Verilog language

Describe a system by a set of modules (equivalent to functions in C)

Keywords, e. g., module, are reserved and in all lower case letters

- Verilog is case sensitive

Operators (some examples)

- Arithmetic: +, - ! ~ * /
- Binary operators: &, |, ^, ~, !
- Shift: << >> Relational: <, <=, >, >=, ==, !=
- Logical: &&, ||

Identifiers

- Equivalent to variable names
- Identifiers can be up to 1024 characters.

Comments start with a "//" for one line or /* to */ across several lines

Number representation

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 number with least significant bit unknown
5'd3      // 5-bit decimal number
-4'b11    // 4-bit two's complement of 0011, or equivalently 1101
```

Data types

Variables of type wires (wire) and registers (reg).

NOTE: A variable of type register does not necessarily represent a physical register

Register variables store the last value that was procedurally assigned

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 tri-state gate

- "reg" variables are initialized to 0 at the start of the simulation.
- "wire" variable not connected to something has the x value.

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 run concurrently

Usually there is a top level module which invokes instances of other modules

Module

Represent bits of hardware ranging from simple gates to complete systems, e. g., a microprocessor

Specified behaviorally, RTL, or structurally

The structure of a module is the following:

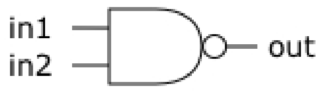
```
module <module name> (<port list>);
<declarations>
<module items>
endmodule
```

<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.
<declarations>	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

example: NAND gate

Here is an RTL specification of a module NAND

```
// Behavioral model of a NAND gate

module NAND(in1, in2, out);
input in1, in2;
output out;

  // continuous assignment statement
  assign out = ~(in1 & in2);

endmodule
```



Default: All undeclared variables are wires and are one bit wide!

GOOD PRACTICE: Declare all variables

Explanation of NAND module

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.

```
// Behavioral model of a NAND gate
module NAND(in1, in2, out);
input in1, in2;
output out;

  // continuous assign statement
  assign out = ~(in1 & in2);

endmodule
```

Instance of a module

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

<instance name> identifies the specific instance of the module

An example parameter passed would be the delay for a gate

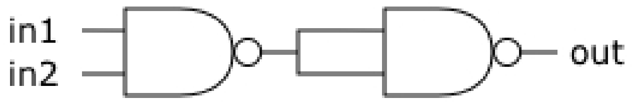
We will not use parameter list in this course!

- For our purposes, to invoke an instance of a module

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

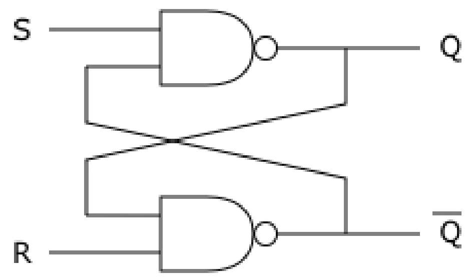
Structural example: AND gate

```
//Structural model of AND gate from two NANDS
module AND(in1, in2, out);
input in1, in2;
output out;

wire w1;

    // two instances of the module NAND
    NAND NAND1(in1, in2, w1);
    NAND NAND2(w1, w1, out);

endmodule
```

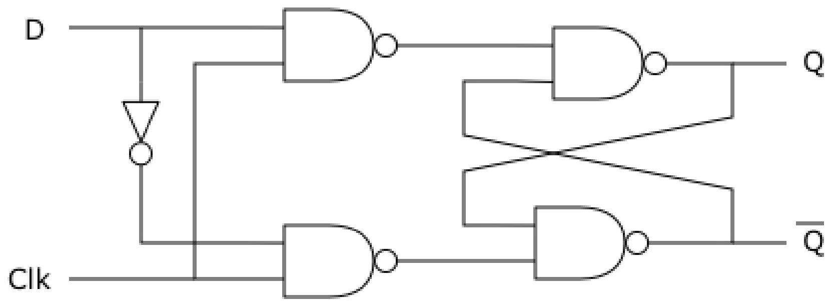


- This module has two instances of the NAND module called NAND1 and NAND2 connected together by an internal wire w1.

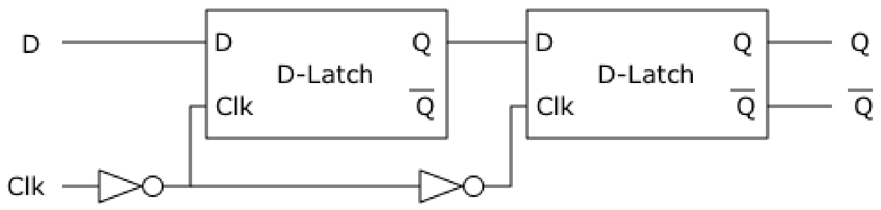
More structural examples



```
module SRLatch(S, R, Q, Qbar);
input S, R;
output Q, Qbar;
    NAND nand1(S, Qbar, Q);
    NAND nand2(R, Q, Qbar);
endmodule
```



```
module DLatch(Clk, D, Q, Qbar);
input Clk, D;
output Q, Qbar;
wire S, R;
    NAND nand1(D, Clk, S);
    NAND nand2(~D, Clk, R);
    SRLatch srlatch1(S, R, Q, Qbar);
endmodule
```



```
module DFlipFlop(Clk, D, Q, Qbar);
input Clk, D;
output Q, Qbar;
wire Qint, Qbarint;
    DLatch dlatch1(~Clk, D, Qint, Qbarint);
    DLatch dlatch2(Clk, Qint, Q, Qbar);
endmodule
```

Continuous vs. procedural assignments

Continuous statement is used to model combinational logic

- Continuous assignments drive wire variables
- Evaluated and updated whenever an input operand changes value

Procedural assignment changes the state of a register

- Used for both combinational and sequential logic
- All procedural statements must be within "always" block

- Example

```
reg A;

always @ (B or C) begin
  A = B & C;
end
```

This is combinational logic

Events

The execution of a procedural statement is triggered by:

- A value change on a wire
- The occurrence of a named event

```
always @ (B or C) begin  // controlled by any value change in B or C
        X = B & C;
end

always @(posedge Clk) Y <= B&C;   // controlled by positive edge of Clk

always @(negedge Clk) Z <= B&C;   // controlled by negative edge of Clk
```

Model of a D-Flip flop

What is the behavior of a D-flipflop ?

- During every positive clock edge, the input is transferred to the output

RTL model

```
module Dflipflop(D, Clk, Q, Qbar);
input D, Clk;
output Q, Qbar;

reg Qint;

    // always is a procedural construct
    // any assignment may be made only to registers
    always @(posedge Clk) Qint <= D;

assign Q = Qint;
assign Qbar = ~Qint;

endmodule
```

Register sizes and assignments

Size of a register or wire in the declaration

```
reg   [0:7] A, B;       // A and B are 8-bit wide with most significant bit as 0th bit
wire  [0:3] Dataout;    // Dataout is a 4-bit wide register
reg   [7:0] C;          // C is a 8-bit register with most significant bit as the 8th bit
                        // This will be the convention adopted in this course!
```

Assignments and concatenations

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

- 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.
- {} brackets means the bits of the two or more arguments separated by commas are concatenated together.

Control constructs

Two control constructs are available:

```
if (A == 4)
        begin
            B = 2;
        end
else if (A == 2)
        begin
            B = 1;
        end
else
        begin
            B = 4;
        end
```

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

Control statement examples

- **1-bit 2-to-1 multiplexer**

```
module mux1bit2to1(a, b, s, out);
input a, b, s;
output out;

     assign out = (~s & a) | (s & b);

endmodule
```

- **Another way to describe**

```
module mux1bit2to1(a, b, s, out);
input a, b, s;
output out;
reg out; // used in procedural statement

always @ (s or a or b)
  if (s == 0) out = a;
  else out = b;

endmodule
```

- **8-bit 4-to-1 multiplexer**

```
module mux8bit4to1(a, b, c, d, s, out);
input [7:0] a, b, c, d;
input [1:0] s;
output  [7:0] out;
reg [7:0] out;
// used in procedural statement

always @ (s or a or b or c or d)
  case (s)
     2'b 00:  out = a;
     2'b 01:  out = b;
     2'b 10:  out = c;
     2'b 11:  out = d;
  endcase

endmodule
```

Blocking/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.

Example: During every clock cycle

- A is ahead of C by 1
- B is same as D

```
// testing blocking and non-blocking
// assignment
module blocking(Clk, A, B);
input Clk;
output [7:0] A, B;

reg [7:0] A, B;
// as these will be used in
// procedural blocks

reg [7:0] C, D;  // two internal registers

    always @(posedge Clk) begin
            // blocking procedural
            // assignment
            C = C + 1;
            A = C + 1;

            // non-blocking procedural
            // assignment
            D <= D + 1;
            B <= D + 1;
    end
endmodule
```

Some tips

Declare ALL variables

- An undeclared variable is treated as a wire!

Declare one variable (especially input/output) per line

- Provide comments for each of those variables
- It will be helpful when you design complex modules

All modules must have the port list defined in the homework/projects

- Even if your solution doesn’t work, you can use the modules we provide

ECE 369

Fundamentals of Computer Architecture

Verilog Programming

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

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 Integrated Circuit (IC) chip

Gate (structural) Level

- Logical gates, flip flops and their interconnection

RTL (dataflow) Level

- The registers and the transfers of vectors of information between registers

Behavioral (algorithmic) Level

- Highest level of abstraction
- Description of algorithm without hardware implementation details (like C programming)

Levels of description

Behavioral level

- Easiest to write and debug, not synthesizable

Register Transfer Level

- synthesizable
- Uses the concept of registers (a set of flipflops) with combinational logic between them

Structural level

- Very easy to synthesize
- A text based schematic entry system

Why Use HDL?

NO OTHER CHOICE:

For large digital systems, gate-level design is unmanageable

Millions of transistors on a digital chip

HDL offers the mechanism to describe, test and synthesize large designs