Slide Set 7

Verilog

Steve Wilton
Dept. of ECE
University of British Columbia
stevew@ece.ubc.ca

Hardware Description Languages

- Need a description level up from logic gates.
- Work at the level of functional blocks, not logic gates
  - Complexity of the functional blocks is up to the designer
  - A functional unit could be an ALU, or could be a microprocessor
- The description consists of function blocks and their interconnections
  - Need some description for each function block (not predefined)
  - Need to support hierarchical description (function block nesting)
- To make sure the specification is correct, make it executable.
  - Run the functional specification and check what it does

Hardware Description Languages

There are many different languages for modeling and simulating hardware.

- **Verilog**
- **VHDL**
- M-language (Mentor)
- AHDL (Altera)
- SystemC
- Aida (IBM / HaL)
- …. and many others

The two most common languages are Verilog and VHDL.

- For this class, we will be using Verilog-XL
- Only because you already know VHDL, and it never hurts to be bilingual!

Verilog from 20,000 feet

Verilog Descriptions look like software programs:

- Block structure is a key principle
- Use hierarchy/modularity to manage complexity

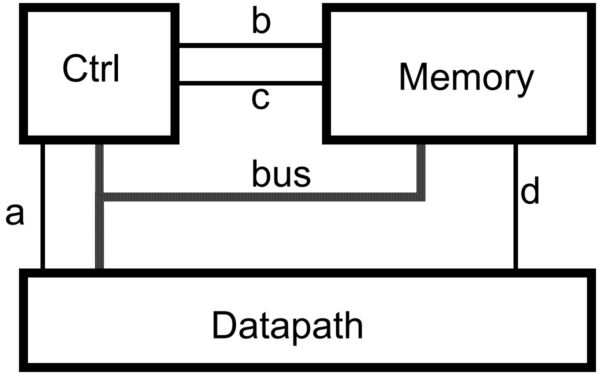
C / Pascal	Verilog
Procedures/Functions	Modules
Procedure parameters	Ports
Variables	Wires / Regs

But they aren't 'normal' programs

- Module evaluation is concurrent. (Every block has its own "program counter")
- Modules are really communicating blocks
- Hardware-oriented descriptions and testing process

Verilog (or any HDL) View of the World

A design consists of a set of communicating modules



- There are graphical user inferfaces for Verilog, but we will not use them
- Instead we will use the text method. Label the wires, and use them as ‘parameters’ in the module calls.

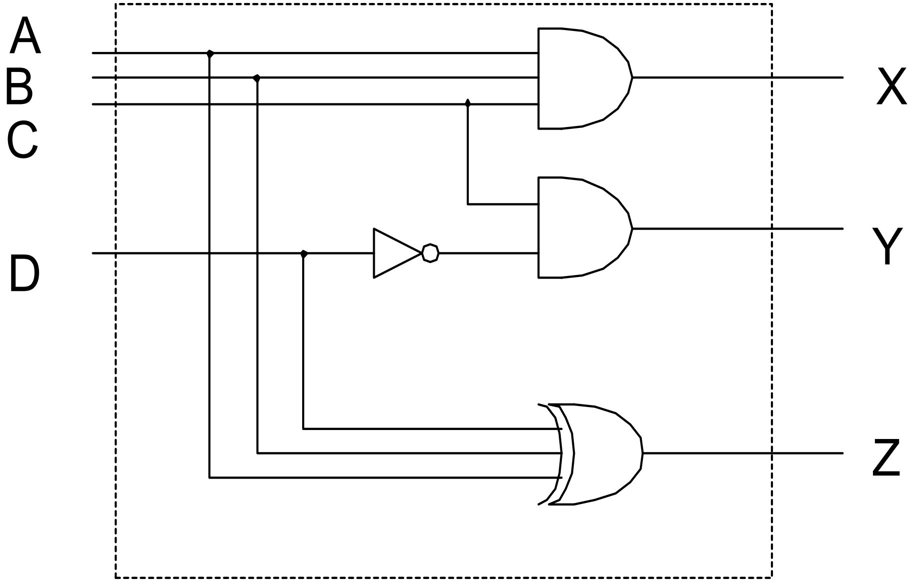
Simple Gate:

```
module xor( A, B, C);
   input A, B;
   output C;

   assign C = (A ^ B);
endmodule;
```

Operation	Operator
~	Bitwise NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR

Can describe a block with several outputs:

```
 module my_gate(A, B, C, D, X, Y, Z);
     input A, B, C, D;
     output  X, Y, Z;

     assign X = A & B & C;
     assign Y = C & ~ D;
     assign Z = A ^ B ^ D;
endmodule;
```



Note: Three assignments performed concurrently

A one bit multiplexer can be described this way:

```
module my_gate(IN1, IN2, SEL, Z);
    input IN1, IN2, SEL;
    output  Z;

     // This is a comment, by the way

    assign Z = (SEL == 1’b0)  ?  IN1 : IN2;
endmodule;
```

Condition (note: 1’b0 is what you would call ‘0’ in VHDL

If condition is true, assign IN1, otherwise, assign IN2

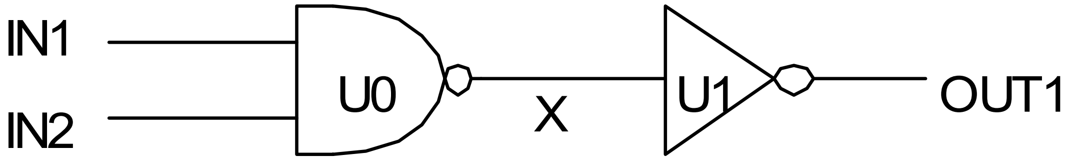
A Four Bit-Multiplexor:

```
 module my_gate(IN1, IN2, SEL, Z);
    input [3:0] IN1, IN2;
    input SEL;
    output [3:0]  Z;

    assign Z = (SEL == 1’b0)  ?  IN1 : IN2;
endmodule;
```

Inputs and outputs are four bit buses

This is written the same as before

Structural Descriptions:



```
module my_gate(IN1, IN2, OUT1);
    input IN1, IN2;
    output OUT1;

    wire X;

    AND_G U0 (IN1, IN2, X);
    NOT_G U1 (X, OUT1);
endmodule;
```

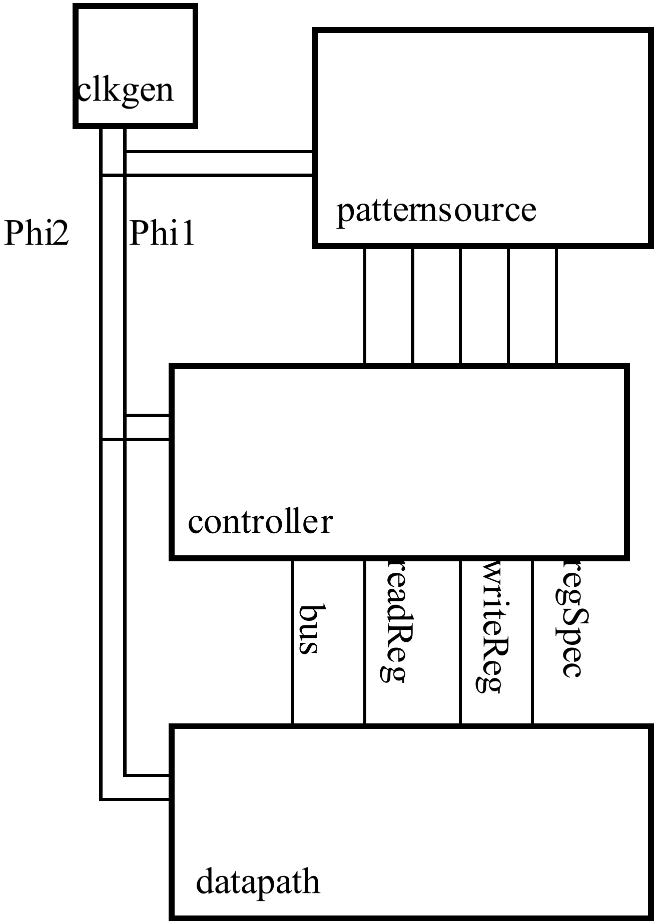
Internal Signal

Submodule name
(defined elsewhere)

Instance Name

Bigger Structural Example

```
module system;
      wire [7:0] bus_v1, const_s1;
      wire [2:0] regSpec_s1, regSpecA_s1, regSpecB_s1;
      wire [1:0] opcode_s1;
      wire Phi1, Phi2, writeReg_s1,
                    ReadReg_s1,nextVector_s1
      clkgen clkgen(Phi1, Phi2);
      datapath datapath(Phi1, Phi2, regSpec_s1, bus_v1,
                  writeReg_s1, readReg_s1);
      controller controller1(Phi1, Phi2, regSpec_s1, bus_v1,
                     const_s1, writeReg_s1, readReg_s1,
                     opcode_s1, regSpecA_s1, regSpecB_s1,
                     nextVector_s1);
      patternsource patternsource(Phi1, Phi2,nextVector_s1,
                     opcode_s1, regSpecA_s1, regSpecB_s1,
                     const_s1);
```



Suppose we have defined:

```
wire [3:0]  S;   // a four bit bus
wire C;         // a one bit signal
```

Then, the expression

```
{ C, S }
```

Is a 5 bit bus:

```
 C S[3] S[2] S[1] S[0]
```

Behavioural Description of an Adder:

```
module adder4( A, B, C0, S, C4);
   input [3:0]  A, B;
   input C0;
   output [3:0]  S;
   output C4;

   assign { C4, S } = A + B + C0;
endmodule;
```

Behavioural Description of an Flip-Flop:

```
module dff_v (CLK, RESET, D, Q);
   input CLK, RESET, D;
   output Q;
   reg Q;

always @(posedge CLK or posedge RESET)
begin
     if (RESET == 1)
          Q <= 0;
    else
          Q <= D;
end
endmodule;
```

What does this mean?

Like a process in VHDL

Equivalent to a "sensitivity list"

REG vs WIRE signals:

- As in VHDL, a process may or may not set the value of each output (for example, in the DFF, Q is not set if CLK is not rising). This implies that some sort of storage is needed for outputs of a always block. Therefore, outputs of an always block must be declared as REG.

- Note: this does not mean a register will actually be used. You can declare purely combinational blocks, where no register is to be used. But, you still must declare the outputs of the always block as REG.

- Rule: All outputs of an always block (a process) must be declared as reg.

Behavioural Description of a Comb. Block:

```
module comp_v ( IN1, IN2, X, Y, Z);
   input IN1, IN2, X, Y;
   output Z;
   reg Z;

always @(IN1 or IN2 or X or Y)
begin
    if (X == Y)
          Z <= IN1;
    else
          Z <= IN2;
endmodule;
```

Activation List

Tells the simulator when to run this block

Allows the user to specify when to run the block and makes the simulator more efficient.

- If not sensitized to every input, you get a storage element

But also enables subtle errors to enter into the design (as in VHDL)

Two forms of activation list in Verilog:

@(signalName or signalName or …)

- Evaluate this block when any of the named signals change

@posedge(signalName); or @negedge(signalName);

- Makes an edge triggered flop. Evaluates only on one edge of a signal.

Activation List Examples

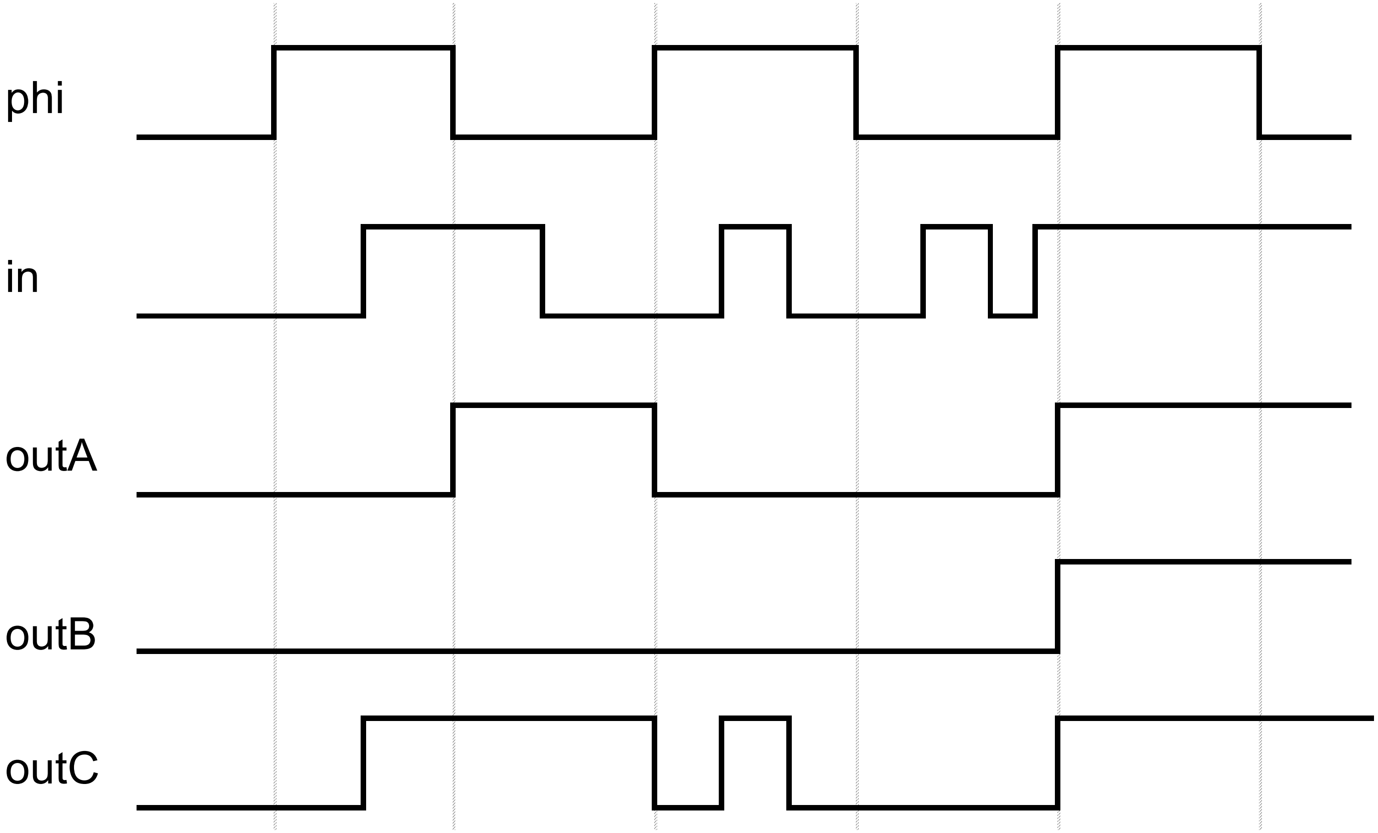
```
always @(phi)
    outA =in;
```

vs

```
always @(phi)
    if(phi) outB = in;
```

vs

```
always @(phi or in)
    if(phi) outC = in;
```



Initial Block

This is another type of procedural block

- Does not need an activation list
- It is run just once, when the simulation starts.

Used to do extra stuff at the very start of simulation

- Initialize simulation environment
- Initialize design

  This is usually only used in the first pass of writing a design.

  Beware, real hardware does not have initial blocks.
- Best to use initial blocks only for non-hardware statements (like $display or $gr_waves)

Wire vs. Reg

There are two types of variables in Verilog:

- – Wires (all outputs of assign statements must be wires)
- – Regs (all outputs of always blocks must be regs)

Both variables can be used as inputs anywhere

- – Can use regs or wires as inputs (RHS) to assign statements
- – assign bus = LatchOutput + ImmediateValue
- – bus must be a wire, but LatchOutput can be a reg
- – Can use regs or wires as inputs (RHS) in always blocks

```
always @ (in or clk)
    if (clk) out = in  (in can be a wire, out must be a reg)
```

+ Delays in Verilog

Verilog simulated time is in “units” or “ticks”.

- Simulated time is unrelated to the wallclock time to run the simulator.
- Simulated time is supposed to model the time in the modelled machine
  - It is increased when the computer is finished modelling all the changes that were supposed to happen at the current simulated time. It then increases time until another signal is scheduled to change values.

User must specify delay values explicitly to Verilog

- # delayAmount
  - When the simulator sees this symbol, it will stop what it is doing, and pause delayAmount of simulated time (# of ticks).
  - Delays can be used to model the delay in functional units, but we will not use this feature. All our logic will have zero delay. Can be tricky to use properly.

+ Delays in Verilog

```
always @(phi or in)
        #10 if (phi) then out = in;
```

This code will wait 10 ticks after either input changes, then checks to see if phi == 1, and then updates the output. If you wanted to sample the input when it changed, and then update the output later, you need to place the delay in a different place:

```
always @(phi or in)
        if (phi) then out = #10 in;
```

This code runs the code every time the inputs change, and just delays the update of the output for 10 ticks.

+ Delays in Verilog

Think about this example:

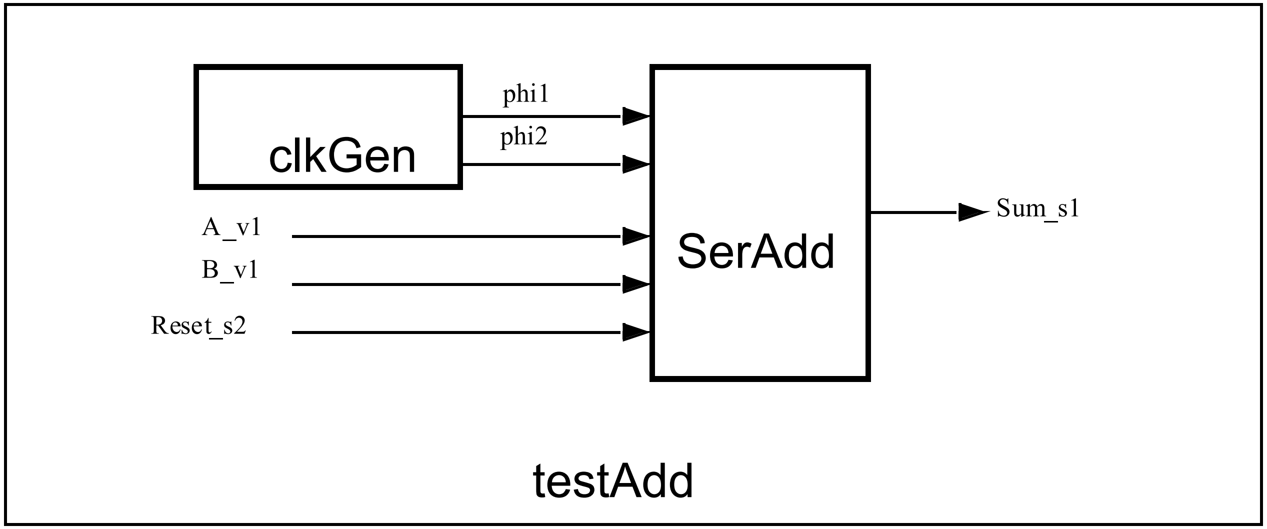
```
always
    #100 out = in;
```

Since the always does not have an activation, it runs all the time. As a result every 100 time ticks the output is updated with the current version of the input.

Delay control is used mostly for clock or pattern generation

Simple Example

Here is a simple example of a serial Adder called serAdd that is called by a top-level module called testAdd



```
// serAdd.v -- 2 phase serial adder module
module serAdd(Sum_s1, A_v1, B_v1, Reset_s2,
       phi1, phi2);
output Sum_s1;
input  A_v1, B_v1, phi1, phi2, Reset_s2;

reg Sum_s1;
reg A_s2, B_s2, Carry_s1, Carry_s2;

always @(phi1 or A_v1)
    if (phi1)
        A_s2 = A_v1;

always @(phi1 or B_v1)
    if (phi1)
        B_s2 = B_v1;

always @(A_s2 or B_s2 or Reset_s2 or Carry_s2 or phi2)
    if (phi2)
      if (Reset_s2) begin
       Sum_s1 = 0;
       Carry_s1 = 0;
      end
      else begin
        Sum_s1   = A_s2 + B_s2 + Carry_s2;
        Carry_s1 = A_s2 & B_s2 |
              A_s2 & Carry_s2 |
              B_s2 & Carry_s2;
      end

always @(Carry_s1 or phi1)
    if (phi1)
        Carry_s2 = Carry_s1;

endmodule
```

```
// testAdd.v -- serial adder test vector generator


// 2 phase clock generator

module clkGen(phi1, phi2);
output phi1,phi2;
reg phi1, phi2;

initial
        begin
                phi1 = 0;
                phi2 = 0;
        end

always
        begin
                #100
                        phi1 = 0;
                #20
                        phi2 = 1;
                #100
                        phi2 = 0;
                #20
                        phi1 = 1;
        end
endmodule

/*
The above clock generator will produce a clock with a period of 240 units of
simulation time.
*/
```

```
/* // test module for the adder
module testAdd; // top level

wire     A_v1, B_v1;
reg      Reset_s2;

serAdd serAdd(Sum_s1, A_v1, B_v1, Reset_s2, phi1, phi2);

/*
 The serial adder takes inputs during phi1
 and produces _s1 outputs during phi2.
 The _s1 output corresponds to the addition of
 the inputs at the previous falling edge of phi1
*/

clkGen clkGen(phi1,phi2);


reg [5:0] tstVA_s1, tstVB_s1;
reg [6:0] accum_Sum;

initial
 $gr_waves("phi1",phi1,"phi2",phi2,
              "Reset_s2",Reset_s2,"A_v1",A_v1,
      "B_v1",B_v1,"Sum_s1",Sum_s1,
      "Carry_s1",serAdd.Carry_s1,
      "accum_Sum",accum_Sum);

/*
Since SerAdd is a serial adder, we put in the operands one bit at a time, and
accumulate the output one bit at a time.
 */
assign A_v1 = tstVA_s1[0];
assign B_v1 = tstVB_s1[0];

always @(posedge phi1) begin
        #10
        release A_v1;
        release B_v1;
end
```

```
always @(posedge phi2) begin
        #10
        force A_v1 = 1'b0;
        force B_v1 = 1'b0;
end

initial begin
    Reset_s2 = 1;
    tstVA_s1 = 6'b01000;
    tstVB_s1 = 6'b11010;
    accum_Sum = 0;
    @(posedge phi1)
        #50 Reset_s2 = 0;
end

always @(negedge phi1) begin
    $display ("A_v1=%h, B_v1=%h,
         sum_s1=%h, time=%d",
         A_v1, B_v1, Sum_s1,$time);
    accum_Sum = accum_Sum << 1 | Sum_s1;
    $display ("tstVA=%h, tstVB=%h,
                sum_s1=%h,accum_Sum=%h\n",
         tstVA_s1,tstVB_s1,Sum_s1,accum_Sum);
end

always @(posedge phi2) begin
 #15
    if (~Reset_s2) begin
      tstVA_s1 = tstVA_s1 >> 1;
      tstVB_s1 = tstVB_s1 >> 1;
      if (tstVA_s1 == 0 && tstVB_s1 == 0) begin
            #800 $stop;
      end
    end
end
endmodule
```

Verilog

From what I’ve told you, you don’t know enough Verilog to thoroughly understand that code.

I am going to leave it up to you to “fill in the holes” by finding a Verilog book or following the Verilog links from the course home page.

How much you learn is up to you:

A “C” student will get by with what is in these notes

A “B” student will want to read a bit more, at least enough to understand the example on the previous pages

An “A” student will want to read quite a bit more