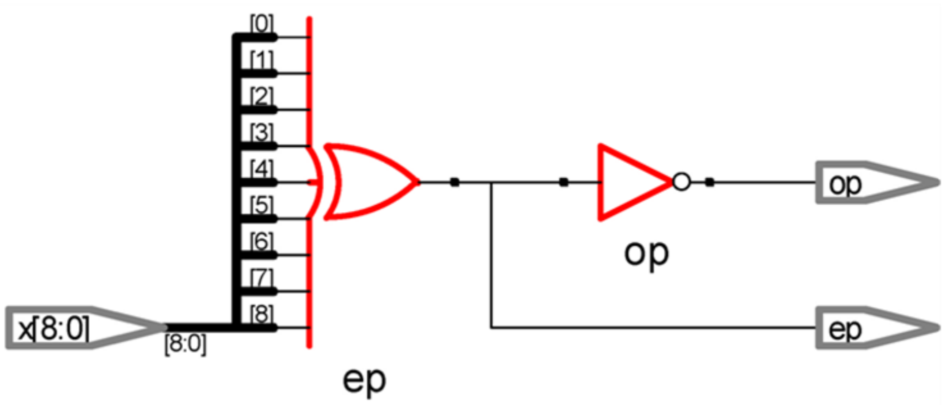
Reduction Operators

- Perform only on one vector operand
  - Carry out a bit-wise operation
  - Yield a 1-bit result
  - Work bit by bit from right to left

Symbol	Operation
&	Reduction and
~&	Reduction nand
|	Reduction or
~|	Reduction nor
^	Reduction exclusive or
~^, ^~	Reduction exclusive nor

A 9-Bit Parity Generator

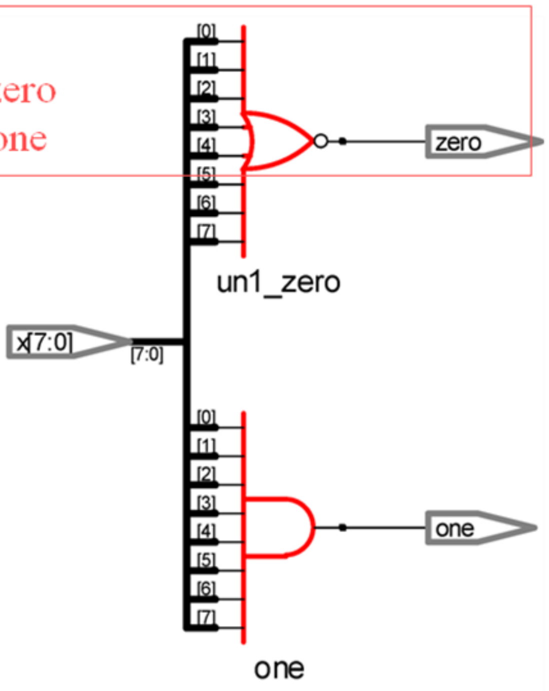
```
// dataflow modeling using reduction operator
assign ep = ^x;     // even parity generator
assign op = ~ep;  // odd parity generator
```

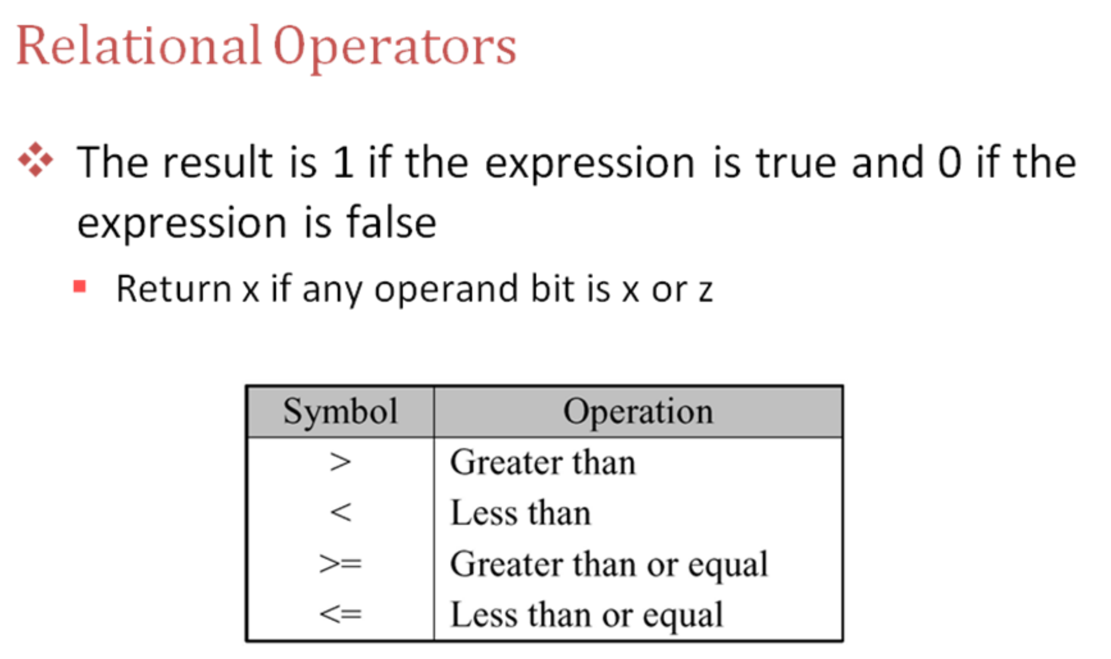


ep=x[0]^x[1] ^x[2] ^x[3] ^x[4] ^x[5] ^x[6] ^x[8] ^x[9];

An All-Bit-Zero/One Detector

```
// dataflow modeling
  assign zero = ~(|x);  // all-bit zero
  assign one = &x;       // all-bit one
```



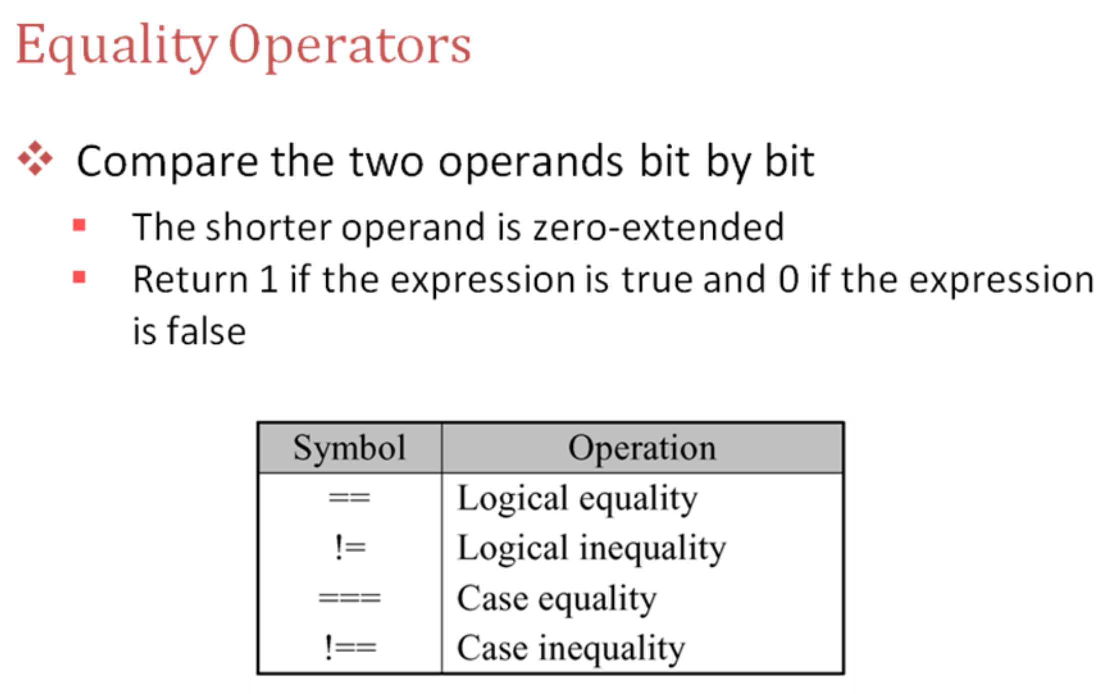


Relational Operators

- The result is 1 if the expression is true and 0 if the expression is false
  - Return x if any operand bit is x or z

Symbol	Operation
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

A = b>c;



Equality Operators

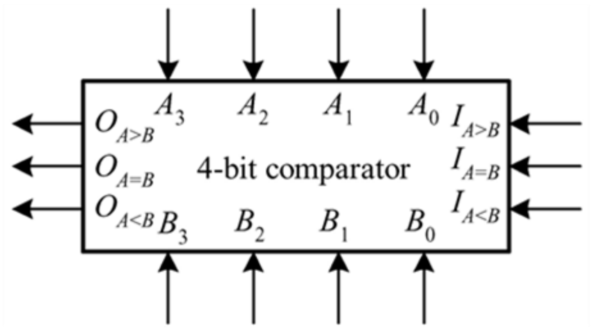
- Compare the two operands bit by bit
  - The shorter operand is zero-extended
  - Return 1 if the expression is true and 0 if the expression is false

Symbol	Operation
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality

Equality Operators

- The operators (==, !=)
  - yield an x if any operand bit is x or z
- The operators (===, !==)
  - yield a 1 if the two operands exactly match
  - 0 if the two operands not exactly match

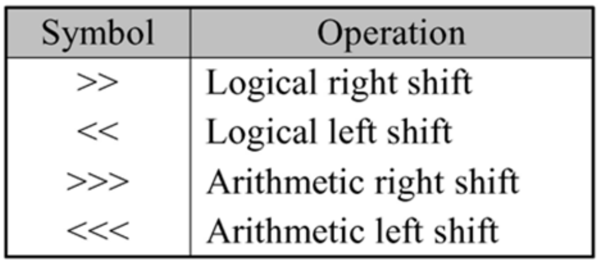
A 4-B Magnitude Comparator



```
// dataflow modeling using relation operators
   assign Oaeqb = (a == b) && (Iaeqb == 1);              //  =
   assign Oagtb  = (a > b) || ((a == b)&& (Iagtb == 1));  // >
   assign Oaltb   = (a < b) || ((a == b)&& (Ialtb == 1));  // <
```

Shift Operators

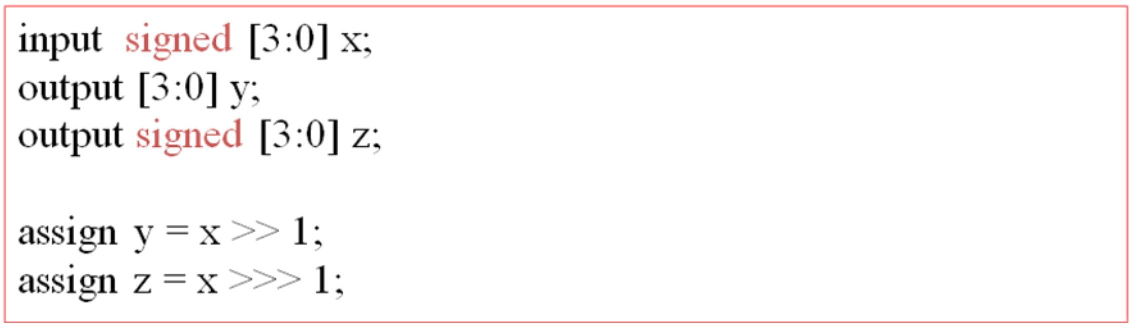
- Logical shift operators
- Arithmetic shift operators



Symbol	Operation
>>	Logical right shift
<<	Logical left shift
>>>	Arithmetic right shift
<<<	Arithmetic left shift

-2= 00010→ 11101+1=11110   01111   11111
10110 >> 11011,   00110>>00011

An Example of Shift Operators



```
input  signed [3:0] x;
output [3:0] y;
output signed [3:0] z;

assign y = x >> 1;
assign z = x >>> 1;
```

x= 0100,   y=x>>1=0010,   z=x>>>1=0010,
x= 1100,   y=x>>1=0110,   z=x>>>1=1110,
x= 1111,   y=x>>1=0111,   z=x>>>1=1111,

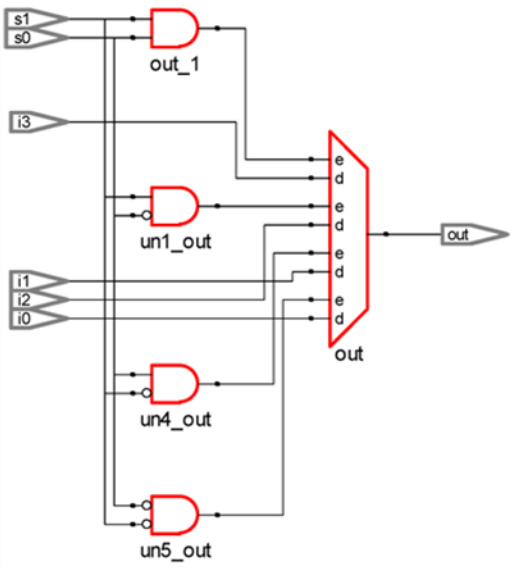
The Conditional Operator

- Usage: condition_expr ? true_expr: false_expr;
  - If condition_expr = x or z: the result = true_expr & false_expr (0 and 0 gets 0, 1 and 1 gets 1, others gets x)
- For example

```
assign out = selection ? in_1: in_0;
```

An Example --- A 4-to-1 MUX

```
// using conditional operator (?:)
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
```



Homework 2:

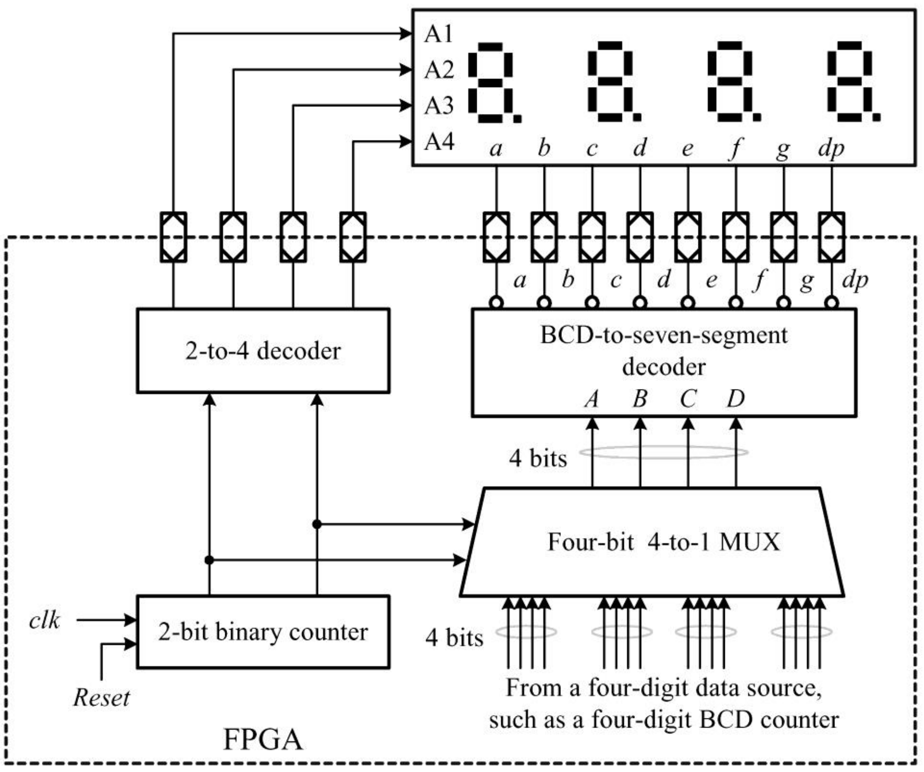


Figure 8.20: The complete logic circuit of a four-digit multiplexing-driven seven-segment LED display.

1. complete a common-cathode seven-segment LED display which integrates the following 4 modules.

(a) Write a 2-bit binary counter module. (**data flow desicription**)

(b) Write a BCD-to-seven-segment decoder module. (**data flow desicription**)

(c) Write a 2-to-4 Decoder module. (**data flow desicription**)

(d) Write a 4-bit 4-to-1 multiplexer module. (**data flow desicription**)

Chap. 6 dataflow Exercise: 1, 2, 3

Chap. 5 gate-level Exercise: 3, 4, 5

Chap. 3 Lexical Exercise: 1, 2, 3, 4, 5

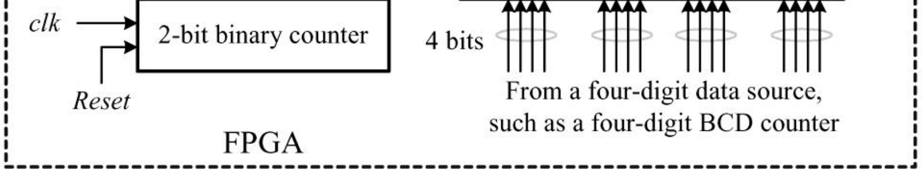
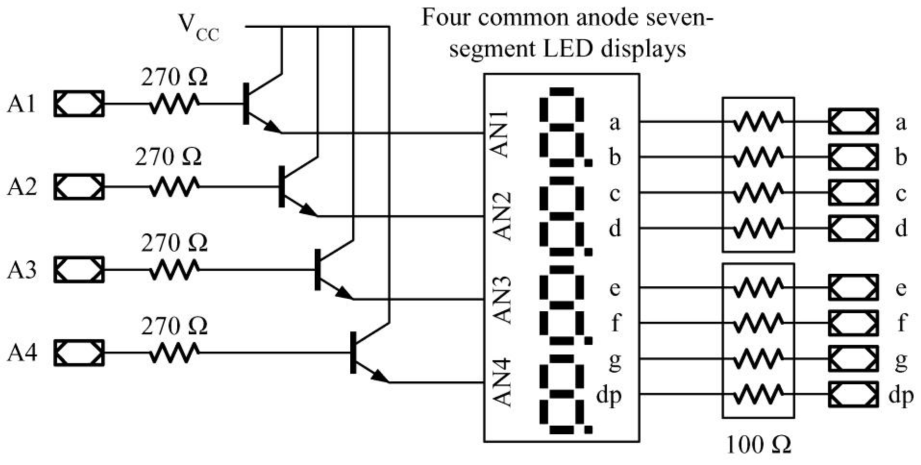
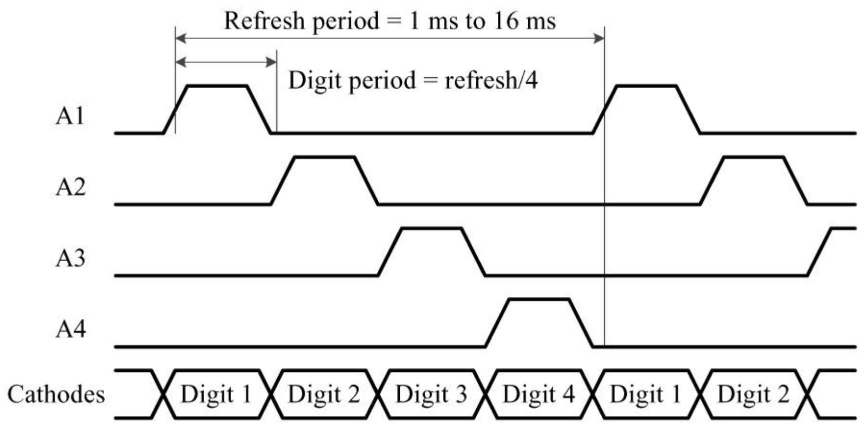

clk
2-bit binary counter
Reset
4 bits
From a four-digit data source, such as a four-digit BCD counter
FPGA

Figure 8.20: The complete logic circuit of a four-digit multiplexing-driven seven-segment LED display.



(a) A four-seven-segment LED display using multiplexing technique



(b) Timing diagram for the four-seven-segment LED display shown in (a)

Figure 8.19: The logic circuit and timing diagram of a four-digit multiplexing-driven seven-segment LED display.

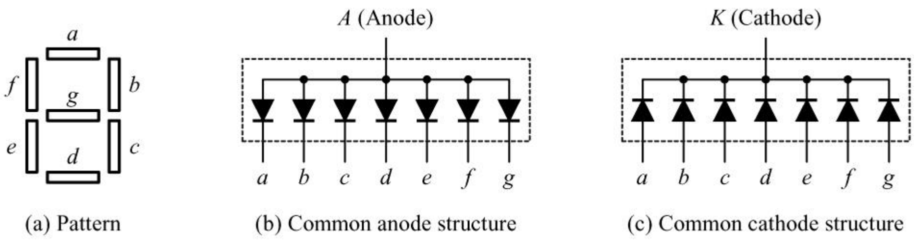
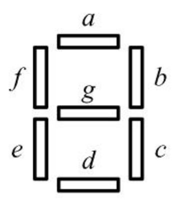
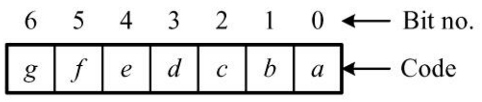


Figure 8.16: The structures of seven-segment LEDs.



(a) Seven-segment LED display

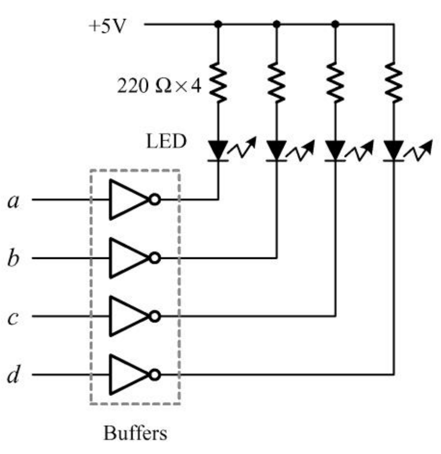


(b) Seven-segment LED display code format

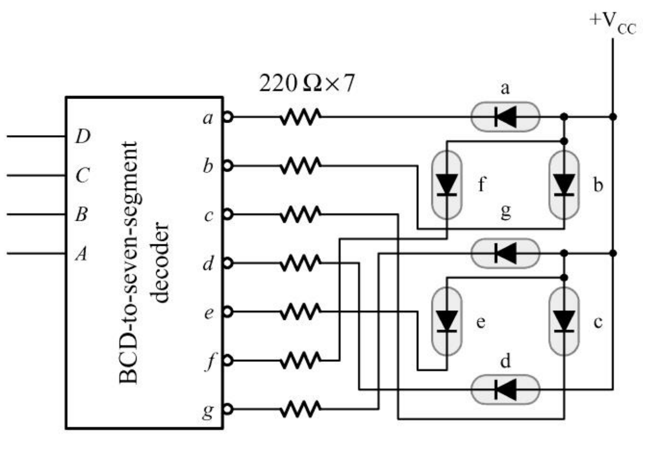
Digit	*g*	*f*	*e*	*d*	*c*	*b*	*a*	Code	Digit	*g*	*f*	*e*	*d*	*c*	*b*	*a*	Code
0	1	0	0	0	0	0	0	40	8	0	0	0	0	0	0	0	00
1	1	1	1	1	0	0	1	79	9	0	0	1	0	0	0	0	10
2	0	1	0	0	1	0	0	24	*A* (*a*)	0	0	0	1	0	0	0	08
3	0	1	1	0	0	0	0	30	*B* (*b*)	0	0	0	0	0	1	1	03
4	0	0	1	1	0	0	1	19	*C* (*c*)	1	0	0	0	1	1	0	46
5	0	0	1	0	0	1	0	12	*D* (*d*)	0	1	0	0	0	0	1	21
6	0	0	0	0	0	1	0	02	*E* (*e*)	0	0	0	0	1	1	0	06
7	1	1	1	1	0	0	0	78	*F* (*f*)	0	0	0	1	1	1	0	0E

(c) Seven-segment LED display code

Figure 8.18: The relationship between digit and common-anode seven-segment LED display code.



(a) LED indicator



(b) Seven-segment LED display

Figure 8.17: Examples of LED indicators and a seven-segment LED display circuit.