The Verilog Hardware Description Language - A Behavioural View

Overview

In this lesson we will

- ✓ Introduce and explore the Verilog behavioural level model.
- ✓ Introduce the behavioural operators.
- ✓ Study the procedural assignment behavioural model.
- ✓ Introduce and study the behavioural flow of control constructs.
- ✓ Examine blocking and nonblocking assignments.
- ✓ Examine real-world effects under the behavioural model.
- ✓ Study Verilog behavioural level models of combinational and sequential circuits.

The Behavioural Model

The behavioral model increases the design abstraction

By an additional level

Thinking about the design

Moves above considerations of the flow of data within the system

Moving to algorithmic level

To the algorithms that express the behavior of the system

At the behavioral level

Model begins to appear more like a C or C++ program than a digital circuit

Remember that it is not a C or C++ program

Flow of control through the system

Expressed in the familiar looping and branching constructs

Rather than in logic gates

Program Structure

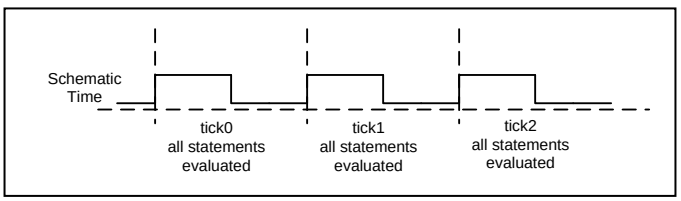
At the behavioral level

One of the major differences between languages such as C or C++ becomes clear.

Unlike either C or C++in which flow of control is generally *sequential*

Flow of control in Verilog is *concurrent*

- ➢ Statements in C or C++ execute in series
- ➢ Those in Verilog execute in parallel



always and initial Statements

At the behavioral level

Verilog program is structured as

Collection of *initial* and/or *always* blocks

Each such block

- ✓ Express a *separate flow of control*
- ✓ Each will finish execution independent of any other block

Module may define multiple *initial* and/or *always* blocks

Such blocks cannot be nested

Cannot have *initial* within *initial* or *always* within *always*

Beyond the input, output, wire statements, and parameter declarations

All behavioral statements must be included in either one of these blocks

Statements contained in an *initial block*

Delimited by *begin* and *end*

Just like { and } in C, C++, or Java

Evaluated one time at the start of a simulation

All *initial* blocks

Evaluated at same high level time tick

Statements contained in an *always* block

Delimited by *begin* and *end*

Just like { and } in C, C++, or Java

Evaluated continuously from the start of a simulation

All *always* blocks

Evaluated at same high level time tick

The *always* and *initial* statements

Two keywords that allow one to set stimuli to a module

The syntax for the *initial* statement is given as

Syntax

```
initial
begin
        Initial statements
end
```

The syntax for the always statement is given as

Syntax

```
always
begin
        Statements to be always executed
end
```

Operators

Like dataflow model

Syntax and operators used in Verilog at the behavioural level

Follow that of the C language very closely

Table below several additional operators

Operator	***Symbol***	***Operation***
Reduction	& A & B	Reduction AND
	~& A ~& B	Reduction NAND
	| A | B	Reduction OR
	~| A ~| B	Reduction NOR
	^ A ^ B	Reduction XOR
	~^ A ~^ B	Reduction XNOR
Condition	?: A ? B : C	If else
Concatenation	{expr0,expr1..exprn-1}	Concatenate smaller expressions
Replication	{number {expr0,expr1..exprn-1}	Expr set replicated number times

Reduction

- Reduction operators
  - Operate on all *bits* of single operand
    - Product 1 *bit* result
  - If any *bit* in operand is z or x
    - Result is x
- Reduction AND
  - If any bit in operand is 0 result is 0 else result is 1
- Reduction NAND
  - Inverse of reduction AND
- Reduction OR
  - If any bit in operand is 1 result is 1 else result is 0
- Reduction NOR
  - Inverse of reduction OR
- Reduction XOR
  - If even number of 1’s in operand result is 0 else result is 1
- Reduction XNOR
  - Inverse of reduction XOR

Condition

- Condition operator
  - Similar to triple operator in C

```
condExpr ? expr0 : expr1
```

  - if condExpr is true
    - return expr0
  - else
    - return expr1

Concatenation

Concatenation is operation of joining bits

From smaller expressions to form larger one

```
a[7:0] = {b[3:0], c[3:0]};
    Builds 8 bit expression from two four bit ones

a[7:0] = {a[3:0], a[7:4]};
    Swaps upper and lower fields within number
```

Replication

Creates expression by replicating and concatenating

Target expression

Specified number of times

```
dbus = { 4{4'b1101} };
    Builds the 16 bit expression 1101110111011101

abus = { 8{dbus[7]}, {dbus} };
    Implements sign extension for a signed number
    Assumes MSB is the sign bit
```

Procedural Assignment

Assignment in the behavioral model differs from

That in either the gate level or dataflow model

In the behavioral model procedural assignment statements

Used to update the circuit state variables

In the dataflow model

Continuous assignment construct - *assign*

Continually updates the value on the net on left hand side

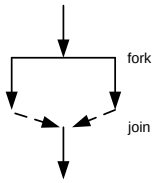
In the behavioral model

Value is only updated

As result of the execution of a procedural assignment statement

Verilog supports

- Two kinds of procedural assignment
  Blocking and *non-blocking*
- Two kinds of blocks
  Sequential and *parallel*




```
fork
      //  all execute in parallel
      S0
      S1
      S2
join
…not supported by synthesis tools
```

- Statements in a *sequential* block
  Delimited by a *begin* and an *end*
  Executed in sequence
- Statements in a *parallel* block
  Delimited by a *fork* and a *join*
  Executed in parallel
    Each statement in fork and join
      Expresses separate thread of execution

Blocking assignment statements

Executed in the order written in a sequential block

- Will block the execution of subsequent statements
  That appear in the *same sequential block*
- Will not block the execution of statements
  That appear in a parallel block or other sequential block

Non-blocking assignment statements

Will not block subsequent statements in a sequential block

Blocking assignment will successively evaluate

Right hand side then the *left hand* side

Each an assignment statement in a *sequential* block

$R_1 \rightarrow L_1, R_2 \rightarrow L_2, R_3 \rightarrow L_3$

Non-blocking assignment will evaluate *all*

Right hand sides then *all* of the *left hand* sides

Each statement in a *sequential* block

$R_1, R_2, R_3 \rightarrow L_1, L_2, L_3$

Syntax for the two types of assignment is given in the following

Syntax

Blocking

```
aVariable = aValue;
```

Nonblocking

```
aVariable <= aValue;
```

The Real-World Affects – Part 3

Delays

Delays may be incorporated on either side of the assignment statement

Syntax

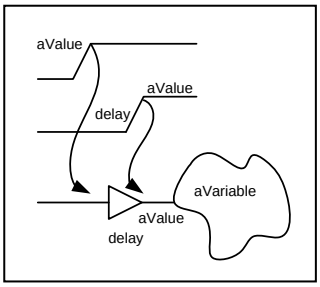
Blocking

```
aVariable = #d  aValue;   //  eval aValue (val at t – delay) → delay (block) before assignment to LHS
#d aVariable = aValue;    //  delay (block) → eval aValue (val at t + delay) then assign to RHS
```

Nonblocking

```
aVariable <= #d aValue;  //  eval aValue (val at t – delay) → delay before assignment to LHS  but continue
#d aVariable <= aValue   //  delay → eval aValue (val at t + delay) then assign to RHS but continue
```

How each is interpreted can be a bit confusing



Within a sequential block

Blocking

The first statement says

Evaluate *aValue*

- Then block processing of other statements in block for d time units

  Before assigning *aValue* to *aVariable*

- Any subsequent use of *aVariable*

  Will get the value from d time units in past

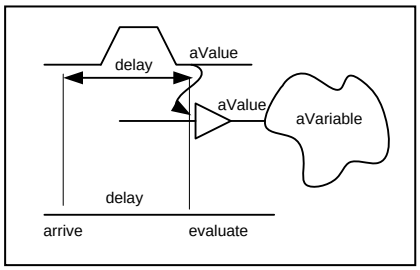
The second statement says

Block for d time units

Before evaluating *aVariable = aValue*

The variable *aVariable* will have the value *aValue*

d time units in future



Non-blocking

The first statement says

Evaluate *aValue*

Schedule *aVariable* to be updated d time units later

However continue processing other statements

Any other variables using the value of *aVariable*

Within the next d time units will be assigned the old value

The second statement says

Wait d time units before evaluating *aVariable = aValue*

Variable *aVariable* will have the value *aValue*

d time units in future

All the initial blocks are running in parallel

```
//  Illustrate Procedural blocking and
//  nonblocking assignment
//  Separate initial block

module blockingNonblocking();
//  declare temp registers
    reg a,b,c,d,e,f,g,h,i,j,k,l;

//  initial block – starts t0

//  initialize reg variables
    initial
    begin
        a = 0; b = 0; c = 0; d = 0; e = 0; f = 0;
        g = 0; h = 0; i = 0; j = 0; k = 0; l = 0;
    end

//  initial block – starts t0
    initial
    begin
    //  delay on right hand side

        //  blocking – execute in sequence
        a = #10 1;    //  t = 10
        b =  #2 1;        //  t = 12
        c =  #4 1;        //  t = 16

        //  nonblocking – arrive @ t = 16
        d <= #10 1;      //  t = 26
        e <=   #2 1;     //  t = 18
        f <=    #4 1;    //  t = 20
    end
```

```
//  Illustrate Procedural blocking and nonblocking assignment
//  Separate initial block

//  initial block – starts t0

initial
begin
//  delay on left hand side

    //  blocking – starts t0
    //  delay before eval
    #10 g = 1;    //  t = 10
      #2 h = 1;    //  t = 12
      #4 i  = 1;    //  t = 16

    //  nonblocking – arrive @ t = 16
    //  delay before eval
    #10 j <= 1;   //  t = 26
      #2 k <= 1;  //  t = 28
      #4 l  <= 1;  //  t = 32

End

//  initial block – starts T0

initial
begin
    $display("\ttime, \ta, \tb, \tc, \td, \te, \tf, \tg, \th, \ti, \tj, \tk, \tl");
    $monitor($time, " \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b,
    \t%b, \t%b, \t%b, t%b",a,b,c,d,e,f,g,h,i,j,k,l);
    #50 $finish(1);
end
```

Parallel

	time,	a,	b,	c,	d,	e,	f,	g,	h,	i,	j,	k,	l
blocking	0	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
	10	1,	0,	0,	0,	0,	0,	1,	0,	0,	0,	0,	0
	12	1,	1,	0,	0,	0,	0,	1,	1,	0,	0,	0,	0
	16	1,	1,	1,	0,	0,	0,	1,	1,	1,	0,	0,	0
nonblocking	18	1,	1,	1,	0,	1,	0,	1,	1,	1,	0,	0,	0
	20	1,	1,	1,	0,	1,	1,	1,	1,	1,	0,	0,	0
	26	1,	1,	1,	1,	1,	1,	1,	1,	1,	1,	0,	0
	28	1,	1,	1,	1,	1,	1,	1,	1,	1,	1,	1,	0
	32	1,	1,	1,	1,	1,	1,	1,	1,	1,	1,	1,	1

From the execution of the code fragment

- Variables *a* and *g* from the two initial blocks change state at time 10
  Variables (*b* and *c*) and (*h*, and *i*) follow similarly
  According to their specified delays or 2 and 4 time units
  After *a* and *g* respectively

- After the blocking statements have been evaluated
  Non-blocking statements are evaluated.
- Variable *d* is assigned the value 1 10 time units
  After the blocking statements in the first initial block
  Expression *j<=1* is evaluated 10 time units
  After the blocking statements in the second initial block
- Variables *e* and *f* are evaluated 2 and 4 time units respectively
  After the blocking statements in the first initial block
- Finally expressions k<=1 and *l <= 1* are evaluated 2 and 4 time units
  After the blocking statements in the second initial block
  Statements in one initial block are executed in sequence

```
//  Illustrate Procedural blocking and
//  nonblocking assignment
//  Single initial block
module blockingNonblocking();
//  declare temp registers
    reg a,b,c,d,e,f,g,h,i,j,k,l;

//  initial block – starts t0
//  initialize reg variables
  initial
  begin
    a = 0; b = 0; c = 0; d = 0; e = 0; f = 0;
    g = 0; h = 0; i = 0; j = 0; k = 0; l = 0;
  end

//  initial block – starts t0
  initial
  begin
    //  delay on right hand side
    //  blocking
    a = #10 1;
    b =   #2 1;
    c =   #4 1;

    //  nonblocking – arrive @ t = 16
    d <= #10 1;  //  t = 26
    e <=   #2 1;  //  t = 18
    f <=   #4 1;   //  t = 20
```

```
    //  delay on left hand side – delay before eval
       //  blocking  – arrive @ t = 16
       #10 g = 1;//  t = 26
         #2 h = 1;//  t = 28
         #4 i  = 1;    //  t = 32

       //  nonblocking – arrive @ t = 32
       #10 j <= 1;   //  t = 42
        #2 k <= 1;   //  t = 44
        #4 l  <= 1;   //  t = 48
    end

    initial
    begin
      $display("\ttime, \ta, \tb, \tc, \td, \te, \tf, \tg, \th, \ti, \tj, \tk, \tl");
      $monitor($time, " \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b,
    \t%b, \t%b, \t%b, \t%b, \t%b",a,b,c,d,e,f,g,h,i,j,k,l);
      #50 $finish(1);
    end
endmodule
```

time,	a,	b,	c,	d,	e,	f,	g,	h,	i,	j,	k,	l
0	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
10	1,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
12	1,	1,	0,	0,	0,	0,	0,	0,	0,	0,	0,	0
16	1,	1,	1,	0,	0,	0,	0,	0,	0,	0,	0,	0
18	1,	1,	1,	0,	1,	0,	0,	0,	0,	0,	0,	0
20	1,	1,	1,	0,	1,	1,	0,	0,	0,	0,	0,	0
26	1,	1,	1,	1,	1,	1,	1,	0,	0,	0,	0,	0
28	1,	1,	1,	1,	1,	1,	1,	1,	0,	0,	0,	0
32	1,	1,	1,	1,	1,	1,	1,	1,	1,	0,	0,	0
42	1,	1,	1,	1,	1,	1,	1,	1,	1,	1,	0,	0
44	1,	1,	1,	1,	1,	1,	1,	1,	1,	1,	1,	0
48	1,	1,	1,	1,	1,	1,	1,	1,	1,	1,	1,	1

Major differences between the two implementations

Reflected in the evaluation times for the variables d, e, f, g, h, and i

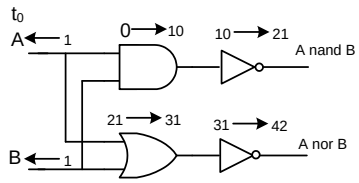
Combinational Logic

The next example implements the earlier NandNor combinational logic circuit

Using a behavioral model

Utilizing both the blocking and non-blocking assignments

Right and left hand side delays




```
module blocking_nonblocking();
   reg a,b, AandB,AorB, AnandB,AnorB;
   reg e,f, EandF,EorF, EnandF,EnorF;

   initial
   begin
      // Blocking Assignment
      a = 1; b = 1;

      //  Delay on the right hand side
      AandB = #10 a&b;              //  t = 10 eval at t = 0
      AnandB = #11 ~AandB;          //  t = 21 eval at t = 10     display at t = 21

      //  Delay on the left hand side
      #10 AorB =  a|b;              //  value appears at t = 21 eval at t = 21     display at t = 31
      #11 AnorB =  ~AorB;           //  value appears at t = 21 eval at t = 31     display at t = 42
   end

// Non blocking Assignment
   initial
   begin
      e = 1; f = 1;

      //  Delay on the right hand side
      EandF   <= #10 e&f;           //  t = 10 eval at t = 0  value unknown
      EnandF <= #11 ~EandF;         //  t = 11 eval at t = 0  value unknown

      //  Delay on the left hand side
      #12 EorF   <= e|f;            //  t = 12 eval at t = 12
      #13 EnorF <= ~EorF;           //  t = 25 eval at t = 25
   end
```

The outputs of the circuits for each of the cases are given

time,	a,	b,	AandB,	AnandB,	AorB,	AnorB,	e,	f,	EandF,	EnandF,	EorF,	EnorF
0	1,	1,	x,	x,	x,	x,	1,	1,	x,	x,	x,	x,
10	1,	1,	1,	x,	x,	x,	1,	1,	1,	x,	x,	x,
12	1,	1,	1,	x,	x,	x,	1,	1,	1,	x,	1,	x,
21	1,	1,	1,	0,	x,	x,	1,	1,	1,	x,	1,	x,
25	1,	1,	1,	0,	x,	x,	1,	1,	1,	x,	1,	0,
31	1,	1,	1,	0,	1,	x,	1,	1,	1,	x,	1,	0,
42	1,	1,	1,	0,	1,	0,	1,	1,	1,	x,	1,	0,

Based upon the order of evaluation of the non-blocking assignment

NAND operand is never assigned a valid value

Tools and Techniques

Before looking a behavioural models of sequential circuitry

Want to look at tools and techniques to

Make development and execution of models simpler

Will look first at those to aid in modeling flow of control

Essential in sequential machines

Flow of Control

The behavioral Verilog model supports

Familiar flow of control constructs

Branches, switches, and loops

Language provides support for event based control

Events

Verilog supports four different types of *event based control*

Given as

- Regular event
- Named event
- OR event
- Level

Each is identified by the event control symbol @

Verilog interprets an *event* as

Change in the value of either a net or a register

Such a change can be used to invoke

Evaluation of either a single statement of a block of statements

Syntax for each is given as follows

Syntax

Regular Event

```
@(signal) action
variable = @( signal) action
signal may be clock, posedge clock, negedge clock for example
```

Named Event

```
event anEvent //  event is a keyword
always @(anEvent) action
```

OR Event

```
always @( signal1 or signal2 or signal3 or…) action
```

Level

```
always wait( signal) action  //  wait is a keyword  can model reset signal
```

Branches

Like the C and C++ languages

Verilog utilizes the *if* and *if else* constructs

Select alternate paths of execution

Based upon the value of a condition variable

Permitted combinations follow the C and C++ syntax

Body of each must be delimited by *begin* - *end*

Syntax

```
if (condition)
   statement;

If (condition)
   statement1;
else
   statement2;

If (condition1)
   statement1;
else If (condition2)
   statement2;
else
   statement 3;
```

If statement comprises a block of statements,
the block must be delimited by the begin-end pair.

Case Statement

The *switch* or *case* statement in Verilog

Uses the Pascal rather than the C language syntax

Unlike the C switch

Once a statement or block of statements is evaluated

Flow of control leaves the case

Do not need *break* statement

Rather than continuing through the remaining alternatives

Syntax

```
case (expression)
   label0:     statement0;
   label1:     statement1;
   .
   .
   labeln-1:      statementn-1;
   default:       defaultStatement;
endcase
```

If statement comprises a block of statements,
the block must be delimited by the begin-end pair

Should always include default statement

Verilog supports two variants on basic *case* or *switch* statement

Syntax

casez – treats all z values in case labels or expressions as don't cares

casex – treats all x and z values in case labels or expressions as don't cares

Loops

Verilog language supports the four common loop constructs.

- while
- repeat
- for
- forever

First three should be familiar

From the C or C++ languages

Forever is unique to Verilog

Useful in embedded systems

Syntax for each is given as

Syntax

```
while(test)
begin
   loop body
end

repeat(repeatcount)
begin
   loop body
end

for(init; test; action)
begin
   loop body
end

init and action are usually assignments.

forever
begin
   loop body
end
```

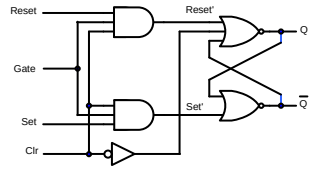
Sequential Logic

As noted earlier behavioural model

Builds design algorithmically

The following three code modules evolve the behavioural implementations of

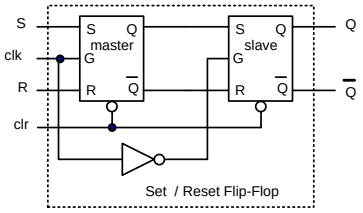
Gated SR latch




```
// Gate Level diagram
// Gated SR Latch with clear
```

```
// Use two SR Latches in a master slave
// configuration to build a flip-flop

module srmsff(q, qnot, s, r, clk, clr);
   input s, r, clk, clr;
   output q, qnot;

   gsrLatch master(qm, qmnot, s, r, clr, clk);
   gsrLatch slave(q, qnot, qm, qmnot, clr, ~clk);

endmodule
```

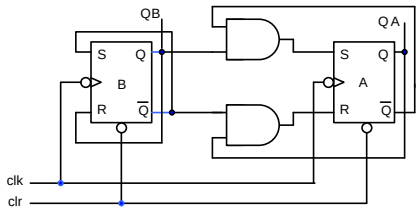
Master-slave SR flip-flop




```
// Use two SR Latches in a master slave
// configuration to build a flip-flop
```

```
// Behavioral Level Model
// Gated SR Latch

module gsrLatch(q, qnot, s, r, clr, enab);
   input s, r,  enab, clr;
   output q, qnot;

   reg q, qnot;

   always@ (~clr or  enab)
   begin
      if(~clr)
      begin
         q = 1'b0;
         qnot = 1'b1;
      end

      else
      begin
         if (s & ~r)
         begin
            q <= s;
            qnot <= r;
         end

         else if (~s & r)
         begin
            q <= s;
            qnot <= r;
         end
       end

  end
endmodule
```

Two bit binary counter




```
// Build a two bit binary up counter using
// master slave SR flip-flops

/*
          tn          tn+1
          A  B        A  B
          0  0        0  1
          0  1        1  0
          1  0        1  1
          1  1        0  0

*/
```

```
// Build a synchronous two bit binary up counter
// using master slave SR flip-flops

module TwoBitCntr(qA, qB, clr, clk);
   input clr, clk;
   output qA, qB;

   reg sA, rA;

   wire qA, qAnot, qB;

   always@(posedge clk)
   begin
      sA = qAnot & qB;
      rA = qA & qB;
   end

   srmsff FFB(qB, qBnot, qBnot, qB, clk, clr);
   srmsff FFA(qA, qAnot, sA, rA, clk, clr);
endmodule
```

The next code module illustrates

More commonly used approach for behavioural modeling of

Counting, timing, or registered types of designs

Rather than working with individual flip-flops

As noted earlier

Design is approached algorithmically

```
//  Build a synchronous two bit binary up counter

module TwoBitCntr(state, clr, clk);
   input clr, clk;
   output[1:0] state;

   reg[1:0] state;

   //  Name the states
   parameter state0 = 2'b00;
   parameter state1 = 2'b01;
   parameter state2 = 2'b10;
   parameter state3 = 2'b11;

   // Build a synchronous two bit binary up counter

   always@(~clr or negedge clk)
   begin
      if(~clr)
      begin
         state = state0;
      end

      else case(state)
         state0:
           state = state1;
         state1:
           state = state2;
         state2:
           state = state3;
         state3:
           state = state0;
      endcase

   end
endmodule
```

Summary

In this lesson we

- ✓ Introduced the Verilog behavioural level model
- ✓ Introduced the behavioural operators
- ✓ Studied the procedural assignment behavioural model
- ✓ Introduced and studied the behavioural flow of control constructs
- ✓ Examined blocking and nonblocking assignments
- ✓ Examined real-world effects under the behavioural model
- ✓ Studied Verilog behavioural level models of combinational and sequential circuits