

CSE 2021 Computer Organization

Appendix Part 2

Verilog Basics

What is an HDL?

- A Hardware Description Language (HDL) is a software programming language used to model the intended operation of a piece of hardware.
- The difference between an HDL and “C”
 - Concurrency
 - Timing
- A powerful feature of the Verilog HDL is that we can use the same language for describing, testing and debugging the system.

An Example

```
module pound_one;
reg [7:0] a,a$b,b,c; // register declarations
reg clk;

initial
begin
    clk=0; // initialize the clock
    c = 1;
    forever #25 clk = !clk;
end
/* This section of code implements
a pipeline */
always @(posedge clk)
begin
    a = b;
    b = c;
end
endmodule
```

Identifiers

- Identifiers are names assigned by the user to Verilog objects such as modules, variables, tasks etc.
- An identifier may contain any sequence of letters, digits, a dollar sign '\$' , and the underscore '_' symbol.
- The first character of an identifier must be a letter or underscore; it cannot be a dollar sign '\$' , for example. We cannot use characters such as '-' (hyphen), brackets, or '#' in Verilog names (**escaped identifiers** are an exception).

Escaped Identifiers

- The use of escaped identifiers allow any character to be used in an identifier.
 - Escaped identifiers start with a backslash (\) and end with white space (White space characters are space, tabs, carriage returns).
 - Gate level netlists generated by EDA tools (like DC) often have escaped identifiers
- Examples:
 - Vclock = 0;
 - \a*b = 0;
 - \5-6
 - \bus_a[0]
 - \bus_a[1]

```

module identifiers; /* Multiline comments in Verilog look like C comments
and // is OK in here. */
// Single-line comment in Verilog.

reg legal_identifier, two__underscores;
reg _OK,OK_,OK_$,OK_123,CASE_SENSITIVE, case_sensitive;
reg Vclock, \a*b ; // Add white_space after escaped identifier.
//reg $_BAD,123_BAD; // Bad names even if we declare them!
initial begin
    legal_identifier = 0; // Embedded underscores are OK,
    two__underscores = 0; // even two underscores in a row.
    _OK = 0; // Identifiers can start with underscore
    OK_ = 0; // and end with underscore.
    OK$ = 0; // $ sign is OK.
    OK_123 = 0; // Embedded digits are OK.
    CASE_SENSITIVE = 0; // Verilog is case-sensitive (unlike VHDL).
    case_sensitive = 1;
    Vclock = 0; // An escaped identifier with \ breaks rules
    \a*b = 0; // but be careful to watch the spaces!
    $display("Variable CASE_SENSITIVE= %d",CASE_SENSITIVE);
    $display("Variable case_sensitive= %d",case_sensitive);
    $display("Variable Vclock = %d",Vclock );
    $display("Variable \\a*b = %d",\a*b );
end
endmodule

```

An Example

Simulation Result of the Example

Variable CASE_SENSITIVE= 0

Variable case_sensitive= 1

Variable /clock = 0

Variable \a*b = 0

Logic values

- Verilog has 4 logic Values:
 - '0' represents zero, low, false, not asserted.
 - '1' represents one, high, true, asserted.
 - 'z' or 'Z' represent a high-impedance value, which is usually treated as an 'x' value.
 - 'x' or 'X' represent an uninitialized or an unknown logic value--an unknown value is either '1' , '0' , 'z' , or a value that is in a state of change.

Data Types

- Three data type classes:
 - Nets
 - Physical connections between devices
 - Example: **wire** a, b;
 - Registers
 - Storage devices, variables.
 - Example: **reg** a; **reg** [7:0] bus;
 - Parameters
 - Constants
 - Example: **parameter** width=32;
parameter A_string =“hello”;

CSE 2021 Computer Organization

Code Structure

Design Entities

Verilog Module Basics

Design Entities

- The **module** is the basic unit of code in the Verilog language.

- Example

```
module holiday_1(sat, sun,weekend);
  input sat, sun;
  output weekend;
  assign weekend = sat | sun;
endmodule
```

Verilog Module

- **Modules contain**
 - **declarations**
 - **functionality**
 - **timing**

```
module name (port_names);
  module port declarations
  data type declarations
  procedural blocks
  continuous assignments
  user defined tasks & functions
  primitive instances
  module instances
  specify blocks
endmodule
```

syntax:

```
module module_name (signal, signal,... signal ) ;
  .      ; //content of module
  .
  ..
  .
endmodule
```

Module Port Declarations

- Scalar (1bit) port declarations:
 - *port_direction port_name, port_name ... ;*
- Vector (Multiple bit) port declarations:
 - *port_direction [port_size] port_name, port_name ... ;*
- *port_direction* : input, inout (bi-directional) or output
- *port_name* : legal identifier
- *port_size* : is a range from [msb:lsb]

```
input a, into_here, george; // scalar ports
input [7:0] in_bus, data;   // vectored ports
output [31:0] out_bus;     // vectored port
inout [maxsize-1:0] a_bus; // parameterized port
```

Module Instances

- A module may be instantiated within another module.
- There may be multiple instances of the same module.

syntax for instantiation:

module_name instance_name (signal, signal,...);

```
module example (a,b,c,d);
input a,b;
output c,d;
. . .
endmodule
```

```
example ex_inst_1(in_1, in_2, w, z);
example ex_inst_2(in_1, in_2, , z); // skip a port
```

Gate-level Primitives

- Verilog has pre-defined primitives that implement basic logic functions.
- Structural modeling with the primitives is similar to schematic level design.

and	nand	or	nor	xor	xnor
buf	not	bufif0	bufif1	notif0	notif1

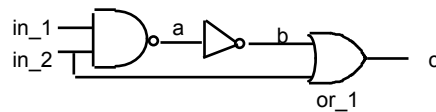
```

module
gate_level_ex(in_1,in_2,c);
output c;
input in_1,in_2;

nand (a, in_1, in_2);
not (b, a);
or or_1(c, in_2, b);

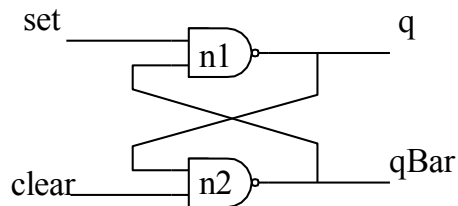
endmodule

```



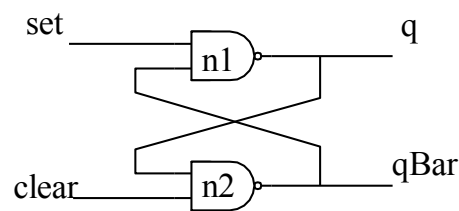
Activity 4

Given the circuit below, develop a Verilog module for the circuit



Activity 4

```
Module simple_latch (q, qBar, set, clear);
  input set, clear;
  output q, qBar;
  nand #2 n1(q,qBar,set);
  nand #2 n2(qBar,q,clear);
endmodule
```



User-Defined Primitives

- We can define primitive gates (a **user-defined primitive** or **UDP**) using a truth-table specification. The first port of a UDP must be an output port, and this must be the only output port (we may not use vector or inout ports).
- An example


```
primitive Adder(Sum, InA, InB);
  output Sum;
  input InA, InB;
  table // inputs : output
    00 : 0;
    01 : 1;
    10 : 1;
    11 : 0;
  endtable
endprimitive
```

Operators

■ Verilog operators (in increasing order of precedence)

- ?: (conditional)
- || (logical or)
- && (logical and)
- | (bitwise or)
- ~| (bitwise nor)
- ^ (bitwise xor)
- ^~ ~^ (bitwise xnor, equivalence)
- & (bitwise and)
- ~& (bitwise nand)
- == (logical) != (logical) === (case) !== (case)
- < (lt)
- <= (lt or equal)
- > (gt)
- >= (gt or equal)
- << (shift left)
- >> (shift right)
- + (addition)
- - (subtraction)
- * (multiply)
- / (divide)
- % (modulus)

CSE 2021 Computer Organization

Procedural Assignment
Continuous Assignment
Control Statement

Procedures

- A Verilog **procedure** is an **always** or **initial** statement, a task , or a function .
- The statements within a sequential block (statements that appear between a **begin** and an **end**) that is part of a procedure execute sequentially in the order in which they appear, but the procedure executes concurrently with other procedures.

Procedural Blocks

- There are two types of procedural blocks:
 - initial blocks - executes only once
 - always blocks - executes in a loop
- Multiple Procedural blocks may be used, if so the multiple blocks are concurrent.
- Procedural blocks may have:
 - Timing controls - which delays when a statement may be executed
 - Procedural assignments
 - Programming statements

Procedural Statement Groups

- When there is more than one statement within a procedural block the statements must be grouped.
- Sequential grouping: statements are enclosed within the keywords **begin** and **end**.
- An example

```
always
begin
    a = 5;          // executed 1st
    c = 4;          // executed 2nd
    wake_up = 1;    // executed 3rd
end
```

Timing Controls (procedural delays)

- **#delay** - simple delay
 - Delays execution for a specific number of time steps.
 - **#5** reg_a = reg_b;
- **@ (edge signal)** - edge-triggered timing control
 - Delays execution until a transition on **signal** occurs.
 - **edge** is optional and can be specified as either **posedge** or **negedge**.
 - Several **signal** arguments can be specified using the keyword **or**.
 - An example : always @ (posedge clk) reg_a = reg_b;
- **wait (expression)** - level-sensitive timing control
 - Delays execution until **expression** evaluates true.
 - **wait (cond_is_true)** reg_a = reg_b;

Procedural assignments

- Assignments made within procedural blocks are called procedural assignments.
 - Value of the RHS of the equal sign is transferred to the LHS
 - LHS must be a register data type (reg, integer, real). NO NETS!
 - RHS may be any valid expression or signal

```
always @ (posedge clk)
begin
    a = 5;           // procedural assignment
    c = 4*32/6;      // procedural assignment
    wake_up = $time; // procedural assignment
end
```

Continuous Assignment

- Continuous assignment assigns a value to a **wire** in a similar way that a real logic gate drives a real wire.
- The main use for continuous assignments is to model combinatorial logic.

syntax: Explicit continuous assignment:
assign net_name = expression;
 where **net_name** is a **net** that has been previously declared

```
module continuous (Ain, Aout);
    input Ain;
    output Aout;
    assign Aout = ~Ain //continuous assignment.
endmodule
```

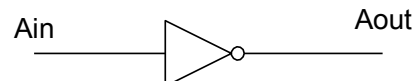


Illustration of Assignment Statements

module assignments

//... Continuous assignments go here.

always // beginning of a procedure

begin // beginning of sequential block

//... Procedural assignments go here.

end

endmodule

Control Statements

- Two types of programming statements:
 - Conditional
 - Looping
- Programming statements only used in procedural blocks

if and if-else

syntax:

if(expression) statement

If the expression evaluates to true then execute the statement

if(expression) statement1

else statement2

If the expression evaluates to true then execute statement1,
if false, then execute statement2.

```
module if_ex(clk);
  input clk;
  reg red,blue,pink,yellow,orange,color,green;
  always @ (posedge clk)
  if (red || (blue && pink))
    begin
      $display ("color is mixed up");
      color <= 0; // reset the color
    end
  else if (blue && yellow)
    $display ("color is greenish");
  else if (yellow && (green || orange))
    $display ("not sure what color is");
  else $display ("color is black");
endmodule
```

for

syntax:

for (assignment_init; expression; assignment)
statement or statement_group

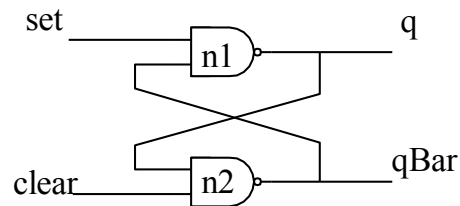
- The ***assignment_init*** is executed once at the start of the loop.
- Loop executes as long as ***expression*** is true.
- The ***assignment*** is executed at the completion of each loop.

```
module for_ex1 (clk);
  input clk;
  reg [31:0] mem [0:9]; // 10x32 memory
  integer i;
  always @ (posedge clk)
    for (i = 9; i >= 0; i = i-1)
      mem[i] = 0; // init the memory to zeros
endmodule
```

Simulating the Verilog Code

- Verilog code of NAND Latch

```
Module simple_latch (q, qBar, set, clear);  
  input set, clear;  
  output q, qBar;  
  nand #2 n1(q,qBar,set);  
  nand #2 n2(qBar,q,clear);  
endmodule
```



Testbench

- A testbench generates a sequence of input values (we call these **input vectors**) that test or **exercise** the verilog code.
- It provides stimulus to the statement that will monitor the changes in their outputs.
- Testbenches do not have a port declaration but must have an instantiation of the circuit to be tested.

A testbench for NAND Latch

```
Module test_simple_latch;
  wire q, qBar;
  reg set, clear;
  simple_latch SL1(q,qBar,set,clear);
  initial
    begin
      #10 set = 0; clear = 1;
      #10 set = 1;
      #10 clear = 0;
      #10 clear = 1;
      #10 $stop;
      #10 $finish;
    end
  initial
    begin
      $monitor ("%d set= %b clear= %b q=%b qBar=%b", $time,
        set, clear, q, qBar);
    end
endmodule
```