100-1 Under-Graduate Project
Logic Design with Behavioral Models

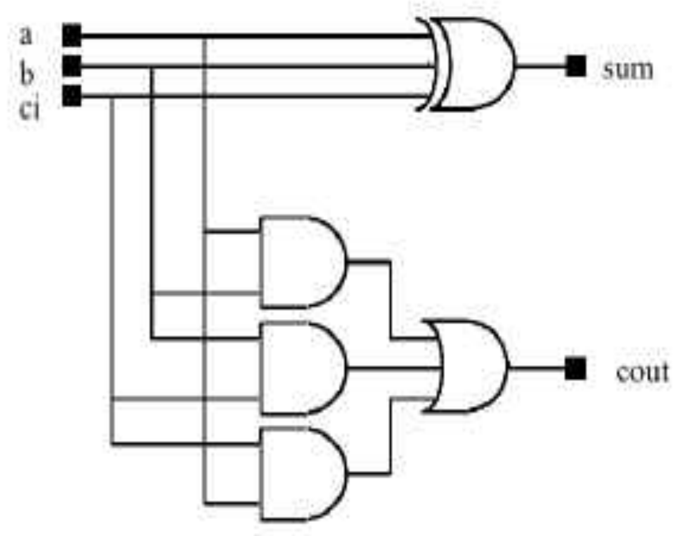
Speaker: Lindaliu
Adviser: Prof. An-Yeu Wu
Date: 2011/10/4

Outline

Various Abstraction of Verilog HDL

❖ Gate level Verilog description of a full-adder

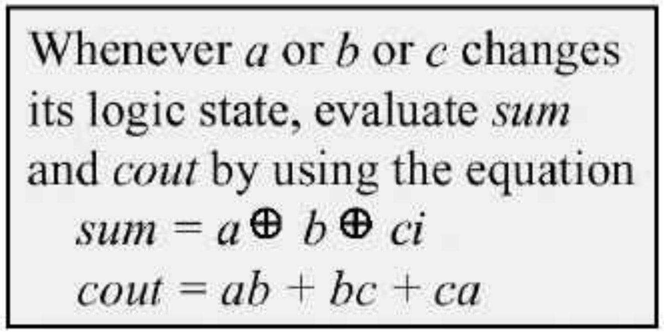
```
module fadder(cout, sum, a, b, cin);
// port declaration
  output  cout, sum;
  input    a, b, cin;
  wire     net1, net2, net3;
// Netlist description
  xor  U0(sum,a,b,cin);
  and U1(net1, a, b);
  and U2(net2, b, cin);
  and U3(net3, cin, a);
  or    U4(cout, net1, net2, net3;
endmodule
```



Various Abstraction of Verilog HDL

- RTL level Verilog description of a full adder

```
module fadder(cout, sum, a, b, cin);
// port declaration
  output  cout, sum;
  input    a, b, cin;
  wire      cout, sum;
// RTL description
  assign  sum = a^b^cin;
  assign  cout = (a&b)|(b&cin)|(cin&a);
endmodule
```



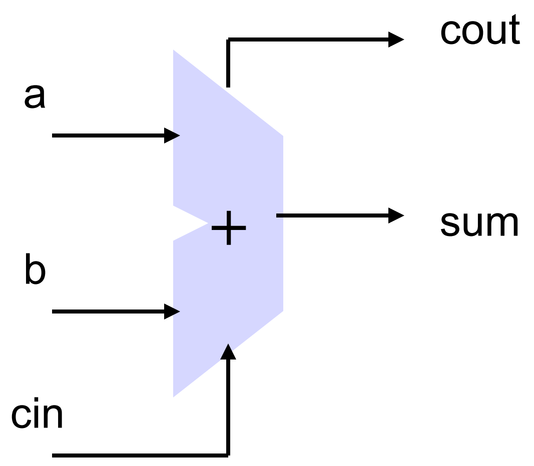
Whenever *a* or *b* or *c* changes its logic state, evaluate *sum* and *cout* by using the equation

$$sum = a \oplus b \oplus ci$$
$$cout = ab + bc + ca$$

Various Abstraction of Verilog HDL

- Behavioral level Verilog description of a full adder

```
module fadder(cout, sum, a, b, cin);
// port declaration
  output  cout, sum;
  input    a, b, cin;
  reg       cout, sum;
// behavior description
  always @(a or b or cin)
    begin
        {cout,sum} = a + b + cin;
    end
endmodule
```



Operators

- Arithmetic Operators

Symbol	Operator
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

- examples:

```
A=4'b0011; B=4'b0100;          // A and B are register vectors
D=6; E=4;                      //D and E are integers
A*B                            // 4'b1100
D/E                            // 1
A+B                            // 4'b0111
in1=4'b101x; in2=4'b1010;
sum=in1 + in2;                 // 4'bx
-10 % 3 = -1                   // Take sign of the first operand
14 % -3 = 2
```

Operators-example (arithmetic)

```
module test;
  reg [3:0] A,B;
  wire [4:0]   sum,diff1,diff2,neg;

  assign sum=A+B;
  assign diff1=A-B;
  assign diff2=B-A;
  assign neg=-A;

  initial
  begin
    #5 A=5;B=2;
    $display("t_sim A B A+B A-B B-A -A");
    $monitor($time,"%d%d%d%d%d%d",A,B,sum,
                       diff1,diff2,neg);
    #10 $monitoroff;
    $monitor($time,"%b%b%b%b%b%b",A,B,sum,
                       diff1,diff2,neg);
endmodule
```

```
t_sim A     B     A+B   A-B   B-A   -A
5     5     2     7     3     29    27
15    0101  0010  00111 00011 11101 11011
```

Operator

- Bit-wise operators

Symbol	Operator
~	Bitwise negation
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
~^, ^~	Bitwise exclusive nor

~(101011)=010100

(010101)&(001100)=000100

(010101)|(001100)=011101

(010101)^(001100)=011001

Operator

- Reduction operators (1-bit result)

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

&(10101010)=1'b0

|(10101010)=1'b1

&(10x0x0x)=1'b0

|(10x01010)=1'b1

Operator

- Logical operators

Symbol	Operator
!	Logical negation
&&	Logical and
||	Logical or
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality

- === : determines whether two words match identically on a bit-by-bit basis, including bits that have values “x” and “z”

Operator

- Shift operator

  >> logical shift right

  << logical shift left

```
module shift_reg(out,in);
  output [5:0] out;
  input [5:0] in;
  parameter shift=3;
  assign out=in<<shift;
endmodule
```

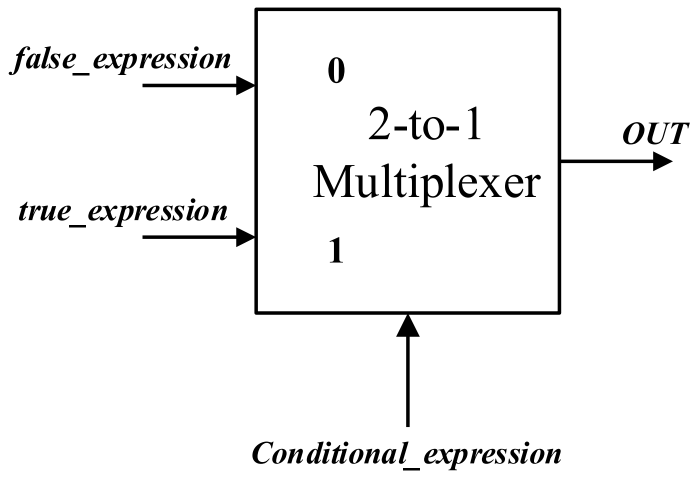
```
reg_in=6'b011100

reg_in<<3   100000
reg_in>>3   000011
```

Operator

- Conditional Operator

  Usage: ***conditional_expression ? true_expression: false_expression;***

- The action of a conditional operator is similar to a multiplexer



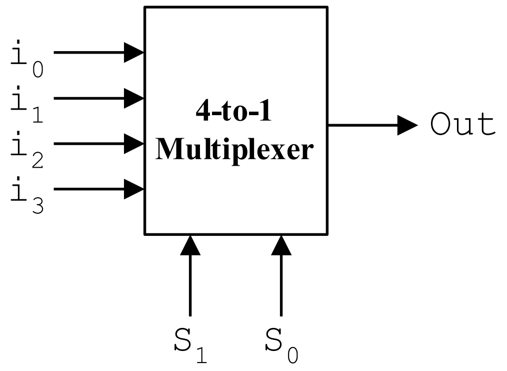
Operator

- Examples of Conditional Operator

```
//model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 36'bz;

//model functionality of a 2-to-1 multiplexer
assign out = control ? in1 : in0;
```

- Conditional operations can be nested. Each *true_expression* or *false_expression* can itself be a conditional operation

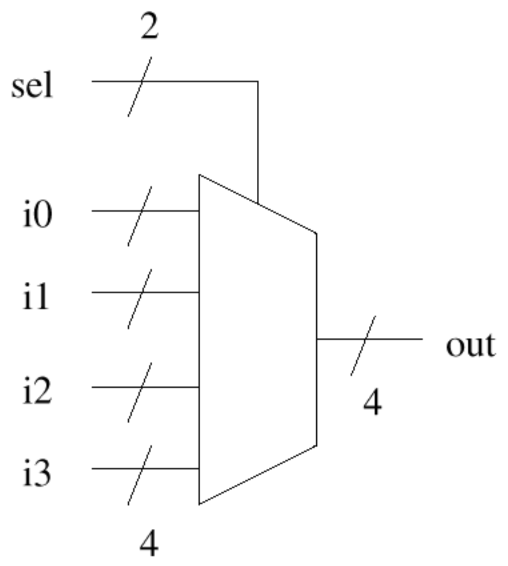
```
assign out = s1 ? ( s0 ? i3 : i2 ) : ( s0 ? i1 : i0 );
```



Operator

- Conditional Operator

```
module MUX4_1(out,i0,i1,i2,i3,sel);
  output [3:0] out;
  input [3:0] i0,i1,i2,i3;
  input [1:0] sel;

  assign out=(sel==2'b00)?i0:
             (sel==2'b01)?i1:
             (sel==2'b10)?i2:
             (sel==2'b11)?i3:
             4'bx;
endmodule
```



Operator

- Concatenation and Replication Operator

Concatenation operator in LHS

```
module add_32 (co, sum, a, b, ci);
    output co;
    output [31:0] sum;
    input   [31:0] a, b;
    input   ci;
      assign #100 {co, sum} = a + b + ci;
endmodule
```

Bit replication to produce *01010101*

```
assign byte = {4{2'b01}};
```

Sign Extension

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

Expression Bit Widths

- x ? y : z
  - Conditional
  - Bit width = max(width(y), width(z))
- {x, …, y}
  - Concatenation
  - Bit width = width(x) + … + width(y)
- {x{y, …, z}}
  - Replication
  - Bit width = x * (width(y) + … + width(z))

Expressions with Operands Containing x or z

- Arithmetic
  - If any bit is x or z, result is all x’s.
  - Divide by 0 produces all x’s.
- Relational
  - If any bit is x or z, result is x.
- Logical
  - == and != If any bit is x or z, result is x.
  - === and !== All bits including x and z values must match for equality

Outline

- Operation
- Assignment
- Blocking and non-blocking
- Appendix

Assignments

- Assignment: Drive value onto nets and registers
- There are two basic forms of assignment
  - continuous assignment, which assigns values to nets
  - procedural assignment, which assigns values to registers
- Basic form

<left hand side>=<right hand side>

Assignments	**Left Hand Side**
Continuous Assignment	Net *wire, tri*
Procedural Assignment	**Register** *reg, integer, real*

Left Hand Side = **LHS**

Right Hand Side = **RHS**

Assignments

- Continuous assignment

```
module holiday_1(sat, sun, weekend);
      input sat, sun; output weekend;
      assign weekend = sat | sun;     // outside a procedure
endmodule
```

- Procedural assignment

```
module holiday_2(sat, sun, weekend);
      input sat, sun; output weekend; reg weekend;
      always #1 weekend = sat | sun;     // inside a procedure
endmodule
```

```
module assignments
         // continuous assignments go here
always begin
         // procedural assignments go here
end
endmodule
```

Continuous Assignment

- Drive a value onto a wire, wand, wor, or tri
  - Use an explicit continuous assignment statement after declaration
  - Specify the continuous assignment statement in the same line as the declaration for a wire
- Used for datapath descriptions
- Used to model combinational circuits

Continuous Assignments

- Convenient for logical or datapath specifications

```
wire [8:0] sum;
wire [7:0] a, b;
wire carryin;

assign sum = a + b + carryin;
```

Define bus widths

Continuous assignment: permanently sets the value of sum to be a +b+carryin

Recomputed when a, b, or carryin changes

Continuous Assignments

- Continuous assignments provide a way to model combinational logic

continuous assignment

```
module inv_array(out,in);
  output [31:0] out;
  input [31:0] in;
  assign out=~in;
endmodule
```

gate-level modeling

```
module inv_array(out,in);
  output [31:0] out;
  input [31:0] in;
  not U1(out[0],in[0]);
  not U2(out[1],in[1]);
  ..
  not U31(out[31],in[31]);
endmodule
```

Continuous Assignments

- Examples of continuous Assignment

  assign out = i1 & i2;

  // ***i1*** and ***i2*** are nets

  assign addr[15:0] =addr1[15:0] ^ addr2[15:0]

  // Continuous assign for vector nets ***addr*** is a 16-bit vector net

  // ***addr1*** and ***addr2*** are 16-bit vector registers

  assign {cout, sum[3:0]}=a[3:0]+b[3:0]+cin;

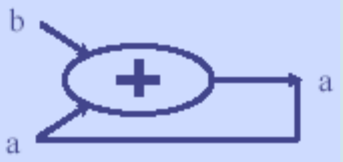
  // **LHS** is a concatenation of a scalar net and vector net
- Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared.

```
wire a;          --declare
assign a=b&c;    --assign

wire a=b&c;      --declare and assign
```

Continuous Assignment

- Avoid logic loop
  - HDL Compiler and Design Compiler will automatically open up asynchronous logic loops
  - Without disabling the combinational feedback loop, the static timing analyzer can't resolve
  - Example

```
assign a = b + a ;
```



Procedural Assignments

- Procedural assignments drive values or expressions onto registers (*reg*, *integer*, *real*, *time*)

```
module adder32(sum,cout,a,b,ci);
  output [31:0] sum;
  output cout;
  input [31:0] a,b;
  input ci;
  reg [31:0] sum;
  reg cout;

  always @(a or b or ci)
    {carry,sum}=a+b+ci;
endmodule
```

Procedural Assignments

- Inside an initial or always block:

```
initial
  begin
    {cout, sum} = a + b + cin;
  end
```

```
always
  begin
    {cout, sum} = a + b + cin;
  end
```

- Just like in C: RHS evaluated and assigned to LHS before next statement executes
- RHS may contain wires and regs
  - Two possible sources for data
- LHS must be a reg
  - Primitives or cont. assignment may set wire values

Outline

- Operation
- Assignment
- Blocking and non-blocking
- Appendix

Procedural Assignments

- *Blocking* assignment statements are executed in the order they are specified in a sequential block

```
reg x,y,z;
reg [7:0] rega,regb;
integer count;
initial
  begin
      x=0;y=1;z=1;
      count=0;
      rega=8'b0;regb=rega;
      #15 rega[2]=1'b1;
      #10 regb[7:5]={x,y,z};
      count=count+1;
  end
```

time	statements executed
0	x=0; y=1; z=1; count=0; rega=0; regb=rega=0;
$\boxed{15}$	rega[2]=1;
$\boxed{25}$	regb[7:5]={x,y,z}; count=count+1;

Procedural Assignments

- *Nonblocking* assignment statements allow scheduling of assignment without blocking execution of the statements that follow in a sequential block. A **<=** operator is used to specify nonblocking assignments

```
reg x,y,z;
reg [7:0] rega,regb;
integer count;
initial
  begin
      x=0;y=1;z=1;
      count=0;
      rega=8'b0; regb=rega;
      rega[2] <= #15 1'b1;
      regb[7:5] <= #10 {x,y,z};
      count <= count+1;
  end
```

time	statements executed

0	x=0; y=1; z=1;
	count=0; rega=0;
	regb=rega=0;
	count=count+1;

$\boxed{\textbf{10}}$	**regb[7:5]={x,y,z};**
$\boxed{15}$	rega[2]=1;

Blocking vs. Non-Blocking (1/2)

- A sequence of nonblocking assignments don't communicate

```
always @(posedge clk)
begin
    a = 1;
    b = a;
    c = b;
end
```

Blocking assignment:

a = b = c = 1

```
always @(posedge clk)
begin
    a <= 1;
    b <= a;
    c <= b;
end
```

Nonblocking assignment:

a = 1

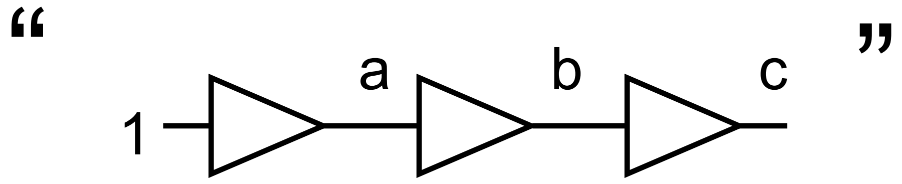
b = old value of a

c = old value of b

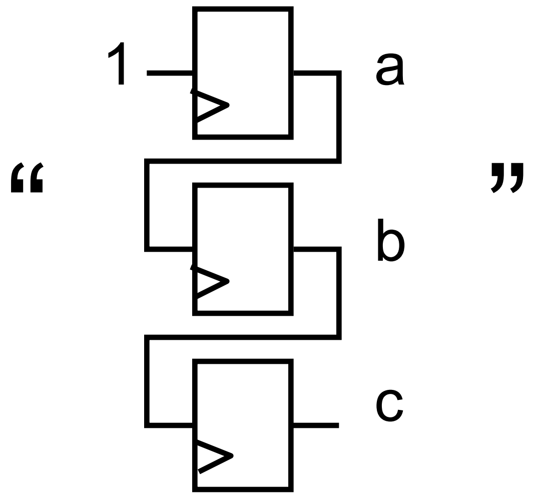
Blocking vs. Non-Blocking (2/2)

- RHS of nonblocking taken from flip-flops
- RHS of blocking taken from wires

```
a = 1;
b = a;
c = b;
```

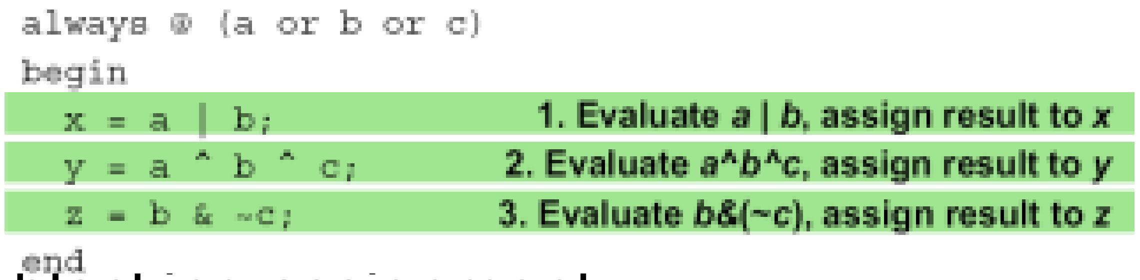


```
a <= 1;
b <= a;
c <= b;
```



Blocking or Non-Blocking?

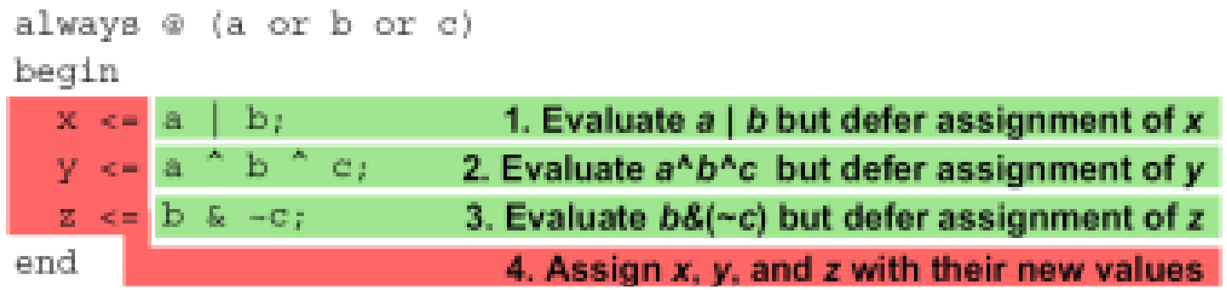
- Blocking assignment
  - Evaluation and assignment are immediate



```
always @ (a or b or c)
begin
   x = a | b;
   y = a ^ b ^ c;
   z = b & ~c;
end
```

1. Evaluate *a* | *b*, assign result to *x*
2. Evaluate *a*^*b*^*c*, assign result to *y*
3. Evaluate *b*&(~*c*), assign result to *z*

- Nonblocking assignment
  - All assignment deferred until all right-hand sides have been evaluated (end of the virtual timestamp)



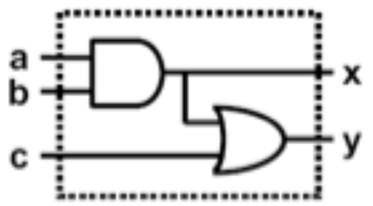
```
always @ (a or b or c)
begin
   x <= a | b;
   y <= a ^ b ^ c;
   z <= b & -c;
end
```

1. Evaluate *a* | *b* but defer assignment of *x*
2. Evaluate *a*^*b*^*c* but defer assignment of *y*
3. Evaluate *b*&(~*c*) but defer assignment of *z*
4. Assign *x*, *y*, and *z* with their new values

Blocking for Combinational Logic

- Both synthesizable, but both correctly simulated?
- Non-blocking assignment do not reflect the intrinsic behavior of multi-stage combinational logic

Blocking Behavior	a	b	c	x	y
(Given) Initial Condition	1	1	0	1	1
a changes; always block triggered	0	1	0	1	1
`x = a & b;`	0	1	0	0	1
`y = x | c;`	0	1	0	0	0



```
module blocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

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

endmodule
```

Nonblocking Behavior	a	b	c	x	y	*Deferred*
(Given) Initial Condition	1	1	0	1	1	
a changes; always block triggered	0	1	0	1	1	
`x <= a & b;`	0	1	0	1	1	x<=0
`y <= x | c;`	0	1	0	1	1	x<=0, y<=1
Assignment completion	0	1	0	0	1	

```
module nonblocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  always @ (a or b or c)
  begin
    x <= a & b;
    y <= x | c;
  end

endmodule
```

Non-Blocking for Sequential Logic

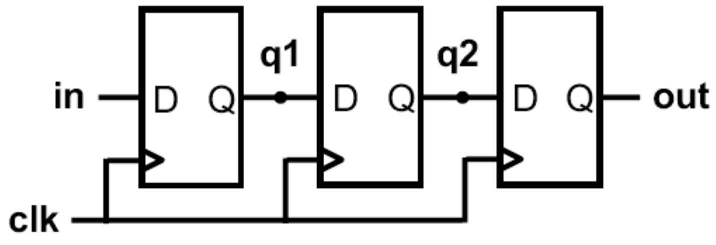
- Blocking assignment do not reflect the intrinsic behavior of multi-stage sequential logic

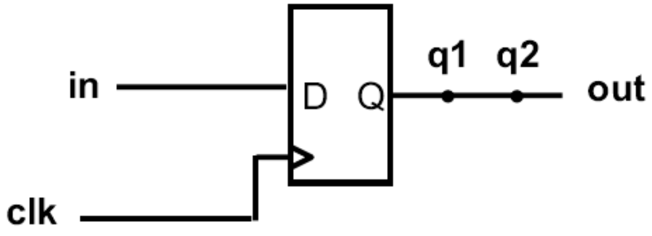
```
always @ (posedge clk)
begin
  q1 <= in;
  q2 <= q1;
  out <= q2;
end
```

```
always @ (posedge clk)
begin
  q1 = in;
  q2 = q1;
  out = q2;
end
```

"At each rising clock edge, *q1*, *q2*, and *out* simultaneously receive the old values of *in*, *q1*, and *q2*."

"At each rising clock edge, *q1* = *in*.
After that, *q2* = *q1* = *in*.
After that, *out* = *q2* = *q1* = *in*.
Therefore *out* = *in*."





Combinational & Sequential Logic Separation in Verilog Code

Mixed Style

```
always@(posedge clk or negedge rst)
begin
    if(~rst) begin
        counter <= 4'd0;
        out     <= 8'd0;
        finish  <= 1'd0;
    end
    else begin
        if(counter==4'd6) begin
            counter <= 4'd0;
            finish  <= 1'd1;
        end
        else begin
            counter <= counter+1'd1;
            finish  <= 1'd0;
        end
        out <= out + counter * in;
    end
end
```

Not recommended ✗

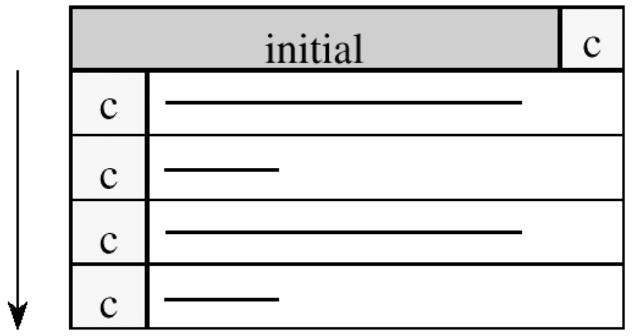
Separated Style

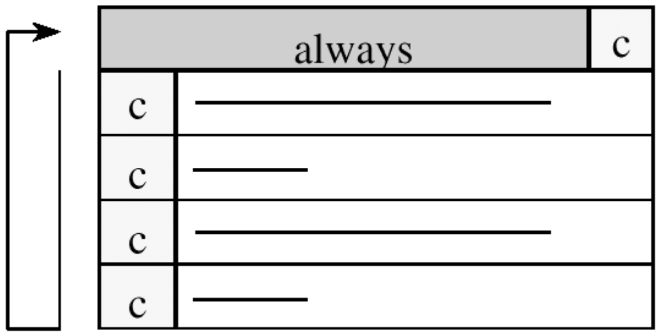
```
// combinational
always@(*) begin
    if(counter==4'd6) begin
        next_counter = 4'd0;
        next_finish = 1'd1;
    end
    else begin
        next_counter = counter+1'd1;
        next_finish = 1'd0;
    end
    next_out = out + counter * in;
end

// sequential
always@(posedge clk or negedge rst)
begin
    if(~rst) begin
        counter <= 4'd0;
        out     <= 8'd0;
        finish  <= 1'd0;
    end
    else begin
        counter <= next_counter;
        finish  <= next_finish;
        out     <= next_out;
    end
end
```

Preferred ✓

Sequential Block

- Sequential block may appear in an always or initial statement





Runs when simulation starts	Runs when simulation starts
Terminates when control reaches the end (one time sequential activity flow)	Restarts when control reaches the end (cycle sequential activity flow)
Good for providing stimulus (testbenches); not synthesizable	Good for modeling/specifying hardware

initial and always

- Run until they encounter a delay

```
initial begin
  #10 a = 1; b = 0;
  #10 a = 0; b = 1;
end
```

- or a wait for an event

```
always
  begin
       wait(i); a = 0;
       wait(~i); a = 1;
  end
```

Sequential and Parallel Blocks

- There are two types of blocks: **sequential** blocks and **parallel** blocks

```
//Illustration 1:sequential block without delay
reg x,y;
reg [1:0] z,w;
initial
  begin
    x=1'b0;
    y=1'b1;
    z={x,y};
    w={y,x};
  end
//Illustration 2: sequential blocks with delay
reg x,y;
reg [1:0] z,w;
initial
  begin
    x=1'b0;
    #5 y=1'b1;
    #10 z={x,y};
    #20 w={y,x};
  end
```

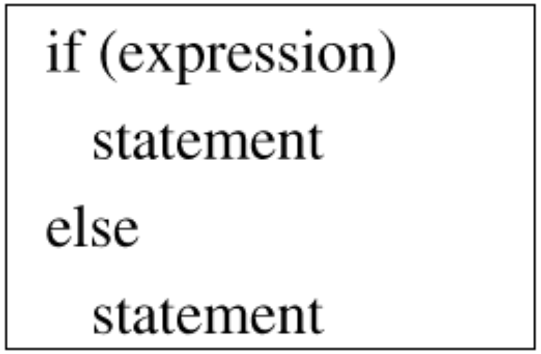
Sequential and Parallel Blocks

- Parallel blocks, specified by keywords fork and join, execute concurrently

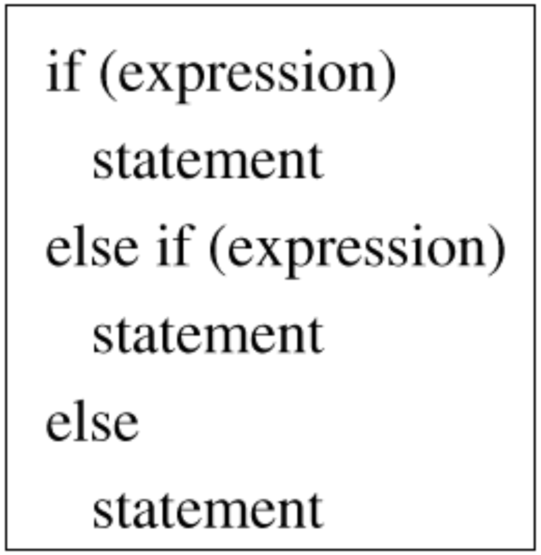
```
//Parallel blocks with delay
reg x,y;
reg [1:0] z,w;
initial
  fork
    x=1'b0;
    #5 y=1'b1;
    #10 z={x,y};
    #20 w={y,x};
  join
```

Conditional Statements

- If and If-else statements



```
if (expression)
    statement
else
    statement
```



```
if (expression)
    statement
else if (expression)
    statement
else
    statement
```

Conditional Statements

If and If-Else Statements (cont.)

- Examples

```
if (rega >= regb)
   result = 1;
else
   result = 0;
```

```
if (index > 0)
   if (rega > regb)
      result = rega;
   else
      result = 0;
else
   $display("* Warning * index is equal or small than 0!");
```

Multiway Branching

- The nested ***if-else-if*** can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the ***case*** statement

```
case (expression)
  alternative1: statement1;
  alternative2: statement2;
  alternative3: statement3;
      ...
  default: default_statement;
endcase
```

```
reg [1:0] alu_control;
      ...
case (alu_control)
  2'b00: y=x+z;
  2'b01: y=x-z;
  2'b10: x*z;
      ...
  default: y='bx;
endcase
```

Multiway Branching

- 4-to-1 Multiplexer with case Statement

```
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;
reg out;

always @(s1 or s0 or i0 or i1 or i2 or i3)
  case ({s1,s0})
    2'd0:out=i0;
    2'd1:out=i1;
    2'd2:out=i2;
    2'd3:out=i3;
    default: out=1'bx;
  endcase

endmodule
```

Multiway Branching

- There are 2 variations of the case statement. They are denoted by keywords casex and casez
  - **`casez`** treats all **`z`** values in the case alternatives or the case expression as don't cases. All bit positions with **`z`** can also represented by **`?`** In that position
  - **`casex`** treats all **`x`** and **`z`** values in the case item or the case expresssion as don't cares

```
reg [3:0] encoding;
integer state;

casex (encoding)
//logic value x represents a don't care bit
  4'b1xxx: next_state=3;
  4'bx1xx: next_state=2;
  4'bxx1x: next_state=1;
  4'bxxx1: next_state=0;
  default: next_state=0;
endcase
```

encoding=4'b10xz   next_state=3

While Loop

- The **while** loop executes until the **while-expression** becomes false

```
initial      //Illustration 1:
  begin
    count=0;
    while(count<128)
    begin
      $display("count=%d",count);
      count=count+1;
    end
  end
```

```
initial
  begin
    reg [7:0] tempreg;
    count = 0;
    tempreg = reg;
    while (tempreg)
      begin
        if (tempreg[0]) count = count + 1;
        tempreg = tempreg >> 1;
      end
```

rega = 101;

tempreg	count
101	1
010	1
001	2

For Loop

- The keyword **`for`** is used to specify this loop. The **`for`** loop contain 3 parts:
  - An initial condition
  - A check to see if the terminating condition is true
  - A procedural assignment to change value of the control variable

```
//Initialize array elements
`define MAX_STAGES 32
reg [0:31] array;
integer i;

initial
begin
  for(i=0;i<32;i=i+2)
    array[i]=0;
  for(i=1;i<32;i=i+2)
    array[i]=1;
end
```

Repeat Loop

- The keyword **`repeat`** is used for this loop. The **`repeat`** construct executes the loop a **fixed** number of times.

```
module multiplier(result, op_a, op_b);
  ...
  reg shift_opa, shift_opb;
  parameter size = 8;
  initial begin
    result = 0;  shift_opa = op_a;  shift_opb = op_b;
    repeat (size)
      begin
        #10  if (shift_opb[1])
                  result = result + shift_opa;
        shift_opa = shift_opa << 1;
        shift_opb = shift_opb >> 1;
      end
    end
endmodule
```

default repeat 8 times

Forever Loop

- The keyword **forever** is used to express the loop. The loop does not contain any expression and executes forever until the **$finish** task is encountered

```
//Clock generation
//Clock with period of 20 units)
reg clk;

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

```
//Synchronize 2 register values
//at every positive edge of clock
reg clk;
reg x,y;

initial
    forever @(posedge clk) x=y;
```

Modeling A Flip-Flop With Always

- Very basic: an edge-sensitive flip-flop

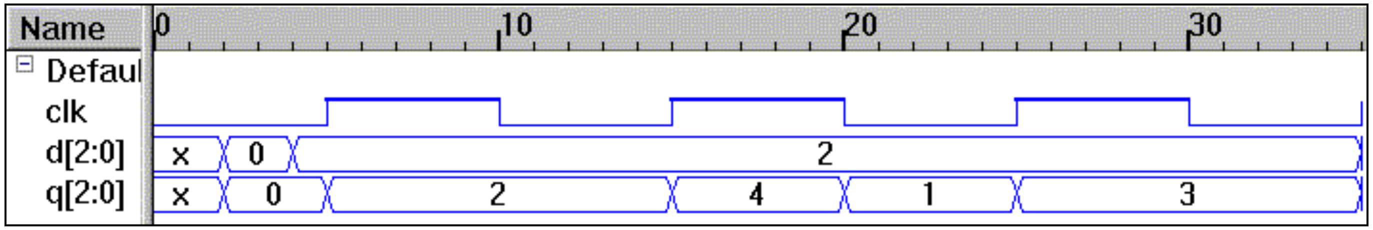
```
reg q;
always @(posedge clk)
q = d;
```

- q = d assignment runs when clock rises: exactly the behavior you expect
- Keywords:
  - posedge for positive edge trigger
  - negedge for negative edge trigger

Timing Controls

- Event-Based Timing Control
  - Regular Event Control

```
module test;
  reg clk;
  reg [2:0] q,d;
  always #5 clk=~clk;
  initial
    begin
      clk=0;
      #2 q=0;d=0;
      #2 d=2;
      @(clk) q=d;
      @(posedge clk) q=4;
      @(negedge clk) q=1;
      q=@(posedge clk) 3;
      #10 $finish;
    end
endmodule
```



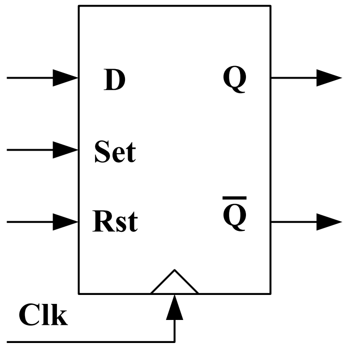
Timing Controls

- Event-Based Timing Control
  - Named Event Control

```
event received_data;
always @(posedge clk)
  begin
    if (last_data_packet)
      -> received_data;
  end
always @(received_data)
  data_buf = {data_pkt[0],data_pkt[1]};
```

  - Event OR Control

```
always @(rst or posedge clk or set)
  begin
    if(rst) q=0;
      else if (set) q=1;
        else q=d;
  end
assign _q=~q;
```



Timing Controls

- **Level-Sensitive Timing Control**

```
always
 wait (count_enable) #20 count = count +1 ;
 // 1. If count_enable is logical 1, count=count+1
 //    after 20 time unit.
 // 2. If count_enable stays at 1, count will be
 //    incremented every 20 time units
```

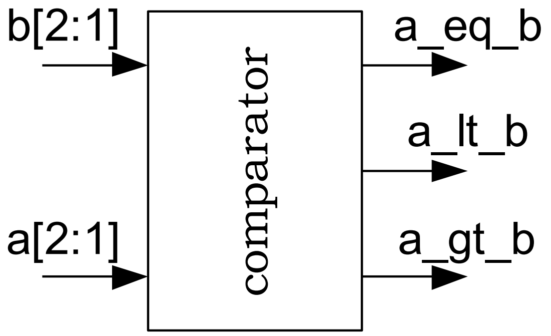
Outline

- Operation
- Assignment
- Blocking and non-blocking
- Appendix

Block Disable

- Disabling named blocks (example: comparator)

```
module comparator(a,b,a_gt_b,a_lt_b,a_eq_b);
  parameter size=2;
  input [size:1] a,b;
  output a_gt_b,a_lt_b,a_eq_b;
  reg a_gt_b,a_lt_b,a_eq_b;
  integer k;

  always @(a or b) begin: compare_loop
    for(k=size;k>0;k=k-1) begin
       if(a[k]!=b[k]) begin
         a_gt_b=a[k];
         a_lt_b=~a[k];
         a_eq_b=0;
         disable compare_loop;
       end
    end
    a_gt_b=0;
    a_lt_b=0;
    a_eq_b=1;
  end
endmodule
```



Tasks

- Task are declared with the keyword **`task`** and **`endtask`**. It must be used if any one of the following conditions is true for the procedure
  - There are delay, timing, or event control constructs in the procedure
  - The procedure has zero or more than one output arguments
  - The procedure has no input argument

Tasks example

```
module operation;
...
  parameter delay=10;
  reg [15:0] A,B;
  reg [15:0] AB_AND,AB_OR,AB_XOR;

  always @(A or B)
    begin
      bitwise_oper(AB_AND,AB_OR,AB_XOR,A,B);
    end
...
  task bitwise_oper;
    output [15:0] ab_and,ab_or,ab_xor;
    input [15:0] a,b;
    begin
      #delay ab_and=a&b;
      ab_or=a|b;
      ab_xor=a^b;
    end
  endtask
endmodule
```

Functions

- Functions are declared with the keywords **function** and **endfunction**. Functions are used if all of the following conditions are true for the procedure
  - There are no delay, timing, or event control constructs in the procedure
  - The procedure returns a single value
  - There is at least one input argument
- There are some peculiarities of functions. When a function is declared, a register with name (name of function) is declared implicitly inside.
- Functions cannot invoke other tasks. They can only invoke other functions

Functions example

```
module check(number);
  input [9:0] number;
  reg         correct;

  always @(number)
  begin
    correct = berger(number);
    $display("The correct of the number: %b is %b
             ",number,correct);
  end
  ...
function berger;
  input [9:0] number;
  reg [2:0] temp3;
  integer i;
    begin
      temp3 = 3'b000;
      for(i=3;i<10;i=i+1)
        temp3 = temp3 + number[i];
      temp3 = temp3 ^ 3'b111;
      if( temp3 === number[2:0])
         berger = 1'b1;
      else
         berger = 1'b0;
      end
endfunction
endmodule
```

```
module test;
  reg [9:0] num;
  check chk1(num);
  initial
    begin
          num = 10'b0111010011;
      #10 num = 10'b0111111011;
      #10 $finish;
    end
endmodule
```

simulation result

```
The correct of the number: 0111010011 is 1
The correct of the number: 0111111011 is 0
```

Differences Between Tasks and Functions

- Tasks and Functions

Functions	Tasks
A function can enable another function but not another task	A task can enable other tasks and functions
Function always execute in 0 simulation time	Tasks may execute in non-zero simulation time
Functions must not contain any delay, event, or timing control statements	Tasks may contain delay, event, or timing control statements
Functions must have **at least one input** argument. They can have more than one input	Task may have **zero or more** arguments of type **input**, **output** or **inout**
Functions always return a single value. They cannot have **output** or **inout** arguments	Tasks do not return with a value but can pass multiple values through **output** and **inout** arguments