Introduction

- Verilog HDL is a Hardware Description Language (HDL)
- HDL is a language used to describe a digital system, for example, a computer or a component of a computer.
- Most popular HDLs are VHDL and Verilog
- For analog systems AHDL
- Mixed-mode systems MAST-HDL (Sabre)
- Verilog programming is similar to C programming
- VHDL programming is similar to PASCAL (some say like Ada) - Is an IEEE standard

Levels of Description

- Switch Level:
  - layout of the wires, resistors and transistors on an IC chip
  - Easiest to synthesize, very difficult to write, not really used
- Gate (Structural) Level:
  - logical gates, flip flops and their interconnection
  - Very easy to synthesize, a text based schematic entry system
- RTL (dataflow) Level
  - The registers and the transfers of vectors of information between registers.
  - Most efficiently synthesizable level
  - Uses the concept of registers with combinational logic
- Behavioral (algorithmic) Level
  - Highest level of abstraction
  - Description of algorithm without hardware implementation details
  - easiest to write and debug, most difficult to synthesize
- We will focus on the RTL and structural level in the lab

Why Use HDL?

- NO OTHER CHOICE
- For large digital systems, gate-level design is dead
- Millions of transistors on a digital chip
- HDL offers the mechanism to describe, test and synthesize such designs
- Impossible to design on a gate or transistor level

- Comments start with a "//" for one line or /* to */ across several lines
- Describe a system by a set of modules (equivalent to functions in C)


```
//A first digital model in Verilog
module simple;
// Simple Register Transfer Level (RTL) example to demo Verilog.
// The register A is incremented by one.  Then first four bits of B is
// set to "not" of the last four bits of A.  C is the "and" reduction
// of the last two bits of A.
//declare registers and flip-flops
reg [0:7] A, B;
reg       C;
// The two "initial"s and "always" will run concurrently
initial begin: stop_at
   // Will stop the execution after 20 simulation units.
   #20 $stop;
end
// These statements done at simulation time 0 (since no #k)
initial begin: Init
   // Initialize the register A.  The other registers have values of "x"
   A = 0;
   // Display a header
   $display("Time   A        B    C");
   // Prints the values anytime a value of A, B or C changes
   $monitor("  %0d %b %b %b", $time, A, B, C);
end
//main_process will loop until simulation is over always begin: main_process
always begin: main_process
   #1 A = A + 1; // #1 means do after one unit of simulation time
   #1 B[0:3] = ~A[4:7];  // ~ is bitwise "not" operator
   #1 C = &A[6:7];      // bitwise "and" reduction of last two bits of A
end
endmodule
```


Explanation

- In module simple, we declared A and B as 8-bit registers and C a 1-bit register or flip-flop. Inside of the module, the one "always" and two "initial" constructs describe three threads of control, i. e., they run at the same time or concurrently. Within the initial construct, statements are executed sequentially much like in C or other traditional imperative programming languages. The always construct is the same as the initial construct except that it loops forever as long as the simulation runs. The notation #1 means to execute the statement after delay of one unit of simulated time. Therefore, the thread of control caused by the first initial construct will delay for 20 time units before calling the system task $stop and stop the simulation. The $display system task allows the designer to print a message much like printf does in the language C. Every time unit that one of the listed variables' value changes, the $monitor system task prints a message. The system function $time returns the current value of simulated time.

Simulation output

```
Time   A          B          C
 0 00000000       xxxxxxxx   x
 1 00000001       xxxxxxxx   x
 2 00000001       1110xxxx   x
 3 00000001       1110xxxx   0
 4 00000010       1110xxxx   0
 5 00000010       1101xxxx   0
 7 00000011       1101xxxx   0
 8 00000011       1100xxxx   0
 9 00000011       1100xxxx   1
10 00000100       1100xxxx   1
11 00000100       1011xxxx   1
12 00000100       1011xxxx   0
13 00000101       1011xxxx   0
14 00000101       1010xxxx   0
16 00000110       1010xxxx   0
17 00000110       1001xxxx   0
19 00000111       1001xxxx   0
Stop at simulation time 20
```

Lexical Conventions

- Keywords, e. g., module, are reserved and in all lower case letters. Verilog is case sensitive
- Spaces are important in that they delimit tokens in the language.
- Numbers are specified in the traditional form of a series of digits with or without a sign but also in the following form:
- <size><base format><number>
  - <size>: number of bits (optional)
  - <base format>: is the single character ' followed by one of the following characters b, d, o and h, which stand for binary, decimal, octal and hex, respectively.
  - <number>: contains digits which are legal for the <base format>

Examples:

- 549 // decimal number
- 'h 8FF // hex number
- 'o765 // octal number
- 4'b11 // 4-bit binary number 0011
- 3'b10x // 3-bit binary, least significant bit unknown
- 5'd3 // 5-bit decimal number
- -4'b11 // 4-bit two's complement of 0011 or 1101

Lexical Conventions

- String: is a sequence of characters enclosed in double quotes. "this is a string"
- Operators (some examples:)
  - Arithmetic: +, - ! ~ * /
  - Shift: << >> Relational: < <= > >= == != === !==
  - Logical && ||.
- Identifier: Equivalent to variable names: Identifiers can be up to 1024 characters.

Program Structure

- A digital system as a set of modules
- Each module has an interface to other module (connectivity)
- GOOD PRACTICE: place one module per file (not a requirement)
- Modules may run concurrently
- Usually one top level module which invokes instances of other modules
- Usually called a stimulus block

MODULES

- represent bits of hardware ranging from simple gates to complete systems, e. g., a microprocessor.
- Can either be specified behaviorally or structurally (or a combination of the two)
- The structure of a module is the following:
  - module <module name> (<port list>);
  - <declares>
  - <module items>
  - endmodule
    - **<module name>**: is an identifier that uniquely names the module.
    - **<port list>** is a list of input, inout and output ports which are used to connect to other modules.
    - **<declares>** section specifies data objects as registers, memories and wires as wells as procedural constructs such as functions and tasks.
    - **<module items>** may be initial constructs, always constructs, continuous assignments or instances of modules.

Behavioral Example: NAND

- Here is a behavior specification of a module NAND
  - // Behavioral Model of a Nand gate
  - // By Dan Hyde, August 9, 1995
  - module NAND(in1, in2, out);
  - input in1, in2;
  - output out;
  - // continuous assign statement
  - assign out = ~(in1 & in2);
  - endmodule

Explanation

- The ports in1, in2 and out are labels on wires. The continuous assignment assign continuously watches for changes to variables in its right hand side and whenever that happens the right hand side is re-evaluated and the result immediately propagated to the left hand side (out).
- The continuous assignment statement is used to model combinational circuits where the outputs change when one wiggles the input.
- Here is a structural specification of a module AND obtained by connecting the output of one NAND to both inputs of another one.

Structural Example: AND

```
module AND(in1, in2, out);
// Structural model of AND gate from two NANDS
 input in1, in2;
 output out;
 wire w1;
            // two instances of the module NAND
 NAND NAND1(in1, in2, w1);
 NAND NAND2(w1, w1, out);
endmodule
```

- This module has two instances of the NAND module called NAND1 and NAND2 connected together by an internal wire w1.

Instance

- The general form to invoke an instance of a module is :

<module name> <parameter list> <instance name> (<port list>);

- <parameter list> are values of parameters passed to the instance.
- An example parameter passed would be the delay for a gate.

Stimulus Block:

- The following module is a high level module which sets some test data and sets up the monitoring of variables.

```
module test_AND;
// High level module to test the two other modules
 reg a, b;
 wire out1, out2;

 initial begin  // Test data
   a = 0;  b = 0;
   #1 a = 1;
   #1 b = 1;
   #1 a = 0;
 end

 initial begin // Set up monitoring
   $monitor("Time=%0d a=%b b=%b out1=%b out2=%b",
          $time, a, b, out1, out2);
 end
          // Instances of modules AND and NAND
 AND  gate1(a, b, out2);
 NAND gate2(a, b, out1);

endmodule
```


- Notice that we need to hold the values a and b over time. Therefore, we had to use 1-bit registers. reg variables store the last value that was procedurally assigned to them (just like variables in traditional imperative programming languages). wires have no storage capacity. They can be continuously driven, e. g., with a continuous assign statement or by the output of a module, or if input wires are left unconnected, they get the special value of x for unknown.
- Continuous assignments use the keyword assign whereas procedural assignments have the form <reg variable> = <expression> where the <reg variable> must be a register or memory. Procedural assignment may only appear in initial and always constructs.
- The statements in the block of the first initial construct will be executed sequentially, some of which are delayed by #1, i. e., one unit of simulated time. The always construct behaves the same as the initial construct except that it loops forever (until the simulation stops). The initial and always constructs are used to model sequential logic (i. e., finite state automata).

Output.

```
Time=0 a=0 b=0 out1=1 out2=0
Time=1 a=1 b=0 out1=1 out2=0
Time=2 a=1 b=1 out1=0 out2=1
Time=3 a=0 b=1 out1=1 out2=0
```

Procedural vs. Continuos Assignments

- Procedural assignment changes the state of a register
  - sequential logic
  - Clock controlled
- Continuous statement is used to model combinational logic.
  - Continuous assignments drive wire variables and are evaluated and updated whenever an input operand changes value. It is important to understand and remember the difference.

Physical Data Types

- modeling registers (reg) and wires (wire).
- register variables store the last value that was procedurally assigned to them
- wire variables represent physical connections between structural entities such as gates
  - does not store anything, only a label on a wire
- The reg and wire data objects may have the following possible values:
  - 0 logical zero or false
  - 1 logical one or true
  - x unknown logical value
  - z high impedance of tristate gate
  - reg variables are initialized to x at the start of the simulation. Any
  - wire variable not connected to something has the x value.

Register Sizes

- Size of a register or wire in the declaration
  - reg [0:7] A, B;
  - wire [0:3] Dataout;
  - reg [7:0] C;
- specify registers A and B to be 8-bit wide with the most significant bit the zeroth bit, whereas the most significant bit of register C is bit seven. The
- wire Dataout is 4 bits wide.

```
initial begin: int1
    A = 8'b01011010;
    B = {A[0:3] | A[4:7], 4'b0000};
end
```

- B is set to the first four bits of A bitwise or-ed with the last four bits of A and then concatenated with 0000. B now holds a value of 11110000. The
- {} brackets means the bits of the two or more arguments separated by commas are concatenated together.

Control Constructs

- Instead of C's { } brackets, Verilog HDL uses begin and end.
- The{ } brackets are used for concatenation of bit strings

```
if (A == 4)
  begin
    B = 2;
  end
else
  begin
    B = 4;
  end
```

```
case (sig)
  1'bz: $display("Signal is floating");
  1'bx: $display("Signal is unknown");
  default: $display("Signal is %b", sig);
endcase
```

```
case (<expression>)
  <value1>: <statement>
  <value2>: <statement>
  default: <statement>
endcase
```


Repetition

- The for statement is very close to C's for statement except that the ++ and
- -- operators do not exist in Verilog. Therefore, we need to use i = i + 1

```
for(i = 0; i < 10; i = i + 1)
 begin
  $display("i= %0d", i);
 end
```

```
i = 0;
while(i < 10)
  begin
    $display("i= %0d", i);
    i = i + 1;
  end
```

```
repeat (5)
 begin
  $display("i= %0d", i);
  i = i + 1;
 end
```


Blocking and Non-blocking Procedural Assignments

- Blocking assignment statement (= operator) acts much like in traditional programming languages.
  - The whole statement is done before control passes on to the next statement.
- Non-blocking (<= operator) evaluates all the right-hand sides for the current time unit and assigns the left-hand sides at the end of the time unit.

```
// testing blocking and non-blocking assignment
module blocking;
reg [0:7] A, B;
initial begin: init1
      A = 3;
      #1 A = A + 1;   // blocking procedural assignment
        B = A + 1;
        $display("Blocking:     A= %b B= %b", A, B );

      A = 3;
      #1 A <= A + 1;  // non-blocking procedural assignment
        B <= A + 1;

      #1 $display("Non-blocking: A= %b B= %b", A, B );
end

endmodule
```


Output

- Blocking: A= 00000100 B= 00000101
- Non-blocking: A= 00000100 B= 00000100

Timing and Delay Control

- If there is no timing control, simulation time does not advance. Simulated time can only progress by one of the following:
- 1. gate or wire delay, if specified.
- 2. a delay control, introduced by the # symbol.
- 3. an event control, introduced by the @ symbol.
- 4. the wait statement.
- The order of execution of events in the same clock time may not be predictable.
- #10 A = A + 1;
  - specifies to delay 10 time units before executing the procedural assignment statement. The # may be followed by an expression with variables.

Events

- The execution of a procedural statement can be triggered with a value change on a wire or register, or the occurrence of a named event. Some examples:

```
@r begin                    // controlled by any value change in
      A = B&C;              // the register r
end

@(posedge clock2) A = B&C;  // controlled by positive edge of clock2

@(negedge clock3) A = B&C;  // controlled by negative edge of clock3

forever @(negedge clock3)   // controlled by negative edge of clock3
begin                       // This has the same effect as the previous statement
      A = B&C;
end
```


Naming events

- Verilog also provides features to name an event and then to trigger the occurrence of that event. We must first declare the event:

  event event6;

  - To trigger the event, we use the -> symbol :
  - -> event6;
  - To control a block of code, we use the @ symbol as shown:
  - @(event6) begin
  - <some procedural code>
  - end

Wait Statement

- The wait statement allows a procedural statement or a block to be delayed until a condition becomes true.
  - wait (A == 3)
  - begin
  - A = B&C;
  - end
- The difference between the behavior of a wait statement and an event is that
  - the wait statement is level sensitive whereas @(posedge clock); is triggered by a signal transition or is edge sensitive.