CS250 Section 2

Yunsup Lee

9/4/09

Image Courtesy www.intel.com
Upcoming dates

- 9/8/09 (12:30pm) - Lab 1 due (No late days for Lab 1!)
  - Submit using SVN (source, build, writeup)
- 9/8/09 - Lab 2 out
  - Write and Synthesize a Two-Stage SMIPSv2 processor
- Did you know about tutorials?
  - Tutorial 3: Build, Run, and Write SMIPS Programs
  - Tutorial 4: Simulating Verilog RTL using Synopsys VCS
  - Tutorial 5: RTL-to-Gates Synthesis using Synopsys DC
  - SMIPS Specificaition
Heads-up for Lab 1

- standard cell library cache directory
  - check dc-syn/Makefile if ALIB_DIRECTORY exists
- dc.tcl changed
  - copy it from ~cs250/lab1/build/dc-syn/dc_scripts
- change clock period to 1 ns
  - change build/dc-syn/constraints.tcl
Some other stuff...

- NX Client
  - You should use this for fast GUI experience
  - However, ilinux1.eecs is suffering with RAM, so please logout
- SVN
  - [https://isvn.eecs.berkeley.edu/cs250/](https://isvn.eecs.berkeley.edu/cs250/) <username>
  - Guarded with your instructional account password
  - Only commit the most recent build - source files are fine
- License server
  - We had an outage on 9/2 Wednesday
  - Now we have a backup license server - look at cs250.bashrc
Some other stuff...

- Please exit the tools after using
  - Don’t keep your tools occupying one or several licenses
  - It’s a shared resource, and you should be fair
- Check the News page for any updates
  - [http://inst.eecs.berkeley.edu/~cs250/fa09/news.html](http://inst.eecs.berkeley.edu/~cs250/fa09/news.html)
- Tools are only supported on ilinux[1-3].eecs
  - Noticed that VCS (old version) is installed on the Sun Boxes
- Labs and Tutorials have version numbers
  - I’ll try hard not to change the version whenever it’s in transit
- Writeup should be in text or PDF
Any questions

- Let’s form groups of two and talk about Lab 1
- Any questions on lab1 or any tools?
- Critical Path
  - Synopsys Design Vision
  - Synopsys IC Compiler
Primary Verilog data type is a bit-vector

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Logic zero</td>
</tr>
<tr>
<td>1</td>
<td>Logic one</td>
</tr>
<tr>
<td>X</td>
<td>Unknown logic value</td>
</tr>
<tr>
<td>Z</td>
<td>High impedance, floating</td>
</tr>
</tbody>
</table>

An X bit might be a 0, 1, Z, or in transition. We can set bits to be X in situations where we don’t care what the value is. This can help catch bugs and improve synthesis quality.
‘wire’ is used to denote a standard hardware net

wire [15:0] instruction;
wire [15:0] memory_req;
wire [ 7:0] small_net;

Absolutely no type safety when connecting nets!
Verilog includes ways to specify bit literals in various bases

- **Binary literals**
  - `8'b0000_0000`
  - `8'b0xx0_1xx1`

- **Hexadecimal literals**
  - `32'h0a34_def1`
  - `16'haxxx`

- **Decimal literals**
  - `32'd42`

Underscores are ignored.

We’ll learn how to actually assign literals to nets a little later.
Verilog module includes a module name and a port list

module adder( A, B, cout, sum );
input [3:0] A;
input [3:0] B;
output cout;
output [3:0] sum;
// HDL modeling of
// adder functionality
endmodule

Note the semicolon at the end of the port list!

Ports must have a direction (or be bidirectional) and a bitwidth
Verilog module includes a module name and a port list

**Traditional Verilog-1995 Syntax**

```
module adder( A, B, cout, sum );
input [3:0] A;
input [3:0] B;
output cout;
output [3:0] sum;
```

**ANSI C Style Verilog-2001 Syntax**

```
module adder( input [3:0] A,
             input [3:0] B,
             output cout,
             output [3:0] sum );
```
A module can instantiate other modules creating a hierarchy

module FA(
  input  a, b, cin,
  output cout, sum
);

  // HDL modeling of 1 bit
  // adder functionality

endmodule
A module can instantiate other modules creating a hierarchy

```
module adder( input [3:0] A, B,
              output cout,
              output [3:0] S );

wire c0, c1, c2;
FA fa0( A[0], B[0], 1'b0, c0,   S[0] );
FA fa1( A[1], B[1], c0,   c1,   S[1] );
FA fa2( A[2], B[2], c1,   c2,   S[2] );
FA fa3( A[3], B[3], c2,   cout, S[3] );
endmodule
```

Carry Chain
Verilog supports connecting ports by position and by name

Connecting ports by ordered list

```verilog
FA fa0( A[0], B[0], 1'b0, c0, S[0] );
```

Connecting ports by name (compact)

```verilog
FA fa0( .a(A[0]), .b(B[0]),
       .cin(1'b0), .cout(c0), .sum(S[0]) );
```

Connecting ports by name

```verilog
FA fa0
(
  .a (A[0]),
  .b (B[0]),
  .cin (1'b0),
  .cout (c0),
  .sum (S[0])
);
```

For all but the smallest modules, connecting ports by name yields clearer and less buggy code.
Three abstraction levels

- **Behavioral Algorithm**: Abstract algorithmic description
- **Manual Register Transfer Level**: Describes how data flows between state elements for each cycle
- **Logic Synthesis**: Low-level netlist of primitive gates
- **Gate Level**: Auto Place + Route

[Diagram showing the flow from Behavioral Algorithm to Gate Level]
Gate-level Verilog uses structural Verilog to connect primitive gates

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

wire [1:0] sel_b;
not not0( sel_b[0], sel[0] );
not not1( sel_b[1], sel[1] );

wire n0, n1, n2, n3;
and and0( n0, c, sel[1] );
and and1( n1, a, sel_b[1] );
and and2( n2, d, sel[1] );
and and3( n3, b, sel_b[1] );

wire x0, x1;
nor nor0( x0, n0, n1 );
nor nor1( x1, n2, n3 );

wire y0, y1;
or or0( y0, x0, sel[0] );
or or1( y1, x1, sel_b[0] );
nand nand0( out, y0, y1 );

endmodule
```
Blocking vs. Non-blocking

- **Blocking** procedural assignment “=”
  - **In simulation** the RHS is executed and the assignment is completed before the next statement is executed. Example:
    
    Assume A holds the value 1 ... A=2; B=A; A=? B=?

- **Non-blocking** procedural assignment “<=”
  - **In simulation** the RHS is executed and all assignment take place at the same time (end of the current time step - not clock cycle). Example:
    
    Assume A holds the value 1 ... A<=2; B<=A; A=? B=?

- **In synthesis** the difference shows up primarily when inferring state elements:

  ```
  always @ (posedge clk) begin
    a = in;
    b = a;
  end
  ```

  b stores in

  ```
  always @ (posedge clk) begin
    a <= in;
    b<= a;
  end
  ```

  b stores the old a
Guideline

i. Use blocking assignments to model combinational logic within an always block ("=").

ii. Use non-blocking assignments to implement sequential logic ("<=").

iii. Do not mix blocking and non-blocking assignments in the same always block.

iv. Do not make assignments to the same variable from more than one always block.
Always blocks have parallel inter-block, sequential intra-block semantics

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

    reg  out, t0, t1;

always @( a or b or c or d or sel )
begin
    t0  = ~( (sel[1] & c) | (~sel[1] & a ) );
    t1  = ~( (sel[1] & d) | (~sel[1] & b ) );
    out = ~( (t0  | sel[0]) & (t1  | ~sel[0]) );
end

demodule
```

The always block is reevaluated whenever a signal in its sensitivity list changes
Always blocks have parallel inter-block, sequential intra-block semantics

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

reg out, t0, t1;

always @( a or b or c or d or sel )
begin
  t0 = ~( (sel[1] & c) | (~sel[1] & a ) );
  t1 = ~( (sel[1] & d) | (~sel[1] & b ) );
  out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end

endmodule

What happens if we accidentally forget a signal on the sensitivity list?
module mux4( input  a, b, c, d 
    input [1:0] sel, 
    output out );

    reg out, t0, t1;

    always @( * )
    begin 
        t0  = ~( sel[1] & c ) | (~sel[1] & a ); 
        t1  = ~( sel[1] & d ) | (~sel[1] & b ) ;
        out = ~( t0 | sel[0]) & (t1 | ~sel[0]) ;
    end

endmodule

Verilog-2001 provides special syntax to automatically create a sensitivity list for all signals read in the always block.
Using continuous assignments

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

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]) );
endmodule

The order of these continuous assignment statements does not matter. They essentially happen in parallel!
Using other operators

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

    assign out = ( sel == 0 ) ? a :
                 ( sel == 1 ) ? b :
                 ( sel == 2 ) ? c :
                 ( sel == 3 ) ? d : 1’bx;

endmodule
```

If input is undefined we want to propagate that information.
More advanced 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
```
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?
What happens if the case statement is not complete?

```
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 )
    q <= d;
end
dendmodule

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

always @( posedge clk )
begin
  q <= d;
end
dendmodule
Why non-blocking statements?

```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;
```

A  +1  B  +1  C
Why non-blocking statements?

```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;
```
Guideline

i. Use blocking assignments to model combinational logic within an always block ("=").

ii. Use non-blocking assignments to implement sequential logic ("<=").

iii. Do not mix blocking and non-blocking assignments in the same always block.

iv. Do not make assignments to the same variable from more than one always block.
Putting it all together: FSM for GCD

A_MUX_SEL_IN = 0
A_MUX_SEL_SUB = 1
A_MUX_SEL_B = 2

B_MUX_SEL_A = 0
B_MUX_SEL_IN = 1

A_MUX_SEL = A_MUX_SEL_IN
A_EN = 1
B_MUX_SEL = B_MUX_SEL_IN
B_EN = 1
OPERANDS_RDY = 1

if (A < B)
A_MUX_SEL = A_MUX_SEL_B
A_EN = 1
B_MUX_SEL = B_MUX_SEL_A
B_EN = 1

if (B != 0)
A_MUX_SEL = A_MUX_SEL_SUB
A_EN = 1

RESULT_VAL = 1
Putting it all together: FSM for GCD

* First make a register for saved state

```verilog
localparam WAIT = 2'd0;
localparam CALC = 2'd1;
localparam DONE = 2'd2;

reg [1:0] state_next;
wire [1:0] state;

vcRdff_pf#(2,WAIT) state_pf
(
    .clk (clk),
    .reset_p (reset),
    .d_p (state_next),
    .q_np (state)
);
```
Putting it all together: FSM for GCD

- Then local parameters for the mux control signals

```plaintext
localparam A_MUX_SEL_X   = 2'dx;
localparam A_MUX_SEL_IN  = 2'd0;
localparam A_MUX_SEL_B   = 2'd1;
localparam A_MUX_SEL_SUB = 2'd2;

localparam B_MUX_SEL_X   = 1'dx;
localparam B_MUX_SEL_IN  = 1'd0;
localparam B_MUX_SEL_A   = 1'd1;
```
Putting it all together:
FSM for GCD

Signals for each state

```
always @(*)
begin
  // Default control signals
  A_mux_sel = A_MUX_SEL_X;
  A_en = 1'b0;
  B_mux_sel = B_MUX_SEL_X;
  B_en = 1'b0;
  operands_rdy = 1'b0;
  result_val = 1'b0;

  case ( state )
    WAIT :
      begin
        A_mux_sel = A_MUX_SEL_IN;
        A_en = 1'b1;
        B_mux_sel = B_MUX_SEL_IN;
        B_en = 1'b1;
        operands_rdy = 1'b1;
        end
    CALC :
      begin
        if ( A_lt_B )
          begin
            A_mux_sel = A_MUX_SEL_B;
            A_en = 1'b1;
            B_mux_sel = B_MUX_SEL_A;
            B_en = 1'b1;
            end
        else if ( !B_zero )
          begin
            A_mux_sel = A_MUX_SEL_SUB;
            A_en = 1'b1;
            end
        end
    DONE :
      begin
        result_val = 1'b1;
        end
  endcase
```

CALC :
```
begin
  if ( A_lt_B )
    begin
      A_mux_sel = A_MUX_SEL_B;
      A_en = 1'b1;
      B_mux_sel = B_MUX_SEL_A;
      B_en = 1'b1;
    end
  else if ( !B_zero )
    begin
      A_mux_sel = A_MUX_SEL_SUB;
      A_en = 1'b1;
    end
end
```
Putting it all together: FSM for GCD

* Then state transitions

```verbatim
always @(*)
begin

  // Default is to stay in the same state
  state_next = state;

  case ( state )
  
  WAIT :
    if ( operands_val )
      state_next = CALC;
  
  CALC :
    if ( !A_lt_B && B_zero )
      state_next = DONE;
  
  DONE :
    if ( result_rdy )
      state_next = WAIT;
  
  endcase

end
```