Lecture 15: System Modeling and Verilog

Deming Chen
Outline

- Modeling Digital Systems
- Introduction to Verilog HDL
- Use of Verilog HDL in Synthesis

Reading
- Appendix A
Model Digital Systems

- Digital system: Any digital circuit that processes or stores information in digital form: gates to functional units.
- Model represents only relevant information and abstracts away irrelevant data.
- Model needed to
  - Specify requirements
  - Communicate understanding of a system to a user
  - Allow testing of design through simulation
  - Allow formal verification
  - Allow automated synthesis
Level of Modeling

- Behavioral Domain
  - Systems
  - Operating Systems

- Architectural Abstraction
  - RTL Abstraction
  - Logic Abstraction
  - Circuit Abstraction

- Structural Domain
  - Processor, SOC
  - Hardware Modules

- Physical Domain
  - Boards, Systems
  - Chips
  - Modules
  - Cells
  - Rectangles

Gajski-Kuhn Y chart
Introduction to Verilog

- Invented around 1983–84 by Gateway Design Automation
- Gateway was bought by Cadence in 1989, and Verilog HDL became a “public” language
- Standard is *Open Verilog International (OVI) IEEE 1364*
- Verilog-based synthesis tool introduced by Synopsys in 1987
- Verilog HDL is NOT a programming language, although it looks like a mixture of C, PASCAL, and ADA constructs
- In contrast to these languages, Verilog HDL *models, represents, and simulates* the functional behavior and timing of hardware
- Supported logic levels are 0, 1, *x (undefined)*, and *z (floating, i.e.: not driven by 0 or 1)*
Verilog HDL Modeling

- Structural models
  - Any hierarchy based on *modules* containing gates, primitives, or instances of other modules
  - Modules are the basic unit of hardware description
  - Modules can not be defined within other modules

- Behavioral models
  - Behavioral description on all levels from the Algorithmic level and Register-Transfer level (RTL), down to Boolean Equations

- Widely used for RTL synthesis (Verilog RTL $\Rightarrow$ Verilog gate netlist)
- Verilog supports all design steps down to gate level simulations using layout-extracted timing information
Built-in Logic Gates (Verilog Primitives)

- and, or, xor, xnor, nand, nor, not, buf
  - and and1 (out1, in1, in2, in3);
  - xor xor1 (out2, in4, in5);
  - not not1 (out3, in6);
- bufif1, bufif0, notif1, notif0 (tristate buffers and inverters)
  - bufif1 b1 (out, in, ctrl)
Basic Gates

Syntax

GATE (drive_strength) # (delays)
instance_name1 (output, input_1, input_2,..., input_N)

Delays is

# (rise, fall) or
# rise_and_fall or
# (rise_and_fall)

```plaintext
and c1 (o, a, b, c, d); // 4-input AND called c1 and
c2 (p, f g);
or #(4, 3) ig (o, a, b); /* or gate called ig (instance name);
          rise time = 4, fall time = 3 */
xor #(5) xor1 (a, b, c); // a = b XOR c after 5 time units
```
Module Example

- Typically inputs are **wire** since their data is latched outside the module. Outputs are type **reg** if their signals were stored inside an **always** or **initial** block.

```verilog
module add_sub(add, in1, in2, oot);
    input add;  // defaults to wire
    input [7:0] in1, in2; wire in1, in2;
    output [7:0] oot; reg oot;
    ...
    endmodule
```

Example
Modeling Combinational Logic

- Example: Half-Adder

**Structural model**

```verilog
module halfadder (sum, cout, a, b);
    output sum, cout;
    input a, b;
    wire cbar;
    xor l1(sum, a, b);
    nand l2(cbar, a, b);
    not l3(cout, cbar);
endmodule
```

**Behavioral model**

```verilog
module halfadder (sum, cout, a, b);
    output sum, cout;
    input a, b;
    assign cout = a & b;
    assign sum = a^b;
endmodule
```
Modeling Circuit Timing

- Example: NAND-Gate using Gate-Propagation Delay

```
timescale 1ns/100ps;
module nand2 (y, a, b);
    output y;
    input a, b;
    nand #1 l1 (y, a, b);
endmodule
```

- `timescale` is read by simulator and applied to all following modules until next timescale directive occurs
  - 1ns: Unit used for delay specifications
    - delay for nand l1 is 1 ns
  - 100ps: Timestep for the simulator (i.e., timing precision of simulator)
Modeling Wire Delays

- Example: *Long* wire connected to gate output

  ```
  module some_function (out, in1, in2);
    parameter wire_delay = 2;
    parameter gate_delay = 1;
    output out;
    input in1, in2;
    wire #wire_delay out;
    some_gate #gate_delay I1 (out, in1, in2);
  Endmodule
  ```

- Wire delays can be modeled independently from gate delays
  - A *wire* does not store its value but must be driven by a continuous assignment statement or by connecting it to the output of a gate or module.

- A *parameter* defines a constant that can be set when you instantiate a *module*. 
## Constants

<table>
<thead>
<tr>
<th>Number</th>
<th># Bits</th>
<th>Base</th>
<th>Decimal Equivalent</th>
<th>Stored</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>3'b101</code></td>
<td>3</td>
<td>Binary</td>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td><code>b11</code></td>
<td>unsized</td>
<td>Binary</td>
<td>3</td>
<td>000000..00011</td>
</tr>
<tr>
<td><code>8'b11</code></td>
<td>8</td>
<td>Binary</td>
<td>3</td>
<td>000000011</td>
</tr>
<tr>
<td><code>8'b1010_1011</code></td>
<td>8</td>
<td>Binary</td>
<td>171</td>
<td>10101011</td>
</tr>
<tr>
<td><code>3'd6</code></td>
<td>3</td>
<td>Decimal</td>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td><code>6'o42</code></td>
<td>6</td>
<td>Octal</td>
<td>34</td>
<td>100010</td>
</tr>
<tr>
<td><code>8'hAB</code></td>
<td>8</td>
<td>Hexadecimal</td>
<td>171</td>
<td>10101011</td>
</tr>
<tr>
<td>42</td>
<td>unsized</td>
<td>Decimal</td>
<td>42</td>
<td>0000...00101010</td>
</tr>
</tbody>
</table>
Blocking vs. Nonblocking Assignments

- Verilog simulators work sequentially, while hardware is inherently parallel.
- To model closer to hardware, nonblocking assignments can be used that work similar to `fork – join` in C language.
- Blocking assignments (=) are done sequentially in the order the statements are written. A second assignment is not started until the preceding one is complete.
- Nonblocking assignments (<=), which follow each other in the code, are done in parallel.

Blocking

```verbatim
/* grab inputs now, deliver ans. later. */

a=1; b=2; c=3;
#5 a = b + c;   // wait for 5 units, and execute a = b + c = 5.
d = a;         // time continues from last line, d = 5 = b+c at t = 5.
```

Non-Blocking

```verbatim
#3 b <= a;   /* grab a at t = 0; Deliver b at t = 3.
#6 x <= b + c; // grab b+c at t = 0, wait and assign x at t = 6.
/* x is unaffected by b’s change. */
```
For synthesis
- One must not mix "\(<=\)" or "\(=\)" in the same procedure.
- "\(<=\)" best mimics what physical flip-flops do; use it for "always @ (posedge clk ..)" type procedures.
- "\(=\)" best corresponds to what C/C++ code would do; use it for combinational procedures.

For simulation, no such constraints.
Modeling Sequential Logic

- Example: Behavioral description of D-Flip-Flop

```verilog
module dff_rs (q, qbar, d, clk, set, reset);
    output q, qbar;
    reg q;
    input d, clk, set, reset;
    wire qbar;

    assign qbar = !q;

    always @(posedge clk)
    begin
        if (reset == 1'b1) q <= 1'b0; // i.e., q = 0
        else if (set == 1'b1) q <= 1'b1;
        else q <= d;
    end
endmodule
```
Translation of Verilog Constructs

- always blocks are re-evaluated only when signals in the header (called a sensitivity list) change.
- always statement can infer sequential and combinational logic
  
  - Positive edge-triggered flip-flop inferred
    ```verilog
    always @(posedge clk)
    q <= d;
    ```
  
  - Level-sensitive latch inferred
    ```verilog
    always @(clk or d)
    if (clk) q <= d;
    ```
  
  - Combinational logic inferred
    ```verilog
    always @(a or b or c_in)
    {c_out, sum} <= a + b + c_in;
    ```
Translation of Verilog Constructs

- assign statements are translated to equivalent combinational logic
  - assign out = (a & b) | c;
  - translated to a 2-input *and* gate feeding a 2-input *or* gate

- *if-else* statement translated to a mux
- *case* statement translated to a mux

- *for* loops used to build cascaded combinational logic
  c = c_in;
  for (i = 0; i <= 7; i = i + 1)
    {c, sum[i]} = a[i] + b[i] + c; // 8-bit ripple adder
  c_out = c;
Example: Generic Parameterized Adder

module adder (cout, sum, a, b, cin);
  parameter width=4;
  output[width-1:0] sum;
  output cout;
  reg[width-1:0] sum;
  reg[width-1:0] c;
  input[width-1:0] a, b;
  input cin;
  wire cout;
  integer n;

  assign cout = c[width-1];

  always @(a or b or cin)
    for (n = 0; n <= (width-1); n = n+1)
      if (n == 0) {c[0], sum[0]} = a[0] + b[0] + cin;
      else {c[n], sum[n]} = a[n] + b[n] + c[n-1];
endmodule
Example: Decoder

module decoder ( input [2:0] a,                           // how does it look like?
           output reg [7:0] y);

// a 3:8 decoder
always @(*)
  case (a)
    3'b000: y <= 8'b00000001;
    3'b001: y <= 8'b00000010;
    3'b010: y <= 8'b00000100;
    3'b011: y <= 8'b00001000;
    3'b100: y <= 8'b00010000;
    3'b101: y <= 8'b00100000;
    3'b110: y <= 8'b01000000;
    3'b111: y <= 8'b10000000;
  endcase
endmodule
Initial Block

- Build a test sequence for verification
- Sequential evaluation
- LHS use register/integer

- Delays

  initial // draw the timing graph
  begin
  #1 reset = 0;
  # 4 reset = 1;
  # 5 reset = 0;
  end
Verifying a Verilog Model Through Simulation

- **Verilog Stimulus for the instance** `toplevel_module`

```
'timescale 1ns/100ps
reg in1, in2, clk;
wire out;
toplevel_module I1 (clk, in1, in2, out);

initial // Define clk
    begin
        clk = 1'b0;
        forever #50 clk = ~clk;
    end

initial
    begin
        $monitor ($time, clk, in1, in2, out);
        in1 = 1'b0;
        in2 = 1'b0;
        #125 in1 = 1'b1;
        @(negedge clk) in2 = 1'b1;
        #200 $finish;
    end
```
Continue from previous slide

If toplevel module represents `always @(posedge clk) out = !(a&b)`, the generated output is:
Use of Verilog HDL in Synthesis

- Synthesis usually refers to RTL synthesis, i.e., the transformation from RTL to gate level.

- While Verilog HDL is a powerful language (even with object-oriented constructs as in C++), only a mostly structural subset of Verilog HDL is synthesizable.
  - *If you can’t imagine hardware behind your code, you can’t synthesize it.*

- Parentheses should be used to group logic.
- Need to provide cycle-by-cycle RTL description.
  - Delays are ignored by synthesis tools.
Verilog Modeling Tips for Logic Synthesis

- RTL specification should be as close as possible to desired structure
- Meaningful names should be used for signals and variables
- Mixing of positive and negative edge-triggered flip-flops should be avoided
  - Result in buffers and inverters in clock tree, which contribute to clock skew
- Avoid multiple assignments to the same variable
- Define if-else and case statements explicitly
  - Latches may be inferred if all conditions are not specified.
Inferring Latches in Combinational Logic

module wrong (out, in1, in2);
  output out;
  reg out;
  input in1, in2;
  always @(in1 or in2)
    case ({in1,in2})
      2'b00 : out = 0;
      2'b01 : out = 1;
      2'b10 : out = 1;
      default: out=x;
    endcase
  endmodule

module right (out, in1, in2);
  output out;
  reg out;
  input in1, in2;
  always @(in1 or in2)
    case ({in1,in2})
      2'b00 : out=0;
      2'b01 : out=1;
      2'b10 : out=1;
      default: out=x;
    endcase
  endmodule

Module wrong infers one latch
Finite State Machine Design

- Finite-state machines (FSMs) are either Moore- or Mealy- machines.
- FSMs can be described using a separate combinational part for output decoding and generation of next state logic and a sequential part to generate present state logic.
- Consider the following Mealy machine (input/output):
  - Mealy machine: output depends on present state and input.
Example of Synthesizable Mealy Machine

- It is always important to verify equivalence between different levels of the design.

```verilog
module fsm (clk, reset, in, out);
input clk, reset, in;
output out;
reg out;
reg[1:0] present_state, next_state;
parameter[1:0] S0=2'b00, S1=2'b01, S2=2'b10, S3=2'b11;

always @(posedge clk)
begin
  if (reset == 1) present_state = S0;
  else present_state = next_state;
end
```
Continued

always @(present_state or in)
  begin
    case(present_state)
      S0: if (in == 1) next_state = S1;
          else next_state = S0;
      S1: if (in == 1) next_state = S2;
          else next_state = S1;
      S2: if (in == 1) next_state = S3;
          else next_state = S2;
      S3: if (in == 1) next_state = S0;
          else next_state = S3;
      default: next_state = S0;
    endcase
  end
Continued: output decoding

always @(present_state or next_state)
    begin
        if ((present_state == S3) && (next_state == S0))
            out=1;
        else out=0;
    end
endmodule
controller.v: complete your logic

module controller(
    i,
    // opcode
    a,b,select_a_hi,select_b_hi,
    // decoding of register addresses
    f,c,p,g_lo,p_lo,ovr,z,
    y_tri,y_data,oe,
    // generation of ALU outputs
    ram0,ram3,
    q0,q3,q0_data,q3_data
    // tristate control of y bus
    //,reg_wr
    // tristate control of RAM shifter
    // tristate control of Q shifter
);
// add additional signals for your design here
Controller.v: complete your logic

// define I/O for synthesized control
input [8:0] i;
input [3:0] a, b;
output [15:0] select_a_hi, select_b_hi;
inout [3:0] f, c, p;
output g_lo, p_lo, ovr, z;
inout [3:0] y_tri;
inout [3:0] y_data;
inout oe;
inout ram0, ram3, q0, q3;
input q0_data, q3_data;
//output reg_wr

// define additional I/Os for your design
Continued: MP2. Controller

// named internal wires carry reusable subexpressions
wire shift_left, shift_right;

// "assign" statements give us algebraic expressions
assign select_a_hi = 16'h0001 << a;
assign select_b_hi = 16'h0001 << b;
assign shift_left = i[8] & i[7];
assign shift_right = i[8] & ~ i[7];
Continued: MP2. Controller

// simpler functionality is better implemented directly in logic gates
buf calcg( g_lo, ~c[3] ); // glitchy with lookahead carry propagation, but shouldn't matter for us :v)
nand calcp( p_lo, p[3], p[2], p[1], p[0] );
xor calcovr( ovr, c[3], c[2] );
nor calczero( z, f[3], f[2], f[1], f[0] );

bufif1 drvy3( y_tri[3], y_data[3], oe );
bufif1 drvy2( y_tri[2], y_data[2], oe );
bufif1 drvy1( y_tri[1], y_data[1], oe );
bufif1 drvy0( y_tri[0], y_data[0], oe );
bufif1 drvraml( ram3, f[3], shift_left );
bufif1 drvramr( ram0, f[0], shift_right );
bufif1 drvqshl( q3, q3_data, shift_left );
bufif1 drvqshr( q0, q0_data, shift_right );
Controller.v: complete your logic

// add your control signals here...
//assign reg_wr = ;
//end

dendmodule
MP2. AM2901

Am2901.v: Connect controller + datapath

**Am2901.v**

module Am2901(cp,i,cin,cout,g_lo,p_lo,ovr,z,f3,d,y,a,b,ram0,ram3,q0,q3,oe);

// define chip pins
input cp; // clock
input [8:0] i; // opcode
...

// define signals between control and datapath
wire [15:0] select_a_hi, select_b_hi;
...

// add your signals (wires)

**controller.v**

module controller(
    i, // opcode
    ...
    // added signals
);

// define I/O for synthesized control
input [8:0] i;
...

// defined additional I/Os for your design
...

demodule
Continued: MP2. AM2901

Am2901.v: Connect controller + datapath

Am2901.v

module Am2901(...);

controller control ( // instantiate either RTL (MP2) or synthesized logic (MP3)
    .i(i),     // opcode
    ...       // add your decoded signals
);

controller.v

module controller(
    i,         // opcode
    ...
    // added signals
);

// define I/O for synthesized control
input [8:0] i;

...  // defined additional I/Os for your design

...  

endmodule
Am2901.v: Connect controller + datapath

Am2901.v

module Am2901(...);

datapath data ( // instantiate your design
schematics from Virtuoso
.cp(cp),  // clock
...
// add your decoded signals
);

endmodule

controller.v

module controller(
    i,        // opcode
    ...
// added signals
);

// define I/O for synthesized control
input [8:0] i;

...

// defined additional I/Os for your
// design

...

endmodule