EN2911X: Reconfigurable Computing
Lecture 05: Verilog (2)

Prof. Sherief Reda
Division of Engineering, Brown University
Fall ‘09
http://scale.engin.brown.edu
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 registers, nets, integers, 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

```verilog
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
```
Example: 4-bit 2x1 MUX

(a) Four-bit 2-to-1 multiplexer using four of the previously defined one-bit 2-to-1 modules named mux2to1

```
module mux2to1_4bit(s, a, b, y);
  input s;
  input [3:0] a, b;
  output [3:0] y;

mux2to1 u3 (.a(a[3]), .b(b[3]), .s(s),
           .y(y[3]));
mux2to1 u2 (.a(a[2]), .b(b[2]), .s(s),
           .y(y[2]));
mux2to1 u1 (.a(a[1]), .b(b[1]), .s(s),
           .y(y[1]));
mux2to1 u0 (.a(a[0]), .b(b[0]), .s(s),
           .y(y[0]));
endmodule
```

Port a of instance u3 connects to port a[3]

(b) Four-bit 2-to-1 multiplexer using a single assign statement

```
module mux2to1_4bit(s, a, b, y);
  input s;
  input [3:0] a, b;
  output [3:0] y;

assign y = (b & !s) | (a & ~s);
endmodule
```

Bus definition for a 4-bit bus; most significant bit is y[3] and least significant bit is y[0].

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

assign

(a) assignment statement ordering does matter in an HLL

\[
\begin{align*}
  a &= 1; \\
  b &= 0; \\
  s &= 0; \\
  na &= 0; \\
  nb &= 0; \\
  y &= na \lor nb; \\
  nb &= b \land s; \\
  na &= a \land \lnot s; \\
  y &= na \lor nb;
\end{align*}
\]

Final \( y \) value is 0.

(b) assign statement ordering does not matter in Verilog

\[
\begin{align*}
  \text{wire} \ na, nb; \\
  \text{assign} \ y &= na \lor nb; \\
  \text{assign} \ nb &= b \land s; \\
  \text{assign} \ na &= a \land \lnot s;
\end{align*}
\]

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

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

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

The \textit{na} variable is assigned twice; the final value of \textit{na} is the last assignment.

(b) illegal use of \texttt{assign} statements

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

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]
Behavioral modeling (combinational)

- Design is expressed in algorithmic level, which frees designers from thinking in terms of logic gates or data flow.
- Designing at this model is very similar to programming in C.
- All algorithmic 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
```
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.

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

• The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion.

• 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.

• It models a block of activity that is repeated continuously in a digital circuit. Multiple behavioral statements must be grouped using begin and end.

• multiple always statement can appear in a module

```verilog
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
```
Events-based timing control

- 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.

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

(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;
  if (ld) q = d;
end
```

ld --- D Q --- q

ld --- G --- d

q_old --- q

ld --- 1 --- q
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

```vhdl
expression
if (alu_control == 0)
  y = x + z;
else if (alu_control == 1)
  y = x - z;
else if (alu_control == 2)
  y = x * z;
else
  y = x;
end

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
endmodule
```
Import the given pin assignment file to make life easy!
DE2 example

module test2(SW, LEDG);
input [3:0] SW;
output [8:0] LEDG;
reg x;

always @(SW[0] or SW[1] or SW[2] or SW[3])
begin
    x = SW[3];
    if(SW[0]) x = SW[2];
    if(SW[1]) x = 0;
end
assign LEDG[0] = x;
endmodule
Lab 1

• Please go through the lab0 tutorial to get familiar with the tool and the synthesis environment
• Please check the class webpage for helpful resources
• Deliverables include
  – Working design (.sof file) which will be tested
  – Written documentation includes
    • Verilog source code with comments
    • Report the amount of logic and routing area utilized in the FPGA
    • Snapshot of the final layout of the FPGA as produced by the synthesis tool