EN164: Design of Computing Systems
Lecture 06: Lab Foundations / Verilog 2

Professor Sherief Reda
http://scale.engin.brown.edu
Electrical Sciences and Computer Engineering
School of Engineering
Brown University
Spring 2011
2. Dataflow modeling

- Module is designed by specifying the data flow, where the designer is aware of how data flows between hardware registers and how the data is processed in the design.

- The continuous assignment is one of the main constructs used in dataflow modeling:
  - `assign out = i1 & i2;`
  - `assign addr[15:0] = addr1[15:0] ^ addr2[15:0];`
  - `assign {c_out, sum[3:0]}=a[3:0]+b[3:0]+c_in;`

- A continuous assignment is always active and the assignment expression is evaluated as soon as one of the right-hand-side variables change.

- Assign statements describe hardware that operates concurrently – ordering does not matter.

- Left-hand side must be a scalar or vector net. Right-hand side operands can be wires, (registers, integers, and real).
Operator types in dataflow expressions

• Operators are similar to C except that there are no ++ or –

• **Arithmetic:** *, /, +, -, % and **
  • **Logical:** !, && and ||
  • **Relational:** >, <, >= and <=
  • **Equality:** ==, !=, === and !==
  • **Bitwise:** ~, &, |, ^ and ^~
  • **Reduction:** &, &&, |, ||, ^ and ^~
  • **Shift:** << and >>
  • **Concatenation:** { }
  • **Replication:** {{}}
  • **Conditional:** ?:
Examples of 2x1 MUX and 4x1 MUX

module mux2to1(s, a, b, y);
output y;
input s, a, b;
assign y = (b & s) | (a & ~s);
// OR THIS WAY
assign y = s ? b : a;
endmodule

module mux4to1(out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
output s1, s0;
assign out = (~s1 & ~s0 & i0) |
            (~s1 & s0 & i1) |
            (s1 & ~s0 & i2) |
            (s1 & s0 & i3);
// OR THIS WAY
assign out = s1 ? (s0 ? i3:i2) : (s0 ? i1:i0);
endmodule
Difference between HLL and Verilog assign

(a) assignment statement ordering does matter in an HLL

\[ a = 1; b = 0; s = 0; \]
\[ na = 0; nb = 0; \]
\[ y = na \lor nb; \]
\[ nb = b \land s; \]
\[ na = a \land \neg s; \]
\[ \text{Final } y \text{ value is 0.} \]

(b) assign statement ordering does not matter in Verilog

\[ \text{wire } na, nb; \]
\[ \text{assign } y = na \lor nb; \]
\[ \text{assign } nb = b \land s; \]
\[ \text{assign } na = a \land \neg s; \]
\[ \text{Final } y \text{ value is 1.} \]

[Example from Thornton & Reese]
Difference between HLL and Verilog assign

(a) assignment statements in an HLL can target the same variable

\[
\begin{align*}
    a &= 1; \\
    b &= 0; \\
    s &= 0; \\
    na &= 0; \\
    nb &= 0; \\
    na &= b \& s; \\
    na &= a \& \sim s;
\end{align*}
\]

The na variable is assigned twice; the final value of na is the last assignment.

(b) illegal use of assign statements

\[
\begin{align*}
    \text{wire na;} \\
    \text{assign na = b \& s;} \\
    \text{assign na = a \& \sim s;}
\end{align*}
\]

Gate outputs are shorted together!

can only work with tri-state drivers

[Example from Thornton & Reese]
Example of a dataflow 4-bit adder

(a) Four-bit adder with no carry-in or carry-out

```verilog
// 4-bit adder
// no carry-in, carry-out
module add4bit (a, b, s);

input [3:0] a, b;
output [3:0] s;
assign s = a + b;
endmodule
```

(b) Four-bit adder with carry-in, carry-out

```verilog
// 4-bit adder with carry-in, carry-out
module add4bit (ci, a, b, s, co);

input ci;
input [3:0] a, b;
output [3:0] s;
output co;
wire [4:0] y;

// do 5-bit sum so that we have access to carry-out
assign y = '{1'b0, a} + {1'b0, b} + {4'b0, ci};
assign s = y[3:0]; // four-bit output
assign co = y[4]; // carry-out
endmodule
```

[Example from Thornton & Reese]
3. Behavioral modeling

• Design is expressed in algorithmic level, which frees designers from thinking in terms of logic gates or data flow.

• All algorithmic or procedural statements in Verilog can appear only inside two statements: always and initial.

• Each always and initial statement represents a separate activity flow in Verilog. Remember that activity flows in Verilog run in parallel.

• You can have multiple initial and always statements but you can’t nest them.

```verilog
  reg a, b, c;

  initial a=1'b0;

  always @*
     begin
       b = a ^ 1'b1;
       c = a + b;
     end
```

Data types

• A `reg` is a Verilog variable type and does not necessarily imply a physical register. Think of it as a variable or place holder. It is unsigned by default.
  ▪ `reg clock;`
  ▪ `reg [0:40] virt_address;`

• Register arrays or memories. Used to model register files, RAMs and ROMs. Modeled in Verilog as a one-dimensional array of registers. Examples.
  ▪ `reg mem1bit[0:1023];`
  ▪ `reg [7:0] membyte[0:1023];`
  ▪ `accessing: membyte[511];`

• Parameters. Define constants and cannot be used as variables.
  ▪ `parameter port_id=5;`
Data types

• integers (signed) and reals. They are type of `reg'.
  ▪ `real delta;
  ▪ `integer i;
  ▪ `initial
  ▪ `begin
    ▪ `delta = 4e10;
    ▪ `i = 4;
  ▪ `end

• Arrays of integers and real.
  ▪ `integer count[0:7];
  ▪ `integer matrix[4:0][0:255];

• Strings can be stored in `reg'. The width of the register variables must be large enough to hold the string.
  ▪ `reg [8*19:1] string_value;
  ▪ `initial
    ▪ `string_value = "Hello Verilog World";
initial statements

• An initial block start at time 0, executes exactly once and then never again.

• If there are multiple initial blocks, each blocks starts to execute concurrently at time 0 and each blocks finish execution independently of the others.

• Multiple behavioral statements must be grouped using begin and end. If there is one statement then grouping is not necessary.

In procedural statements (initial, always) LHS must be of type registers (and its derivatives)

```vhvl
reg x, y, m;
initial m=1'b0;
initial begin
    x = 1'b0;
    y = 1'b1;
end
```
always statements

• The always statement starts at time 0 and executes the statements in the always block when the events in its sensitivity list occur.

• Powerful constructs like if, if-else, case, and looping are only allowed inside always blocks.

• always statements can be used to implement both combinational or sequential logic.

• Multiple behavioral statements must be grouped using begin and end.

• Multiple always statement can appear in a module.

module mux2to1(s,a,b,y);
input s,a,b;
output y;

reg y;

//use boolean ops
always @(a or b or s)
begin
  y = (b & s) | (a & ~s);
end
endmodule
Sensitivity list of events

- An event is the change in the value on a register or a net. Events can be utilized to trigger the execution of a statement of a block of statements.

- The @ symbol is used to specify an event control.

- For combinational logic, any net that appears on the right side of an “=” operator in the always block should be included in the event list.

- [for sequential – ignore for now] Statements can be executed on changes in signal value or at a positive (posedge) or negative (negedge) transition of the signal.

```module mux2to1(s,a,b,y);
input s,a,b;
output y;

reg y, na, nb;

//use intermediates
//and implicit event
//list
always @*
begin
    nb = b & s;
    na = a & ~s;
    y = na | nb;
end
endmodule
```
always statements

- Any net that is assigned within an always block must be declared as a `reg` type; this does not imply that this net is driven by a register or sequential logic.

- The “=” operator when used in an always block is called a blocking assignment.

- If there is some logic path through the always block that does not assign a value to the output net then a latch is inferred.

- The logic synthesized assumed the blocking assignments are evaluated sequentially. This means that the order in which assignments are written in an always blocks affects the logic that is synthesized.

![Diagrams](a) Incorrect, produces an inferred latch as no assignment is made to q if ld is ‘0’

```vhdl
always @(ld or d)
begin
  if (ld) q = d;
end
```

(b) Correct, produces combinational logic

```vhdl
always @(ld or d or q_old)
begin
  q = q_old;
end
```

![Logic Diagrams](a) Incorrect, produces an inferred latch as no assignment is made to q if ld is ‘0’

```vhdl
always @(ld or d)
begin
  if (ld) q = d;
end
```

(b) Correct, produces combinational logic

```vhdl
always @(ld or d or q_old)
begin
  q = q_old;
end
```
always statements

• Because of the sequential nature of an always block, the same net can be assigned multiple times in an always block; the last assignment takes precedence.

[Example from Thornton & Reese]
Conditional statements

- Very similar to C
- Can always appear inside always and initial blocks

```verbatim
// if(x)
begin
  y = 'b1;
  z = 'b0;
end

// expression
if (count < 10)
  count = count + 1;
else
  count = 0;

reg [1:0] alu_control;
..
case (alu_control)
  2'd0 : y = x + z;
  2'd1 : y = x - z;
  2'd2 : y = x * z;
  default: y = x;
endcase
```
Example: Mux4x1

module mux4x1(out, i0, i1, i2, i3, s1, s0);
  output out;
  input i0, i1, i2, i3;
  input s1, s0;
  reg out;
  
  always @(s1 or s0 or i0 or i1 or i2 or i3)
  begin
    case({s1, s0})
      2’d0: out = i0;
      2’d1: out = i1;
      2’d2: out = i2;
      2’d3: out = i3;
    endcase
  end
endmodule
Level sensitive latch (D-Latch)

- The Verilog implementation of a D-latch is an always block that makes a nonblocking assignment ("<=") of d to q when the g input is nonzero.

- When g input is zero, then the always block does not make any assignment to q, causing the synthesis tool to infer a latch on the q output as the q output must retain its last known d value when g was nonzero.

- Nonblocking assignments ("<=") as opposed to blocking assignments ("=") should be used in always blocks that are used to synthesize sequential logic.

[from Thornton & Reese]
Edge-triggered storage element (D-FF)

- The `@` symbol is used to specify an event control.
- Statements can be executed on changes in signal value or at a positive (posedge) or negative (negedge) transition of the signal.
- In general, edge-triggered storage elements are preferred to level-sensitive storage elements because of simpler timing requirements.
- The 1-bit edge-triggered storage elements provided by FPGA vendors are DFFs because of their simplicity and speed.

[Thornton & Reese]
module quest3(CLOCK_50, LEDR);
input CLOCK_50;
output reg [17:0] LEDR;
integer count;
always @(posedge CLOCK_50)
begin
    if(count == 50000000)
    begin
        LEDR[0] <= !LEDR[0];
        count <= 0;
    end
    else count <= count + 1;
end
endmodule