Session #4

Behavioral Modeling

- In this type of modeling we do not consider the boolean equations or the gate level structure of an given digital circuits.
- We describe the behavior of the given circuit as given in the example.

Behaviour Description: Pseudo – code of Multiplexer

```
if select line is HIGH
   output = A;
else
   output = B;
```

It is difficult to arrive at boolean equations or gate level architecture of a complex digital circuit like a microprocessor.

Hence, we describe the behavior of the circuit and let the computer algorithms to build the circuit for us which can finally be implemented on silicon.

Structured Procedures

- There are two structured procedural blocks in Verilog.
  - `initial`
  - `always`
- All behavioral code can appear only in these blocks
- Unlike C Programming language, activity flows in Verilog run in parallel rather than in sequence.
- Each `always` or `initial` block in Verilog HDL represents a separate activity flow. All activity flows start at '0' time and continues further.
- These blocks cannot be nested.
- As a rule, the execution of procedural block A should not interfere with the execution of procedural block B unless it was the intent of the programmer.
- Procedural assignments (equations) can be written inside these blocks, in this case the keyword assign is not prefixed.

Concurrent assignments should be written outside the structured procedural blocks.

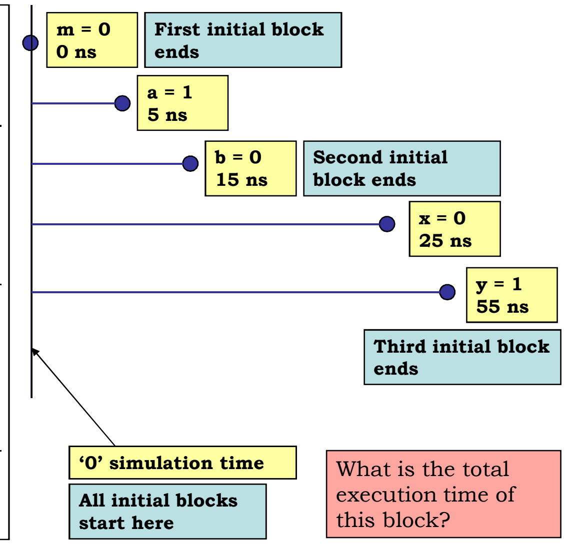
"`initial`" Block

- An `initial` block starts at execution time '0' and executes exactly once during a simulation session and then it does not execute again.
- If there are multiple `initial` blocks, each block starts to execute concurrently at '0' simulation time.
- But, each block finishes execution independently of other blocks.
- Multiple behavioral statements should be grouped using the statements `begin` and `end`.
- Typical usage of `initial` blocks is for initialization, monitoring, waveforms and other processes which should be executed only once during the entire simulation run.

"initial" Block

```
module stimulus;
reg x,y,a,b,m;
initial
        m = 1'b0;
initial
begin
        #5 a = 1'b1;
        #10 b = 1'b0;
end
initial
begin
        #25 x = 1'b0;
        #30 y = 1'b1;
end

endmodule
```



What is the total execution time of this block?

"*always*" Block

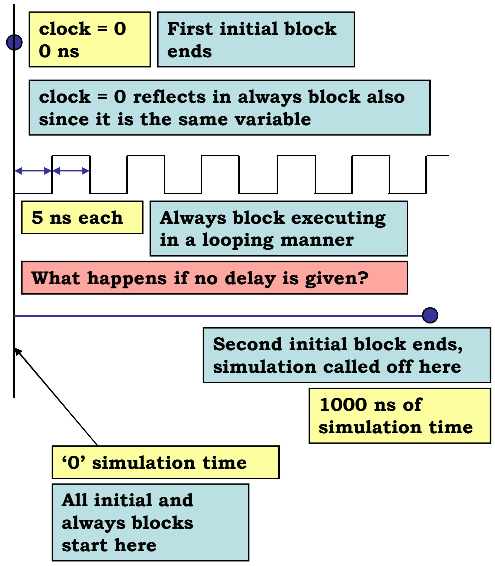
- The ***always*** starts at time '0' and executes the statements in the always block continuously in a looping fashion.
- This statement is used to model a block of activity that is repeated continuously. Example: Any digital circuit.
- The ***always*** statement should have at least one timing control statement (delays or event control), otherwise this may lead to problems in compilation, simulation and sythesis.
- The ***always*** statement can be used to detect and react on specific event (e.g positive edge of clock)

"*always*" Block

```
module clockgen( clock );
output clock;
reg clock;
initial
        clock = 1'b0;
always
begin
        #5; // Delay
        clock = ~clock;
end
initial
        #1000 $finish;

endmodule
```

What happens if ***$finish*** is not there?



Procedural assignments

- Procedural assignments:
  - assign values to registers: ***reg***, ***integer***, ***time***, ***real***, ***realtime*** data types.
  - The left-hand side of an assignment is a register (bit-select, part-select or whole element).
  - The right-hand side - any expression
  - may occur within ***always***, ***initial***, ***task*** and ***functions***.

```
1.
module and_gate(a, b, out);
input a, b;
output out;
reg out;
        always@(a or b)
                out = a & b ;
endmodule

2.
module and_gate(a, b, out);
input a, b;
output out;
//reg out;        -if not reg, net is assumed!
        assign out = a & b ;
        always @ (a or a)
        out = a & b;  //-illegal for net!
endmodule
```

Sensitivity List

- Suppose, an always block should be written in such a way that, if an event on a signal occurs, then it should execute the procedural statements with in it? How can we have such a control?

Sensitivity list is a list of signals generally written along with the ***always*** block.

The ***always*** block waits for some event to occur on any of the signals which are listed and then executes all the procedural statements in the always block.

After execution finishes the control returns to the sensitivity list and the ***always*** block waits for another event to occur on any of the signals in the list.

- On the first place, why after all we should need such a facility?

  Because of the nature of the digital hardware!!!

Any digital circuit responds to a change in the signal and settles down until there is a next event on a signal to which it is sensitive to.

```
module and_gate(a, b, out);
input a, b;
output out;
reg out;
  always @(a or b)
         out = a & b ;
endmodule
```

Sensitivity list

"*always*" Block

- The always block in in itself represents a complete digital entity where the following should be noted:
  - The sensitivity list, parameters used for checking conditions, the RHS part of all procedural assignments act as the inputs to the given always block
  - The LHS of all equations are considered as outputs
  - The sensitivity list represents a set of signals which can cause the outputs to change
  - Outputs depend upon signals in the sensitivity list and also on few other signals that comprise the always block

Best approach to design a digital circuit is to split the functionality into simpler parts. Express each of them in a separate always block. Their connection is established through the use of signal in various always blocks.

Also make sure that a particular signal is not assigned a value (i.e. used as an output) in more than one always block. But why?

Multiple Driver Problem!!!

However, a variable can be used as an input in any number of always blocks.

“ *if* ” Conditional Statement

if (expression) statement;

```
if (reset) q = 0;
```

if (expression) statement;
else statement;

```
if (reset) q = 0;
else q = d;
```

if (expression) statement;
else if (expression) statement;
else statement;

```
if (enable) q = d;
else if (reset) q = 0;
else q = 1'bz;
```

“ *if* “ conditional statement works on a priority basis i.e. the priority of execution of statements depends on the order in which it has been written?

“ *case* ” Conditional Statement

case **(expression)**
option_1 : statement_1;
option_2 : statement_2;
option_3 : statement_3;
. . .
default: default_statement;
endcase

```
case (select)
        2'b00 : out = in[0];
        2'b01 : out = in[1];
        2'b10 : out = in[2];
        2'b11 : out = in[3];
        default: out = 1'bx;
endcase
```

If the “**expression**” equals to any of the options given (option_1, option_2,…), then the statements for that option get executed.

If none of the options are matching then default statements get executed. The default statements are written with “**default**” keyword.

What does this code represent?

Default statement is a must when all possible options for the expression are not specified. Unintended latch logic gets inferred otherwise.

“ *casez* ” Conditional Statement

```
casez (expression)
    option_1 : statement_1;
    option_2 : statement_2;
    option_3 : statement_3;
    . . .
    default: default_statement;
endcase
```

Treats all **z** values in the case alternatives or the case expression as don’t cares.

All bit positions with **z** can also be represented by **?** in that position.

Bit positions with **x** are not treated as don’t cares!!!

```
casez (in)
  4'b1zzz : out = in[0];
  4'b01zz : out = in[1];
  4'b001z : out = in[0];
  4'b0001 : out = in[1];
  default : out = 1'bx;
endcase
```

```
casez (in)
  4'b1??? : out = in[0];
  4'b01?? : out = in[1];
  4'b001? : out = in[0];
  4'b0001 : out = in[1];
  default : out = 1'bx;
endcase
```

The two examples have the same results. Both z values and ? are treated as don’t cares

“ *casex* ” Conditional Statement

```
casex (expression)
    option_1 : statement_1;
    option_2 : statement_2;
    option_3 : statement_3;
    . . .
    default: default_statement;
endcase
```

Treats all ***x*** and ***z*** values in the alternatives or the case expression as don’t cares.

All bit positions with **z** or **x** can also be represented by **?** in that position.

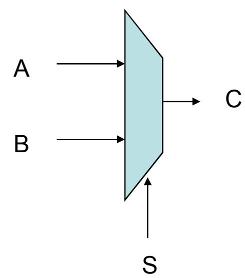
```
casex (in)
  4'b1xxx : out = in[0];
  4'b01zz : out = in[1];
  4'b001x : out = in[0];
  4'b0001 : out = in[1];
  default : out = 1'bx;
endcase
```

```
casex (in)
  4'b1??? : out = in[0];
  4'b01?? : out = in[1];
  4'b001? : out = in[0];
  4'b0001 : out = in[1];
  default : out = 1'bx;
endcase
```

The two examples have the same results. Values z and x and ? are treated as don’t cares

Case Study: Multiplexer

- Behavioural Model




```
module mux ( a,b,s,c );

 input a,b,s;
 output c;

 always@( s )
  if( s )
   c = b;
  else
   c = a;

endmodule
```

What is the case equivalent?

Are the gates which are needed to build a multiplexer a concern in this case?

Is this code correct?

Exercise 1

- Describe a multiplexer using behavioural Verilog description 4:1
- Test the model in the simulator of your choice.

45 Minutes

Exercise 2

- Model a 8 bit priority encoder
- Test the circuit in the simulator of your choice.

45 Minutes

Exercise 3

- Design a BCD to seven segment encoder.
- Test the circuit in the simulator of your choice.

45 Minutes

Exercise 4

- Model a comparator for 4 bit data A and B, which will identify the following conditions. Test the module in the simulator of your choice
  - A > B
  - A < B
  - A = B

45 Minutes