Lecture Notes from Week 3.

Verilog Overview (this lecture is not meant to provide a detailed description of Verilog syntax and symantics, only an overview of major concepts in Verilog modeling).

Two Aspects: Structural and Functional (Behavioral)

Structural: Basic structural element of Verilog is the module, which specifies the I/O ports of a hardware unit. For example:

```
// the output "bits" is the number of bits of the input value "data" that are set (1).
module CountBits(data, bits);
input [7:0] data;
output [4:0] bits;
…
endmodule
```

This segment of Verilog specifies a hardware unit called CountBits that takes eight bits of "data" as input, and produces four bits of "bits" as output. We have not yet determined how the value of bits is computed from data.

A purely structural approach would be define another module called ConditionalInc that conditionally increments a four bit value. ConditionalInc could have inputs A[3:0] and CONTROL, and outputs Z[3:0] as in

```
module ConditionaInc(A,CONTROL,Z);
input [3:0] A;
input CONTROL;
output Z;
…
endmodule
```

Though we have not specified how ConditionalInc performs the computation to produce Z, we can now write module CountBits as

```
module CountBits(data, bits);
input [7:0] data;
output [4:0] bits;
wire [7:0] tmp1,tmp2,tmp3,tmp4,tmp5,tmp6,tmp7;

Inc0 ConditionalInc(0,data[0],tmp1);
Inc1 ConditionalInc(tmp1,data[1],tmp2);
…
Inc7 ConditionaInc(tmp7,data[7],bits);

endmodule
```

Module CountBits instantiates eight instances of ConditionalInc (with instance names Inc0 through Inc7) to compute the final result of bits. The intermediate tmp values are declared as wires because they serve only to connect outputs of one instance to the inputs of another, and as such, do not have to 'hold' or 'drive' a signal value.

A structural implementation of ConditionalInc might include more levels of instance hierarchy, but more likely, it would be specified by instantiating Verilog primitive modules such as and-gates, or-gates, inverters, etc. Verilog primitives are simply pre-defined modules supported by the Verilog language that include a variety of logic functions.

The choice of modeling CountBits as a combinational or sequential system is independent of the choice between structure and function. The above example is likely to be implemented as a combinational systems, but it is also possible for ConditionalInc to be implemented by a bit-serial incrementer, for example.

A functional model for a combinational implementation of CountBits is shown below.

```
module CountBits(data, bits);
input [7:0] data;
output [4:0] bits;
assign bits = data[0] + data[1] + data[2] +…+ data[7];
endmodule
```

In this example, a continuous assignment is used to directly compute the value of bits with little suggestion as to the actual physical implementation that should be used. A continuous assignment will be re-evaluated by the simulator every time any of the values on the left-hand-side (LHS) change…so the continuous assignment is a way to model combination logic. Another equivalent way to model combinational logic is with an "always" statement that has all of the LHS value in its sensitivity list, as in:

```
module CountBits(data, bits);
input [7:0] data;
output [4:0] bits;
reg [8:0] i;
reg [4:0] bits;
always @(data[7:0])  // this is the sensitivity list
 for (i=0; i<8;i=i+1)
     bits = bits + data[i];
endmodule
```

Just like the continuous assignment, this procedural always block is re-executed every time any of the data bits change. And also like the continuous assignment, the final new value for bits is computed in zero simulation time, the for-loop notwithstanding. Notice that in this case, we need to declare bits and i as Verilog registers (reg). Note that "i" is declared as a 9 bit value so that the for-loop can detect when i greater than 7. This does not imply that these values will necessarily be mapped to a storage element in the final

implementation. In fact, in this combinational model, they won't. A good synthesis tool will produce the exact same hardware implementation for both models.

The sensitivity list after the always statement "@(data[7:0]) is the list of signals that cause the statement contained in the always block to be re-executed. In a combinational system, this list must contain all of the values that affect the output. Notice that "i" is not contained in the sensitivity list because it is a locally computed valued that is re-initialized each time the always statement is executed.

An always block can also be used to model synchronous sequential systems by replacing the sensitivity list with a positive or negative edge of the clock input. The following is a synchronous sequential version of CountBits.

```
module CountBits(data, bits, clock);
input [7:0] data;
input clock
output [4:0] bits;
reg [7:0] oldData, tmp;
reg [4:0] bits;
always @(posedge clock)
 if (data != oldData) begin
     oldData <= data;
     tmp <= data;
     bits <= 0;
 else begin
     // if using procedural assignments the order
     // of these next two statements is important
     // but not if we use RTL assignments as below
     bits <= bits + tmp[0];
     tmp <= tmp >> 1;
 end
endmodule
```

This model is a bit more complicated. Only on the rising edge of the clock will the always block will be re-evaluated. In this model, the final value of bits is computed by sequentially scanning through the tmp value, considering only one bit of input per clock cycle. This model requires 8 clock cycles to compute the result. Notice that the "RTL" assignment '<=' has been used in favor of the procedural assignment '='. In this case, either would have worked just as well. The difference between the two is that RTL assignments are evaluated in simultaneous simulation time, as though each LHS value is a clock register operating in parallel with all of the others. The results of a sequence of RTL assignments are order independent. In contrast, a sequence of procedural assignments are order dependent, with the results of earlier assignments possibly effecting the results of later ones. As a simple example, the following sequence of RTL assignments implements a swap:

```
A <= B;
```

```
B <= A;
```

While the following one does not. At the end of the this sequence both A and B will be assigned to the original value of B.

```
A=B;
B=A;
```

State Machines

Generally, state machines consist of a state register whose value changes only on a specific clock transition, and a block of combinational logic that computes outputs and next state as a function of inputs and current state. So we can write a general state machine in Verilog as:

```
module…
…
always @(posedge clock)
     if (reset) state = `start;    // implements
synchronous reset
     else state = next_state;

always @(<inputs>, <state>) begin
     next_state = F(inputs, state);
     outputs = G(inputs, state);
end

endmodule
```

The sequential version of CountBits, which is a state machine, could be written in this form as in the following example, where we define state to be the concatenation of values {bits, oldData and tmp}.

```
module CountBits(data, bits, clock);
input [7:0] data;
input clock
output [4:0] bits;
reg [19:0] state;
assign bits = state[19:16];
always @(posedge clock)
     state = next_state;    // no reset input (self
resetting)

always @(data)
 if (data != state[15:8])
     state = {4'b0,data,data};  // clear bits, set
oldData and tmp to data
 else
```

```
      state = {state[19:16] + state[0], state[15:8],
1'b0,state[7:1]};

endmodule
```

Next week we will talk about how these models are synthesized into structural models that map directly to hardware elements such as FPGA Logic Modules/CLB's or ASIC standard cells.