CIS 410/510
HDL: Verilog Fundamentals
Part II

Prof. Michel A. Kinsky
Assignments

• Continuous and procedural assignment statements are very different

Continuous assignments are for naming and thus we cannot have multiple assignments for the same wire

```verbatim
wire out, t0, t1;
assign t0  = ~( (sel[1] & c) | (~sel[1] & a) ) ;
assign t1  = ~( (sel[1] & d) | (~sel[1] & b) ) ;
assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) ) ;
```

Procedural assignments hold a value semantically, but it is important to distinguish this from hardware state

```verbatim
reg out, t0, t1, temp;
always @( *)
begin
  temp = ~( (sel[1] & c) | (~sel[1] & a) ) ;
t0  = temp;
temp = ~( (sel[1] & d) | (~sel[1] & b) ) ;
t1  = temp;
out  = ~( (t0 | sel[0]) & (t1 | ~sel[0]) ) ;
end
```
Always Blocks

- Always blocks can contain more advanced control constructs

```verilog
module mux4( input  a, b, c, d
        input [1:0] sel,
        output  out );

reg  out;
always @(  *  )
begin
  if (  sel ==  2'd0  )
    out = a;
  else if (  sel ==  2'd1  )
    out = b;
  else if (  sel ==  2'd2  )
    out = c;
  else if (  sel ==  2'd3  )
    out = d;
  else
    out = 1'bx;
end
endmodule
```

```verilog
module mux4( input  a, b, c, d
        input [1:0] sel,
        output  out );

reg  out;
always @(  *  )
begin
  case (  sel  )
    2'd0 : out = a;
    2'd1 : out = b;
    2'd2 : out = c;
    2'd3 : out = d;
    default : out = 1'bx;
  endcase
end
endmodule
```
Case Statements

- What happens if the case statement is not complete?

```verilog
module mux3( input a, b, c,
             input [1:0] sel,
             output out );

  reg out;

  always @( *)
  begin
    case ( sel )
      2'd0 : out = a;
      2'd1 : out = b;
      2'd2 : out = c;
    endcase
  end
endmodule
```

If sel = 3, mux will output the previous value. What have we created?
Case Statements

• What happens if the case statement is not complete?

```verilog
module mux3( input a, b, c
            input [1:0] sel,
            output out );

reg out;

always @( * )
begin
    case ( sel )
      2'd0 : out = a;
      2'd1 : out = b;
      2'd2 : out = c;
      default : out = 1'bx;
    endcase
end

endmodule
```

We can prevent creating state with a default statement.
module latch
(
    input clk,
    input d,
    output reg q
);

always @(clk)
begin
    if (clk)
        d = q;
end
endmodule

module flipflop
(
    input clk,
    input d,
    output q
);

always @(posedge clk)
begin
    d = q;
end
endmodule

Edge-triggered always block
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @(posedge clk )
  A_out = A_in;
assign B_in = A_out + 1;
always @(posedge clk )
  B_out = B_in;
assign C_in = B_out + 1;
always @(posedge clk )
  C_out = C_in;
Active Event Queue

On clock edge all those events which are sensitive to the clock are added to the active event queue in any order!

```verilog
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
  A_out = A_in;

assign B_in = A_out + 1;

always @(posedge clk)
  B_out = B_in;

assign C_in = B_out + 1;

always @(posedge clk)
  C_out = C_in;
```
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
  A_out = A_in;

assign B_in = A_out + 1;

always @(posedge clk)
  B_out = B_in;

assign C_in = B_out + 1;

always @(posedge clk)
  C_out = C_in;

Active Event Queue
More Verilog Semantics

```
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
  A_out = A_in;

assign B_in = A_out + 1;

always @(posedge clk)
  B_out = B_in;

assign C_in = B_out + 1;

always @(posedge clk)
  C_out = C_in;
```
More Verilog Semantics

```verilog
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk) begin
    A_out = A_in;
end

assign B_in = A_out + 1;

always @(posedge clk) begin
    B_out = B_in;
end

assign C_in = B_out + 1;

always @(posedge clk) begin
    C_out = C_in;
end
```

Active Event Queue

A

1

B

2

C
More Verilog Semantics

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @(posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @(posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @(posedge clk )
  C_out = C_in;
```

A evaluates and as a consequence 1 is added to the event queue
More Verilog Semantics

wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
  A_out = A_in;
assign B_in = A_out + 1;

always @(posedge clk)
  B_out = B_in;
assign C_in = B_out + 1;

always @(posedge clk)
  C_out = C_in;

Active Event Queue

Event queue is emptied before we go to next clock cycle
More Verilog Semantics

```verilog
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk )
  A_out = A_in;
assign B_in = A_out + 1;

always @(posedge clk )
  B_out = B_in;
assign C_in = B_out + 1;

always @(posedge clk )
  C_out = C_in;
```

Active Event Queue

Event queue is emptied before we go to next clock cycle
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
    A_out = A_in;

assign B_in = A_out + 1;

always @(posedge clk)
    B_out = B_in;

assign C_in = B_out + 1;

always @(posedge clk)
    C_out = C_in;
Non-Blocking Assignments

wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
    A_out <= A_in;
assign B_in = A_out + 1;

always @(posedge clk)
    B_out <= B_in;
assign C_in = B_out + 1;

always @(posedge clk)
    C_out <= C_in;

Non-blocking procedural assignments add an extra event queue

Active Event Queue

Non-Blocking Queue
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk )
A_out <= A_in;

assign B_in = A_out + 1;

always @(posedge clk )
B_out <= B_in;

assign C_in = B_out + 1;

always @(posedge clk )
C_out <= C_in;

Non-blocking procedural assignments add an extra event queue
Non-Blocking Assignments

Non-blocking procedural assignments add an extra event queue

```
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
begin
  A_out <= A_in;
  B_out <= B_in;
  C_out <= C_in;
end

assign B_in = A_out + 1;
assign C_in = B_out + 1;
```

The order of non-blocking assignments does not matter!
Common Patterns

- Common patterns for latch and flip-flop inference

```verilog
always @( clk )
begin
    if ( clk )
        D <= Q;
end

always @( posedge clk )
begin
    D <= Q;
end

always @( posedge clk )
begin
    if ( enable )
        D <= Q;
end
```
Blocking vs. Non-blocking

- Guidelines for using blocking and non-blocking assignment statements
  - Flip-flops should use non-blocking
  - Latches should use non-blocking
  - Combinational logic should use blocking
  - Do not mix combinational and sequential logic in the same always block
  - Do not assign to the same variable from more than one always block
Behavioral Verilog Usage

• Behavioral Verilog is used to model the abstract function of a hardware module

  ‣ Characterized by heavy use of sequential blocking statements in large always blocks

  ‣ Many constructs are not synthesizable but can be useful for behavioral modeling
    • Data dependent for and while loops
    • Additional behavioral datatypes: integer, real
    • Magic initialization blocks: initial
    • Magic delay statements: #<delay>
High-level Behavior

- Verilog can be used to model the high-level behavior of a hardware block

```verilog
module factorial( input [ 7:0] in, output reg [15:0] out );

integer num_calls;
initial num_calls = 0;

integer multiplier;
integer result;
always @(*)
begin

multiplier = in;
result = 1;
while ( multiplier > 0 )
begin
result = result * multiplier;
multiplier = multiplier - 1;
end

out = result;
num_calls = num_calls + 1;
end

endmodule
```

- Initial statement
- Variables of type integer
- Data dependent while loop
Delay Statements

- Delay statements should only be used in test module

```verilog
module mux4
(
    input    a,
    input    b,
    input    c,
    input    d,
    input [1:0] sel,
    output   out
);

wire #10 t0  = ~( (sel[1] & c) | (~sel[1] & a) );
wire #10 t1  = ~( (sel[1] & d) | (~sel[1] & b) );
wire #10 out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
endmodule
```

Although this will add a delay for simulation, these are ignored in synthesis.
• Even synthesizable blocks can be more behavioral

```verilog
module ALU
(
    input [31:0] in0,
    input [31:0] in1,
    input [1:0] fn,
    output [31:0] out
);

assign out = ( fn == 2'd0 ) ? ( in0 + in1 ) :
               ( fn == 2'd1 ) ? ( in0 - in1 ) :
               ( fn == 2'd9 ) ? ( in1 >> in0 ) :
               ( fn == 2'd10 ) ? ( in1 >>> in0 ) :
               32'bx;
endmodule
```

Although this module is synthesizable, it is unlikely to produce the desired hardware.
System Testing

reg [ 1023:0 ] exe_filename;

initial
begin

    // This turns on VCD (plus) output
    $vcdpluson(0);

    // This gets the program to load into memory from the command line
    if ( $value$plusargs( "exe=%s", exe_filename ) )
        $readmemh( exe_filename, mem.m );
    else
        begin
            $display( "ERROR: No executable specified! (use +exe=<filename>)" );
            $finish;
        end

    // Stobe reset
    #0    reset = 1;
    #38   reset = 0;

end
Which abstraction is the right one?

- Designers usually use a mix of all three:
  - Early on in the design process they might use mostly behavioral models.
  - As the design is refined, the behavioral models begin to be replaced by dataflow models.
  - Finally, the designers use automatic tools to synthesize a low-level gate-level model.
Take away points

- Structural Verilog enables us to describe a hardware schematic textually.
- Verilog can model hardware at three levels of abstraction: gate level, register transfer level, and behavioral.
- Understanding the Verilog execution semantics is critical for understanding blocking + non-blocking assignments.
- Designers must have the hardware they are trying to create in mind when they write their Verilog.
Next Class

- Combinational Circuit Design