Verilog

- Behavioral Modeling -

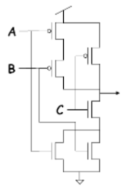
May 2014

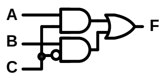
Ando KI

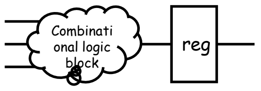
Contents

Levels of abstraction

- Structural levels
  - Switch level
    - Referring to the ability to describe the circuit as a netlist of transistor switches.
  - Gate level
    - Referring to the ability to describe the circuit as a netlist of primitive logic gates and functions.
- Functional levels
  - Register transfer level
    - Referring to the ability to describe the circuit as data assignment.
  - Behavioral level
    - Referring to the ability to describe the behavior of circuit using abstract constructs such as loops and processes.







Verilog behavioral model

- Verilog <u>behavioral models</u> contain <u>procedural statements</u>.
- The procedural statements <u>control the simulation</u> and <u>manipulate variables</u> of the data types.
- The procedural statements are contained within <u>procedures</u>.
- Each procedure starts a separate activity flow.
- All of the activity flows are <u>concurrent</u>.

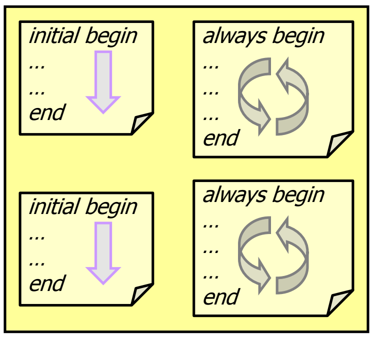
- There are two types of procedures.
  - 'initial' construct
    - Starts at simulation time 0.
    - Executes once.
  - 'always' construct
    - Starts at simulation time 0.
    - Execute repetitively.

Behavioral control constructs

- ‘**initial**’ block
  - One-time sequential activity flow from simulation start.
  - Initial blocks start execution at simulation time zero and finish when their last statement executes.
- ‘**always**’ block
  - Cyclic (repetitive) sequential activity flow
  - Always blocks start execution at simulation time zero and continue until simulation finishes.

```
initial statement;
```

```
always statement;
```



Procedural assignments

- The assignment is the basic mechanism for placing values into nets and variables.

Assignment type	Left-hand side (LHS)
Continuous	Net (wire)
Procedure	Variable (reg, integer, or time variable)

- The procedural assignment
  - It assigns values to variables.
  - It occurs within procedures, such as always, initial, task, and function.

```
wire  [3:0] w;
reg [3:0] a, b, c, d;
initial a = 4'h4;
always @ (c) b = c;
always @ (w) d = w;
```

  - Refer to the continuous assignment
    - It assigns values to nets.
    - It occurs whenever the value of the right-hand side changes.

```
wire   wire_tmp1;
assign  wire_tmp1 = my_value;
wire wire_tmp2 = my_value;
```

Types of procedural assignments

- Blocking procedural assignment
  - Shall be executed before the execution of the statements that follow it in a sequential block.
    - Note that 'parallel block' is made of 'fork/join'.
    - Note that 'sequential block' is called 'begin/end' block.

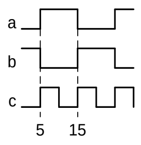
```
variable = expression;
```

- Non-blocking procedural assignment
  - Allows assignment scheduling without blocking the procedural flow.
  - Can be used whenever several variable assignments within the same time step.

```
variable <= expression;
```

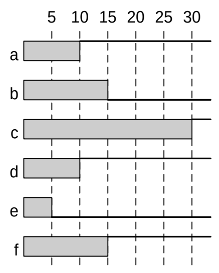
Procedural assignment (1/3)

```
module top;
reg a, b, c;
initial begin
        a = 0;
        b = 1;
        c = 0;
end
always c = #5 ~c;
always @ (posedge c) begin
        a <= b;
        b <= a;
end
endmodule
```



'a' and 'b' swap their value at every rising edge of 'c'.

```
module top;
reg a, b, c, d, e, f;
initial begin
        a = #10  1;
        b = #5   0;
        c = #15  1;
end
initial begin
        d <= #10  1;
        e <= #5   0;
        f <= #15  1;
end
endmodule
```



Note that what are different with blocking and non-blocking assignments.

```
#delay  b = statement;
    // the execution (assignment) is delayed by 'delay' time units.
a = #delay statement;
    // the 'statement' is evaluated, but the result
    // is updated to 'a' after 'delay' time units.
```

Procedural assignment (2/3)

```
module top;
reg a, b;
initial begin
        a = 0;
        b = 1;
        a <= b;
        b <= a;
    #10 $finish;
end
endmodule
```

The result:
a = 1;
b = 0;

```
module top;
reg a;
initial a  =      1;
initial a <= #4  0;
initial a <= #4  1;
endmodule
```

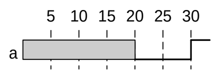
The result is indeterminate, since several parallel block runs concurrently.

```
module top;
reg a;
initial a = 1;
initial begin
        a <= #5  0;
        a <= #5  1;
end
endmodule
```

a will be 1.
It is determinate.

```
module top;
  reg a;
  initial #20 a <= #10 1;
  initial #15  a <= #5  0;
endmodule
```

The result is determinate, since two parallel blocks update ‘a’ in different time slot.



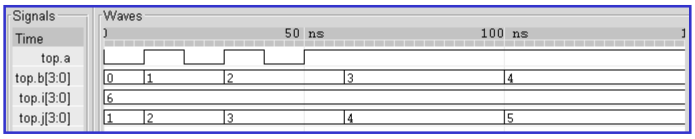
Procedural assignment (3/3)

- ‘codes/verilog/behavioral/block_non’

```
`timescale 1ns/1ns
module top;
  reg a;
  reg [3:0] b;
  reg [3:0] i, j;
  initial begin
        for (i=0; i<=5; i=i+1)          a <= # (i*10) i[0];
        for (j=0; j<=5; j=j+1) # (j*10) b <= j;
  end
  initial begin
        $dumpfile("wave.vcd");
        $dumpvars(1);
        #1000 $finish;
  end
endmodule
```

```
a <= #0   0;
a <= #10  1;
a <= #20 0;
a <= #30 1;
a <= #40 0;
a <= #50 1;
```

```
#0  b <= 0;
#10 b <=  1;
#20 b <= 2;
#30 b <= 3;
#40 b <= 4;
#50 b <= 5;
```



‘if-else’ statement

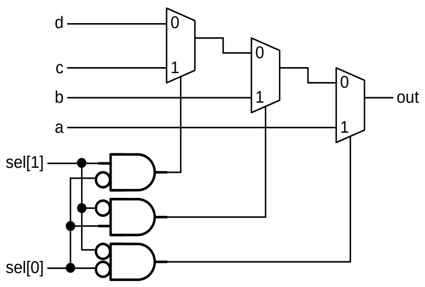
- The ‘if-else’ statement is used to make a decision as to whether a statement is executed or not.
- The evaluation of ‘expression’ will be true when the result is non-zero known value. Otherwise, false when the result is 0, x, or Z.

```
if (expression) statement;
// the statement is executed,
//  when the express is true.
```

```
if (expression) statementA;
else            statementB;
```

```
if (expressionA)       statementA;
else if (expressionB) statementB;
else                  statementC;
```

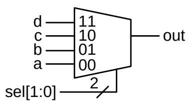
```
always @ (sel or a or b or c or d)
  if      (sel == 2'b00) out = a;
  else if (sel == 2'b01) out = b;
  else if (sel == 2'b10) out = d;
  else                   out = d;
```



‘case’ statement

- ‘case’ statement is a multiway decision statement that tests whether an expression matches one of a number of other expressions and branches accordingly.
- ‘default’ statement shall be optional.
  - Use of multiple default statements in one case statement shall be illegal.
- If one of the case item matches the case expression given in parentheses, then the statement associated with that case item shall be executed.
- If all comparison fail and the default item is given, then the default item statement shall be executed.
  - If the default is not given, then none is executed.

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



Looping statements (1/2)

- 'forever'
  - Continuously executes a statement
- 'repeat'
  - Executes a statement a fixed number of times.
  - If the expression evaluates to unknown or high impedance, it shall be treated as zero.
- 'while'
  - Executes a statement until an expression becomes false.
- 'for'
  - If the condition results in zero, the for loop shall exit.

```
forever statement;
```

```
repeat (expression) statement;
```

```
while (expression) statement;
```

```
for (initial; condition; step)
       statement;
```

Looping statements (2/2)

```
parameter size=8, longsize=16;
reg [size:1] opa, opb;
reg [longsize:1] result;
begin : mult
       reg [longsize:1] shift_opa, shift_opb;
       shift_opa = opa; shift_opb = opb;
       result = 0;
       repeat (size) begin
          if (shift_opb[1]) result = result + shift_opa;
          shift_opa = shift_opa << 1;
          shift_opb = shift_opb >> 1;
       end
end
```

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

```
begin
    initial_assignment;
    while (condition) begin
         statement;
         step_assignment;
    end
end
```

```
for (initial_assignment;
      condition;
      step_assignment) statement;
```

Procedural timing control

- Types of timing control
  - Delay control
    - An expression specifies the timing duration between initially encountering the statement and when the statement actually executes.
  - Event control
    - It allows statement execution to be delayed until the occurrence of some simulation event occurring.
    - Implicit event & named event

Delay control

- A procedural statement following the delay control shall be delayed in its execution with respect to the procedural statement preceding the delay control by the specified delay.
- If the delay_expression evaluates to an unknown or high-impedance value, it shall be interpreted as zero delay.

```
# delay_express    statement;
```

```
#10 rega = regb;
// the execution (assignment) is delayed by 10 time units.
```

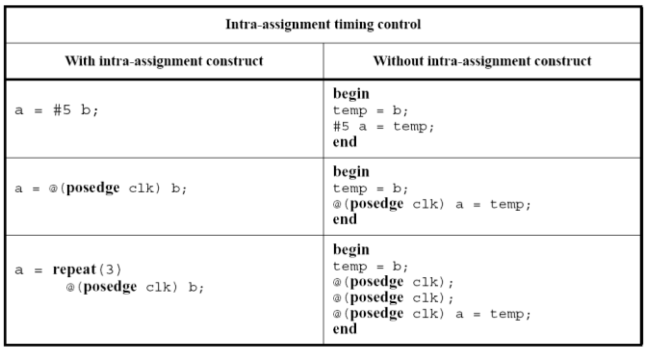
```
#d          rega = regb;   // d is defined as a parameter
# ((d+e)/2) rega = regb; // delay is average of d and e
#regr       regr = regr +1; // delay is the value in regr
```

Intra-assignment delay/event

- An intra-assignment delay or event shall delay the assignment of the new value to the LHS, but the RHS expression shall be evaluated before the delay, instead of after the delay.

```
repeat (a) @ (event_expression);
// it wait until 'a' times events occurances.
```

```
a = #intra_assignment_delay    statement;
// the 'statement' is evaluated, but the result
// is updated to 'a' after 'delay' time units.
```



Intra-assignment timing control	
With intra-assignment construct	**Without intra-assignment construct**
`a = #5 b;`	**begin** `temp = b;` `#5 a = temp;` **end**
`a = @(posedge clk) b;`	**begin** `temp = b;` `@(posedge clk) a = temp;` **end**
`a = repeat(3) @(posedge clk) b;`	**begin** `temp = b;` `@(posedge clk);` `@(posedge clk);` `@(posedge clk) a = temp;` **end**

Wire/net delay

```
wire #(delay) a, b, c;

//specify the delay properties of a wire
// "delay" = "min:typ:max"
// "delay" ="rising, falling, turn-off"
```

```
wire #(delay) a = b & c;

// implicit continuous assignment delay
```

Implicit event

- Implicit event comes from the value changes on nets and variable.
- Kinds of event
  - Positive edge
    - Towards the value 1
  - Negative edge
    - Towards the value 0
  - If the expression evaluates to more than a 1-bit result, the edge transitions shall be detected on the LSB.

```
// level sensitive
@ expression    statement;
// edge sensitive
@ (posedge expression) statement;
@ (negedge expression) statement;
```

```
@r rega = regb; // controlled by any value change in r
@ (posedge clk) rega = regb;
@ (negedge clk) rega = regb;
```

```
always @ (posedge clk or negedge reset) begin … end
always @ (posedge clk or sig1 or sig2) begin … end
always @(*) // equivalent to @(a or b or c or d or f)
y = (a & b) | (c & d) | myfunction(f);
always @* begin // equivalent to @(a or b or c or d or
tmp1 or tmp2)
    tmp1 = a & b; tmp2 = c & d; y = tmp1 | tmp2;
end
always @* begin // equivalent to @(a or b or c or d)
    x = a ^ b;
    @* // equivalent to @(c or d)
    x = c ^ d;
end
```

	To	0	1	x	z
From					
0		No edge	posedge	posedge	posedge
1		negedge	No edge	negedge	negedge
x		negedge	posedge	No edge	No edge
z		negedge	posedge	No edge	No edge

Named event

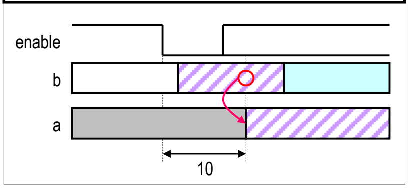
- Named event is also called as abstract event.
- The keyword 'event' declares a new data type called event.
- The event trigger operator '->' makes event.

```
module flop_event (clk, rstb, data, q, qb);
input   clk, rstb, data;
output  q, qb;
reg     q;
event   rise_event;
assign  q_bar = ~q;
@ (posedge clk) -> rise_event;
@ (rise_event or negedge rstb)
    if (rstb==1'b0) q <= 0;
    else            q <= data;
endmodule
```

Level-sensitive event control

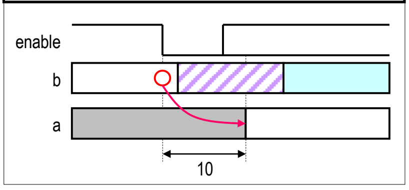
wait_statement ::= **wait (** expression **)** statement_or_null

```
begin
    wait (!enable) #10 a = b;
   #10 c = d;
end
/* If the value of enable is 1 when the
block is entered, the wait statement will
delay the evaluation of the next statement
(#10 a = b;) until the value of enable
changes to 0. If enable is already 0 when
the begin-end block is entered, then the
assignment "a = b;" is evaluated after a
delay of 10 and no additional
delay occurs. */
```



Intra-assignment delay

```
begin
    wait (!enable) a = #10 b;
   #10 c = d;
end
/* If the value of enable is 1 when the
block is entered, the wait statement will
delay the evaluation of the next statement
(a = #10 b;) until the value of enable
changes to 0. If enable is already 0 when
the begin-end block is entered, then the
assignment "a = b;" is evaluated, but its
effect is delayed of 10. */
```



Block statements

- The block statements are a means of <u>grouping two or more statements</u> together so that they act syntactically like a single statement.
- Types of block statement
  - Sequential block
    - Begin-end block
    - The procedural statements in it shall be executed sequentially in the given order.
  - Parallel block
    - Fork-join block
      - The procedural statements in it shall be executed concurrently.
- Block names
  - The block statement can be named by adding ': name_of_block' after the keyword 'begin' or 'fork'.

Sequential and parallel block

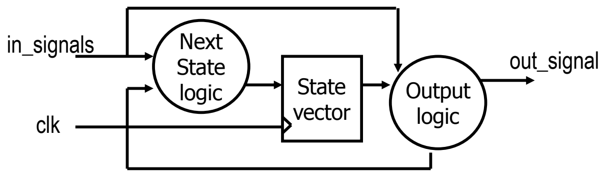
- Sequential block
  - Statements shall be executed in sequence, one after another.
  - Control shall pass out of the block after the last statement executes.
- Parallel block
  - Statement shall execute concurrently.
  - Control shall pass out of the block when the last time-ordered statement executes.

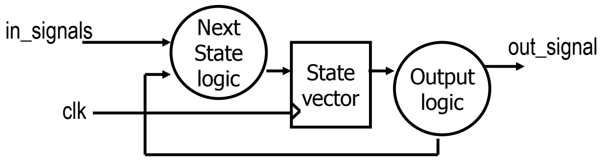
```
parameter d = 50;
reg [7:0] r;
begin : an_example_of_seq_block
    #d  r = 'h35;
    #d  r = 'hE2;
    #d  r = 'h00;
    #d  r = 'hF7;
    #d  -> end_wave;
end
```

```
parameter d = 50;
reg [7:0] r;
fork : an_example_of_par_block
    #d  r = 'h35;
    #d*2  r = 'hE2;
    #d*3  r = 'h00;
    #d*4  r = 'hF7;
    #d*5  -> end_wave;
join
```

State machine

- Mealy state machine
  - The output logic is always a function of the current state (state vector) and the inputs.
- Moore state machine
  - The output logic is only a function of the current state, i.e. inputs are not included.





Implementing state machine

- One process approach
  - One always block for computing and updating the state vector and outputs
- Two processes approach
  - Case 1
    - One always block for updating the state vector
    - One always block for both the output and the next state logic
  - Case 2
    - One always block for output
    - One always block for updating the state vector and the next state logic
- Three processes approach
  - One always block for updating the state vector
  - One always block for the output logic
  - One always block for the next state logic

Contents

See the project 02

Reading

- IEEE Std. 1364-2001, IEEE Standard Verilog Hardware Description Language. (Chapter 9. Behavioral Modeling)