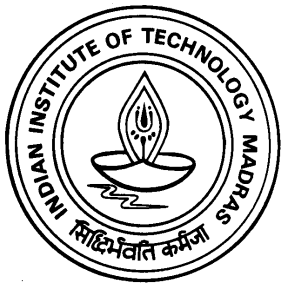
EE2001 - Digital systems lab

Vinita Vasudevan



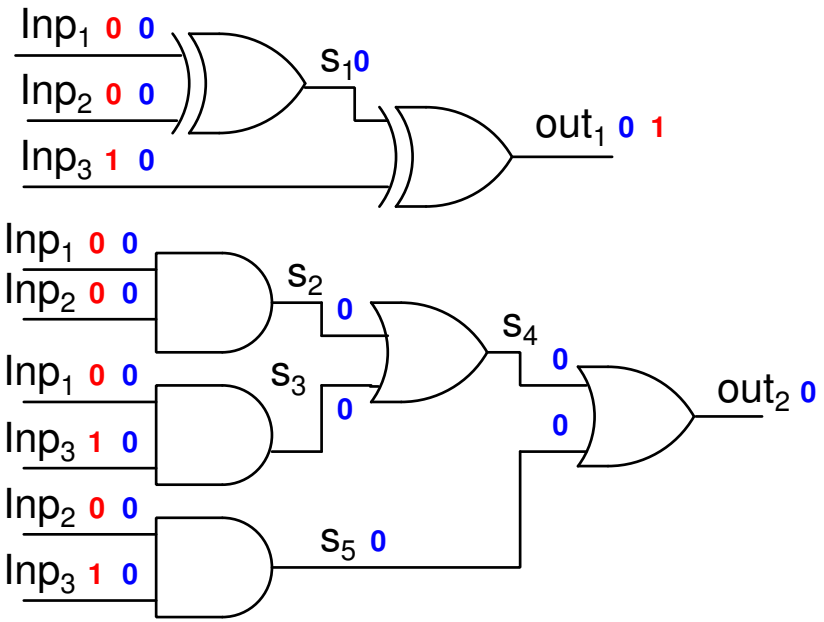
Basic shell commands

Login and open a terminal. The default shell in Linux is Bash (Bourne again shell).

- After you login, you are in your home directory (/home/rollnumber). Check using pwd (environment variable that will give you the current working directory)
- Create a directory called verilog (mkdir verilog) and change to that directory (cd verilog). Figure out what cd .. does.
- List files in a directory (ls or ls -la)
- Get help about a command - man command-name. (man ls will show you the manual pages for ls)
- Set permissions for files - read man pages for chmod.

Logic Simulation: Limitations of using C

- Forced to use unsigned char/short int as the data type even though we are interested only in the last bit.
  It is inefficient in terms of memory
- In a conventional programming language, statements are executed sequentially. The order in which the statements are executed matters.
  Cannot call functions in arbitrary order. Functions whose outputs depend only on primary inputs must be evaluated first (level 1 gates). This must be followed by functions that depend on primary inputs and level 1 outputs and so on.
  Functions must be evaluated in topological order



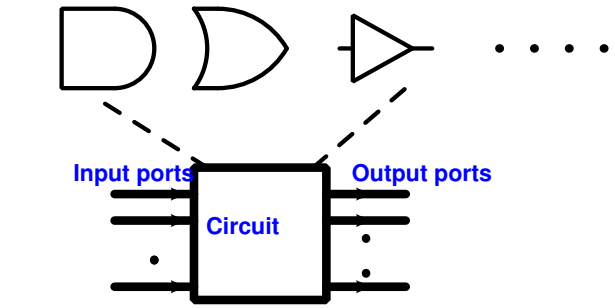

```
while (fscanf(fin,"%hu...", ...)!=EOF)
{
  XOR(&s1, Inp1, Inp2);
  XOR(&out1, s1, Inp3);
  AND(&s2, Inp1, Inp2);
  AND(&s3, Inp1, Inp2);
  AND(&s5, Inp1, Inp2);
  OR(&s4, s2, s3);
  OR(&out2, s4, s5);

}
```

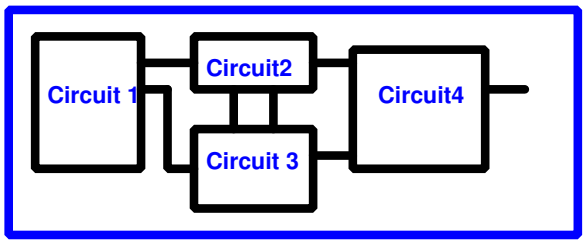
The functions are called and executed even if their inputs do not change. Inefficient in terms of simulation time

What kind of a simulator do we want?

- Declare variables in terms of number of bits required for representation
- A gate must evaluate only when one or more of its inputs change - Event driven simulator
- The order in which functions calling the gates are typed in should not matter. Inherently, the digital system operates in "parallel". Gates evaluate independently when there is a change in their inputs. We need a simulator that mimics this behaviour.
- Digital systems are large: Need to be able to describe behaviour hierarchically.
- Should be versatile - need to describe a variety of components: Memory, buses, CPU, logic circuits ....

Introduction to a Hardware Description Language (Verilog)





- Uses modules (functions) and interfaces to the modules called ports (arguments).
- Basic gates are predefined (both as gates and bit operators)
- Hierarchical modelling
- Mixed modelling - Separate behaviour from implementation (eg: sum = a + b; '+' indicates addition)

Verilog data, modules and Ports

Data Values:

- 0, 1, X (unknown), Z (tristate)
- a[3:0] $\Longrightarrow$ a[0] is the LSB and a[3] is the MSB

```
module Circuit(out1, out2, Inp1, Inp2);

input Inp1, Inp2;
output out1, out2;

  HDL Modelling of functionality

endmodule
```

```
module A(a, b, sum, cout);

input[3:0] a, b;
output[3:0] sum;output cout;

 HDL Modelling of functionality

endmodule
```

Half Adder - Gate level and Data flow model

a	b	sum	cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- Gates and, or, not, xor, nand, nor, xnor predefined
- Bit operators similar to C
- The inputs a and b are in the "sensitivity list". If there is any change in a or b, the corresponding output is re-evaluated.

Gate level Model

```
//Half adder module

module ha(a, b, sum, cout);

input a, b;
output sum, cout;

xor x1(sum, a, b);
and a1(cout, a, b);

endmodule
```

Data flow model of half adder:

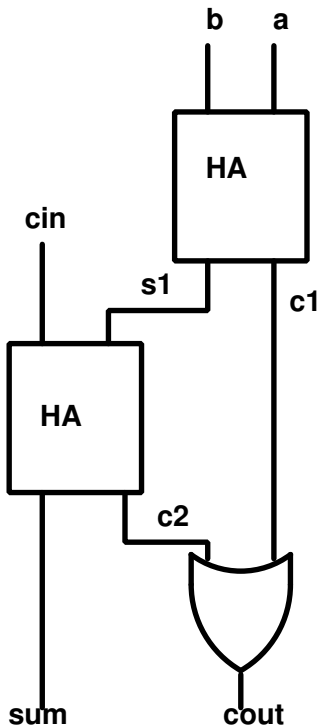
```
//Half adder module

module ha(a, b, sum, cout);

input a, b;
output sum, cout;

assign sum = a ^ b;
assign cout= a & b;

endmodule
```

Hierarchical Modelling

a	b	cin	sum	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$sum(a, b, cin) = \Sigma(1, 2, 4, 7)$$
$$= a \oplus b \oplus cin$$
$$cout(a, b, cin) = \Sigma(3, 5, 6, 7)$$
$$= a\ b + b\ cin + a\ cin$$

Use two half adders to build a full adder



Hierarchical modelling

A module can contain other modules through module instantiation.

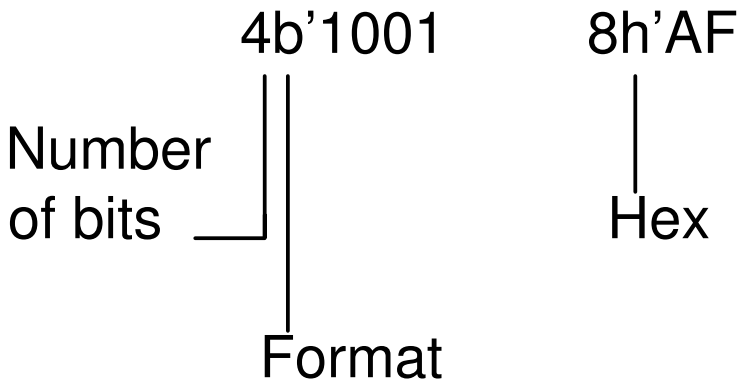
- Modules are connected together using nets
- Ports are attached to nets either by position or name

```
module FA(ain, bin, carryin, sum, cout);

input ain, bin, carryin;
output sum, cout;
wire s1, c1, c2;  These are nets connecting the half adders and OR gates

 ha ha1(.a(ain), .b(bin), .cout(c1), .sum(s1));// Bind to signals by name
ha ha1(ain, bin, s1, c1);  // By position
ha ha2( s1, carryin, sum, c2);
or or1( cout, c1, c2);
// assign cout = c1 | c2;
endmodule
```

Can interchange the order of the statements without affecting the output

Testbench

- The testbench is also a module
- Data type called “reg”. Can assign values to variables that are declared as reg.
- Data format



- Declared as
  reg[3:0] a;
  a = 4’b1010;

```
    //Half adder test bench

module test_ha;           //test bench module
wire s, c;                 //outputs from half adder
reg a, b;                 //inputs to the half adder module

ha DUT(a, b, s, c);         //instantiate the device under test (half adder)

initial
begin
$monitor("At t=%t, a=%b, b=%b, s = %b, c=%b", $time, a, b, s, c);
a = 1;  b = 1;        // t = 0
#10 a = 1;  b = 0;   //change values of a and b after 10 units of time
#10 a = 0;  b = 1;   // t = 20
#10 a = 0;  a = 0;

#10 $finish;
end
endmodule
```

Experiment 2: Logic Simulation using Verilog

Objective:Model circuits using Gate-level and data flow technique. Create test benches to test the circuit.

- Repeat the same experiment that you did in experiment-1 but now in Verilog using
- Use the same circuit you picked last time.
  a. Gate-level modelling
  b. Data-flow modelling

  You should have two different files describing the circuit using various approaches and one file containing the testbench.
- Show that the waveform of the circuit from the structural/ hierarchical model is identical to that of the data flow model for all 2^N input combinations
- Create an error signal called *ERROR* in the testbench that goes HIGH if there is a mismatch between the data flow model and the structural model.

BACK UP SLIDES

Experiment 2: Logic Simulation using Verilog

Objective:Model circuits using Gate-level and data flow technique. Create test benches to test the circuit.

- Build a full adder using
  a. Gate-level modelling
  b. Data-flow modelling

  You should have two different files descibing the circuit using various approaches and one file containing the testbench. Do not use half adders to build the full adder.
- Use the full adder to build a five bit adder and do a logic simulation

- A and B are 4 bit twos complement numbers (ranging from -8 to +7). The range of A+B is -16 to 14, which requires 5 bits. Design a circuit that has a select input S so that if S = 1, the output is A+B (addition), else the output is A-B. Before doing the addition, remember to do a sign extension of the inputs. Use the 5 bit adder you built in the previous question.
- Learn to use concatenate and conditional assign.
- The testbench and circuits should be in different files. The full adder and five bit adder can be in a single file. The adder/subtractor must be in a different file.

Iverilog and gtkwave

- to "compile" use the command
  iverilog file1.v file2.v -o circuit
- To run
  vvp circuit
- To check waveforms use the package gtkwave. Need to add the following statements in the testbench.

```
$dumpfile("test_circuit.vcd");
$dumpvars(0,test_circuit); //Name of testbench module
```

  Here the module for the testbench is test_circuit.

- To get the units of time use

```
`timescale 1ns/100ps  //Before module declaration

$timeformat(-9,1,"ns",5); //Before monitor statement
$monitor( "%t  ", $time, "ain=%b, bin=%b, cin=%b, sum=%b, carry=%b", ain, bi
```