Hardware Description Languages: Verilog

- Syntax, Loops, Behavioral modeling

Initial block

- Starts at time 0, executes exactly once during a simulation
- Multiple initial blocks--> each block starts to execute concurrently at time 0

```
module stimulus;
initial
begin
#5 a = 1'b1;
#25 b = 1'b0;
end
endmodule
```

Simple Behavioral Model: the always block

- always block
  - starts at time 0 and executes continuously in a loop
  - Always waiting for a change to a trigger signal
  - Then executes the body

```
module and_gate (out, in1, in2);
  input   in1, in2;
  output  out;
  reg     out;

  always @(in1 or in2) begin
     out = in1 & in2;
  end
endmodule
```

Not a real register!!
A Verilog register
Needed because of assignment in always block

Specifies when block is executed
I.e., triggered by which signals

`always` Block

- Procedure that describes the function of a circuit
  - Can contain many statements including `if`, `for`, `while`, `case`
  - Statements in the `always` block are executed *sequentially*
    - (Continuous assignments <= are executed in *parallel*)
  - `begin`/`end` used to group statements

Concurrent

- All statements in Verilog are concurrent (executed simultaneously)- unless they are inside a sequential block
- All modules inside a file run concurrently

Sequential

- Initial and always procedural blocks: Initial blocks start execution at time zero and execute only once.
- Always blocks loop to execute over and over again, and as the name suggests, they execute always.
- if, case structures, for and while loops: Always appear inside an always block.

..cont

- All sequential Verilog statements must be inside a always/initial block.
- Sequential statements are placed inside a begin/end block and executed in sequential order within the block.
- blocks themselves are executed concurrently.

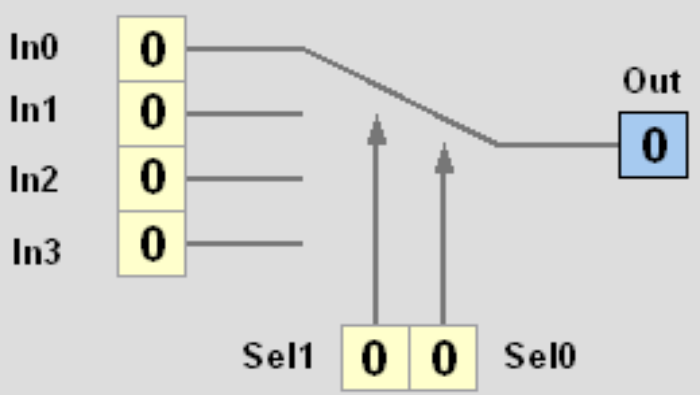
Event control @

```
@(clock) q = d; //q = d is executed whenever signal clock changes value
@(posedge clock) q = d; //q = d is executed whenever signal clock does
                        //a positive transition ( 0 to 1,x or z,
                        // x to 1, z to 1 )
@(negedge clock) q = d; //q = d is executed whenever signal clock does
                        //a negative transition ( 1 to 0,x or z,
                        //x to 0, z to 0)
```

```
always @( reset or clock or d)
                                //Wait for reset or clock or d to
change
```

```
always @(*)
begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end
```

Behavioral style: Mux




```
module mux_4_to_1 (Out,In0,In1,In2,In3,Sel1,Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;
reg Out;

always @(Sel1 or Sel0 or In0 or In1 or In2 or In3)
  begin
    case ({Sel1, Sel0})
    2'b00 : Out = In0;
    2'b01 : Out = In1;
    2'b10 : Out = In2;
    2'b11 : Out = In3;
    default : Out = 1'bx;
    endcase
  end

endmodule
```

Operators

- a = ~ b; // ~ is a unary operator. b is the operand
- a = b && c; // && is a binary operator. b and c are operands
- a = b ? c : d;
  - ?: is a ternary operator. b, c and d are operands

Verilog Operators

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	case equality	equality
!=	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

Verilog `if`

- Same as C if statement

```
// Simple 4:1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;    // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;              // target of assignment

  always @(sel or A or B or C or D)
    if (sel == 2'b00) Y = A;
    else if (sel == 2'b01) Y = B;
    else if (sel == 2'b10) Y = C;
    else if (sel == 2'b11) Y = D;

endmodule
```

..cont

An output must be declared as a reg only if it is assigned using a "procedural assignment"

```
output reg a;
always @*
 a = !b;  //assign a=!b;
```

Verilog case

- Sequential execution of cases
  - Only first case that matches is executed (implicit break)
  - Default case can be used

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;       // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;                 // target of assignment

  always @(sel or A or B or C or D)
    case (sel)
      2'b00: Y = A;
      2'b01: Y = B;
      2'b10: Y = C;
      2'b11: Y = D;
    endcase
endmodule
```

Conditions tested in sequential order

Verilog case

- With default

```
// Simple binary encoder (input is 1-hot)
module encode (A, Y);
input  [7:0] A;           // 8-bit input vector
output [2:0] Y;           // 3-bit encoded output
reg    [2:0] Y;           // target of assignment

  always @(A)
    case (A)
      8'b00000001: Y = 0;
      8'b00000010: Y = 1;
      8'b00000100: Y = 2;
      8'b00001000: Y = 3;
      8'b00010000: Y = 4;
      8'b00100000: Y = 5;
      8'b01000000: Y = 6;
      8'b10000000: Y = 7;
      default:     Y = 3'bXXX;    // Don't care when input is not 1-hot
    endcase
endmodule
```

Verilog for

```
// simple encoder
module encode (A, Y);
input  [7:0] A;           // 8-bit input vector
output [2:0] Y;           // 3-bit encoded output
reg    [2:0] Y;           // target of assignment

integer i;                // Temporary variables for program only
reg [7:0] test;

  always @(A) begin
    test = 8b'00000001;
    Y = 3'bX;
    for (i = 0; i < 8; i = i + 1) begin
       if (A == test) Y = i;
       test = test << 1;
    end
  end
endmodule
```

while

```
initial
begin
        count = 0;

        while (count < 128) //Execute loop till count is 127.
                            //exit at count 128
        begin
                 $display("Count = %d", count);
                count = count + 1;
        end
end
```

Compiler directives - $display

```
//Display the string in quotes
$display("Hello Verilog World");
-- Hello Verilog World
//Display value of current simulation time 230
$display($time);
-- 230
//Display value of 41-bit virtual address 1fe0000001c at time 200
reg [0:40] virtual_addr;
$display("At time %d virtual address is %h", $time, virtual_addr);
-- At time 200 virtual address is 1fe0000001c
```

Compiler directives

- 'define WORD_SIZE 32
- 'include header.v
- `define MEM_SIZE 1024
- $display("The maximum memory size is %h", 'MEM_SIZE);

Future topics

- Functions
- Tasks
- File operations
- User defined Primitives

Summary

- 2 modules
  - Design: Main logic- can contain multiple modules
  - Testbench – no ports, only meant to assign values to inputs, generate waveform
    - Initial values of signals go in inital block
    - Repeating signal assignments go in always block
- Structural verilog
  - Describe circuit structure, connect gates, modules
  - Should not be written in always or initial block
- Behavioral verilog
  - Code inside always block
  - Always block has trigger, usually clock, reset etc
  - If-else, switch, for, while etc

Practice - 2

1. Simulate the dff.v (it contains the testbench too).

- Important: Once you run ./a.out, it will keep running infinitely, because it is in an always block. You need to hit Ctrl +Z to stop it, else, the vcd will become a large file and will never end.
- Note that in the testbench, the clock and d input are being toggled using the following statements:
  - always
  - #3 clk=~clk;
  - always
  - #5 d=~d;
- Add a synchronous reset input to this D-Flip Flop. You will need to change the testbench to simulate it
- Create another Verilog module to make this an asynchronous reset input to this D-Flip Flop. Note the difference between the two types of reset

Practice - 2

2. Simulate the counter.v (it contains the testbench too)

- Important: Once you run ./a.out, it will keep running infinitely, because it is in an always block. You need to hit Ctrl +Z to stop it, else, the vcd will become a large file and will never end.
- This is a 4 bit up-counter.
- Note that, the counter wraps around. Once it reaches “1111”, it goes back to 0000--> you should be able to see why.
- Convert this to an up-down counter. You will need to introduce a signal called- “up_down”. If up_down=1, count up. If up_down=0, count down.

Practice - 2

3. Simulate the counter.v (it contains the testbench too)

- Important: Once you run ./a.out, it will keep running infinitely, because it is in an always block. You need to hit Ctrl +Z to stop it, else, the vcd will become a large file and will never end.
- This is a 4 bit up-counter.
- Note that, the counter wraps around. Once it reaches “1111”, it goes back to 0000--> you should be able to see why.
- Convert this to an up-down counter. You will need to introduce a signal called- “up_down”. If up_down=1, count up. If up_down=0, count down.