Logic Level Modeling (Ch. 6)

Prof. Jakub Szefer
Dept. of Electrical Engineering, Yale University

EENG 428 / ENAS 968
Cloud FPGA
Logic Level Modeling

This lecture is mostly based on contents of Chapter 6, from “The Verilog Hardware Description Language” book [1], 5th edition. Example figures and (modified) code are from the textbook unless otherwise specified.

Topics covered:
• Logic gates and nets
• Four logic level values
• Continuous assignment
• Logic delay modeling
• Specifying time units
Logic Level Modeling

• Behavioral modeling focuses on describing behavior and functionality of a circuit
  • Behavioral code is written almost as a software-like function

• **Logic level modeling is used to model the logical structure of a design**
  • Specify ports (inputs and outputs)
  • Any submodule instances
  • Connection between the submodules
  • Logical functions

• Levels of logic modeling
  1) Gate level – describe design in terms of interconnection of logical gates
     • Use gate-level primitives
  2) Continuous assignment statements – describe designs using Boolean algebra-like expressions
     • Use `assign` statements
  3) Transistor switch level – describe at level of MOS and CMOS transistors

Note at the gate level, the actual implementation in FPGA will not use exactly the gates used in Verilog – all code is compiled to use Look-Up Tables (LUTs) in the FPGAs.

FPGA and ASIC vendors provide modules such as LUT, DFF, etc. to let users exactly specify the hardware that will be implemented.
Logic Gates and Nets

- Describing and modeling design at the logic level can be done using Verilog’s built-in logic gate and switch primitives:

<table>
<thead>
<tr>
<th>n_input gates</th>
<th>n_output gates</th>
<th>tristate gates</th>
<th>pull gates</th>
<th>MOS switches</th>
<th>bidirectional switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>buf</td>
<td>bufif0</td>
<td>pullup</td>
<td>nmos</td>
<td>tran</td>
</tr>
<tr>
<td>nand</td>
<td>not</td>
<td>bufif1</td>
<td>pulldown</td>
<td>pmos</td>
<td>tranf0</td>
</tr>
<tr>
<td>nor</td>
<td>notif0</td>
<td></td>
<td>cmos</td>
<td>tranf1</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>notif1</td>
<td></td>
<td>rmos</td>
<td>rtran</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td></td>
<td></td>
<td>rpmos</td>
<td>rtranf0</td>
<td></td>
</tr>
<tr>
<td>xnor</td>
<td></td>
<td></td>
<td>rcmos</td>
<td>rtranf1</td>
<td></td>
</tr>
</tbody>
</table>

- The gates (and switch level primitives) can be interconnected using **nets**
  - A **net** is a Verilog type, pretty much a representation of a physical wire
  - **nets** do not store values or charges, except a **tirreg** type of a **net**
Logic Gates and Nets

- Textbook example of full adder module
  - Each gate is interconnected using **nets**
  - Inputs and outputs are wire **nets** by default
  - Undefined **nets** will be made into wires by default

```verilog
module fullAdder
(output cOut, sum,
 input aIn, bIn, cIn);

wire x2;
	nand (x2, aIn, bIn),
	(cOut, x2, x8);
	xnor (x9, x5, x6);

nor (x5, x1, x3),
	(x1, aIn, bIn);

or (x8, x1, x7);

not (sum, x9),
	(x3, x2),
	(x6, x4),
	(x4, cIn),
	(x7, x6);

endmodule
```

Common problem is undefined wires that become 1-bit wires by default and cause logic to work incorrectly!

Can use `default_nettype none` to disable automatic definition of wire nets
Logic Gate Instance Options

Gates (and modules, see later slides) can be instantiated with a number of options

- **Instance names** – are used in hierarchical design, a good practice is to assign meaningful and unique instance names to all modules and gates

- **Gate delay** – is used to quantify the number of time units from when any input changes to when output changes, this is the propagation delay

- **Drive (and charge) strength** – used for modeling physical behavior of wires
  - The drive strengths are: supply, strong, pull, weak, and highz (all nets except trireg)
  - The charge strengths are: large, medium and small strengths (for trireg only)
  - Higher drive strength can supply the needed current faster when switching

Drive strength details from: [http://verilog.renerta.com/source/vrg00047.htm](http://verilog.renerta.com/source/vrg00047.htm)
Four Logic Level Values

• The values that may be driven onto a net are:
  0 – a logic zero, or FALSE condition
  1 – a logic one, or TRUE condition
  x – an unknown logic value (any of 0, 1, or in a state of change)
  z – a high-impedance condition

• Gate truth tables with respect to the possible logic levels (see textbook’s Appendix D):

<table>
<thead>
<tr>
<th>AND</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOT</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bufi0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>T</td>
</tr>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

L indicates 0 or z; H indicates 1 or z
Different net types

- **nets** are used to model electrical connections
  - **nets** store no charges and are just a connection
  - Except **trireg** that models wires as capacitors that store charge
- Many net types are supported in Verilog:
  - wire, tri, tri1, supply0, wand, triand, tri0, supply1, wor, and trior

<table>
<thead>
<tr>
<th>Net Type</th>
<th>Modeling Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>wire and tri</td>
<td>Used to model connections with no logic function. Only difference is in the name. Use appropriate name for readability.</td>
</tr>
<tr>
<td>wand, wor, triand, trior</td>
<td>Used to model the wired logic functions. Only difference between wire and tri version of the same logic function is in the name.</td>
</tr>
<tr>
<td>tri0, tri1</td>
<td>Used to model connections with a resistive pull to the given supply</td>
</tr>
<tr>
<td>supply0, supply1</td>
<td>Used to model the connection to a power supply</td>
</tr>
<tr>
<td>trireg</td>
<td>Used to model charge storage on a net. See Chapter 10.</td>
</tr>
</tbody>
</table>
Logic Level Modeling Example

- Textbook of Hamming encoder and decoder and a simple testbench
  - Hamming codes can be used to detect and correct errors in transmitted data
  - Example uses code that can correct 1 error

- Example test module:
module testHam;

reg [1:8] original;
wire [1:8] regenerated;
wire [1:12] encoded, messedUp;

integer seed;
initial begin
    seed = 1;
    forever begin
        original = $random (seed);
        #1 $display("original=%h, encoded=%h, messed=%h, regen=%h", original, encoded, messedUp, regenerated);
    end
end

hamEncode hIn (original, encoded);
hamDecode hOut (messedUp, regenerated);

assign messedUp = encoded ^ 12'b 0000_0010_0000;
endmodule
Logic Level Modeling Example

module hamEncode
(input [1:8] vIn,
output [1:12] valueOut
);

wire h1, h2, h4, h8;
xor (h1, vIn[1], vIn[2], vIn[4], vIn[5], vIn[7]),
(h2, vIn[1], vIn[3], vIn[4], vIn[6], vIn[7]),
(h4, vIn[2], vIn[3], vIn[4], vIn[8]),
(h8, vIn[5], vIn[6], vIn[7], vIn[8]);

assign valueOut = {h1, h2, vIn[1], h4, vIn[2:4], h8, vIn[5:8]};
endmodule
module hamDecode
( input [1:12] vin,
  output [1:8] valueOut);

wire c1, c2, c4, c8;
wire [1:8] bitFlippers;

xor (c1, vin[1], vin[3], vin[5], vin[7], vin[9], vin[11]), (c2, vin[2], vin[3], vin[6], vin[7], vin[10], vin[11]), (c4, vin[4], vin[5], vin[6], vin[7], vin[12]), (c8, vin[8], vin[9], vin[10], vin[11], vin[12]);

demux mux1 (bitFlippers, c1, c2, c4, c8, 1'b1);
xor8 x1 (valueOut, bitFlippers, {vin[3], vin[5], vin[6], vin[7], vin[9], vin[10], vin[11], vin[12]});

endmodule
Continuous Assignment

• A different way to describe logic is using `assign` continuous assignment statements

```verilog
module oneBitFullAdder
  ( output cOut, sum,
    input aIn, bIn, cIn
  );
  assign sum = aIn ^ bIn ^ cIn,
             cOut = (aIn & bIn) | (bIn & cIn) | (aIn & cIn);
endmodule
```

• Continuous assignment can specify delays

```verilog
module combiningDelays
  ( input a, b,
    output c
  );
  wire #10 c;
  assign #5 c = ~a;
  assign #3 c = ~b;
endmodule
```

Boolean algebra-like expression for each of two assign statements

Functions, but not tasks, can be used in assign statements

```verilog
module multiplexor
  ( input a, b, c, d,
    input [1:0] select,
    output e
  );
  assign e = mux (a, b, c, d, select);
endmodule
```

Use default case to capture possible x on the inputs (not needed for synthesizable code, no x)

```verilog
function mux
  ( input a, b, c, d,
    input [1:0] select
  );
  case (select)
    2'b00: mux = a;
    2'b01: mux = b;
    2'b10: mux = c;
    2'b11: mux = d;
    default: mux = 'bx;
  endcase
endfunction
```
Continuous Assignment

- Continuous assignment can be also used with *inouts* to specify high-impedance output

```verilog
module Memory_64Kx8
(input [7:0] addrBus,
inout [15:0] dataBus,
input we, re, clock
);

reg [7:0] out;
reg [7:0] Mem [65535:0];

assign dataBus = (re) ? out: 16'b0;
always @(negedge re or addrBus)
    out = Mem[addrBus];
always @(posedge clock)
    if (we == 0)
        Mem[addrBus] <= dataBus;
endmodule
```

Use `inout` to model a shared data bus where master can drive the bus (writes) or reads the bus (reads).

Assign output to high-impedance when not sending data from memory.
• Most of Verilog code written uses both structural and behavioral descriptions mixed together to describe the design:
  • Use structural for module interconnection, gate-level modeling of certain modules that need specific gate-level implementation
  • Use behavioral for describing modules where specific hardware implementation is less important

**Structural description**

```verilog
module sbus;

parameter Tclock = 20,
    Asize = 5,
    Dsize = 16,
    Msize = 32;

reg clock;
wire rw;
wire [Asize-1:0] addr;
wire [Dsize-1:0] data;

master #(Asize, Dsize) m1 (rw, addr, data, clock);
slave #(Asize, Dsize, Msize) s1 (rw, addr, data, clock);

initial begin
    clock = 0;
    $monitor("rw=%d, data=%d, addr=%d at time %d", rw, data, addr, $time);
end

always
    #Tclock clock = !clock;

endmodule
```

**Behavioral description**

Note again, that even with gate-level specification, actual hardware will be different (e.g. using LUTs on FPGAs)
Logic Delay Modeling

- Gate and net specification can include information about delays
  - Gates can have specified propagation delay for transition to 1, transition to 0, and transition to z, resulting in up to three delay values used with `#` operator:

  `#(d1, d2, d3)`

- Delays are used by simulators to determine when to update the values on output of the gates

<table>
<thead>
<tr>
<th>From value</th>
<th>To value</th>
<th>2 Delays specified</th>
<th>3 Delays specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>d1</td>
<td>d1</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>min(d1, d2)</td>
<td>min(d1, d2, d3)</td>
</tr>
<tr>
<td>0</td>
<td>z</td>
<td>min(d1, d2)</td>
<td>d3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>d2</td>
<td>d2</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>min(d1, d2)</td>
<td>min(d1, d2, d3)</td>
</tr>
<tr>
<td>1</td>
<td>z</td>
<td>min(d1, d2)</td>
<td>d3</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>d2</td>
<td>d2</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>d1</td>
<td>d1</td>
</tr>
<tr>
<td>x</td>
<td>z</td>
<td>min(d1, d2)</td>
<td>d3</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>d2</td>
<td>d2</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
<td>d1</td>
<td>d1</td>
</tr>
<tr>
<td>z</td>
<td>x</td>
<td>min(d1, d2)</td>
<td>min(d1, d2, d3)</td>
</tr>
</tbody>
</table>
Verilog allows for three values to be specified for each of the rising, falling, and turn-off delays

- A three-valued delay specification:
  
  `#(d1, d2, d3)`

- Can be expanded to:
  
  `#(d1_min: d1_typ: d1_max, d2_min: d2_typ: d2_max, d3_min: d3_typ, d3_max)`
Logic Delay Modeling

Textbook example of modeling logic gate delays in a tri-state latch:

```verilog
module triStateLatch
  ( output qOut, nQOut,
    input clock, data, enable
  );

  tri qOut, nQOut;
  not #5 (ndata, data);
  nand #(3,5) d(wa, data, clock),
    nd(wb, ndata, clock);
  nand #(12, 15) qQ(q, qN, wa),
    nQQ(nq, q, wb);
  bufif1 #(3, 7, 13) qDrive (qOut, q, enable),
    nQDrive(nQOut, nq, enable);
endmodule
```
Delays Across Modules

- Verilog allows for modeling delays across a whole module using the `specify` block.
  - `source => destination = (delays)` – specifies the delays, same order as for wires: transition to 1, transition to 0.
  - `(a, b => c, d) = (delays)` – combines multiple delay specifications into one: a to c, a to d, b to c, and c to d.

```verilog
module dEdgeFF
(input clock, d, clear, preset,
 output q
);

specify
specparam tRiseClkQ = 100,
tFallClkQ = 120,
tRiseCtlQ = 50,
tFallCtlQ = 60;

(clock => q) = (tRiseClkQ, tFallClkQ);
(clear, preset => q) = (tRiseCtlQ, tFallCtlQ);
endspecify

// Code of the module goes here...
endmodule
```
Specifying Time Units

- Verilog simulator works in term of time units, each # delay is in time units
  - To assign specific time unit magnitude to delays use: `timescale <time_unit> / <time_precision>`
  - Example: `timescale 10 ns / 1 ns`

- The precision is used to quantify how fine-grained the simulator keeps track of time
Cycle-Accurate Specification (Ch. 7)

Prof. Jakub Szefer
Dept. of Electrical Engineering, Yale University

EENG 428 / ENAS 968
Cloud FPGA
Cycle-Accurate Specification

This lecture is mostly based on contents of Chapter 7, from “The Verilog Hardware Description Language” book [1], 5th edition. Example figures and (modified) code are from the textbook unless otherwise specified.

Topics covered:

• Clock events
• Input and output sets of always blocks
• Reset function
• Mealy and Moore state machines
• Behavioral synthesis
Cycle-Accurate Specification
Cycle-Accurate Specification

• When using cycle-accurate specification, also called scheduled behavior, the system is described using behavioral (always) statements

• One of the most common ways to describe synchronous digital systems
  • All the state (registers) get updated on a clock edge

• Cycle-accurate specification specifies what happens in each clock cycle as the design runs
  • Actual implementation of the logic is left to the synthesis tools when using behavioral description
  • Or can be specified using logic-level description
  • Typically mix both types

• Cycle-accurate specification basically specifies that state machines that define the operation of the system and inputs and outputs, and computations done by the state machines
Clock Events and Cycle-Accurate Specification

- Cycle-accurate specification is divided into clock events
  - Each clock event defines a start of a new state
  - Behavioral description after each clock event defines the behavior in that state

Sample state machine and state diagram from textbook:
Input and Output Sets of `always` Blocks

• **Internal Register Set** – the set of all registers used on the left-hand side of statements in the `always` block, that are not used by other `always` blocks

• **Input Set** – the set of all the signals on the right-hand side of statements in the `always` block, that are not in the internal register set

• **Output Set** – the set of all signals on the left-hand side of statements in the `always` block, that are not in the input set

```verilog
module inOutExample
(input [7:0] r, s,
input clock,
output reg [7:0] qout );

reg [7:0] q;
always begin
  @ (posedge clock)
  q <= r + s;
  @ (posedge clock)
  qout <= q + qout;
end
endmodule
```
Cycle-Accurate Behavior

- When working with cycle-accurate specification, all the states and registers are updated on the clock edge
  - The `always` block specifies which edge: `posedge` or `negedge`
  - Can mix both, but typically just use one or the other

![Diagram showing state transitions and logic expressions for cycle-accurate behavior](image)

Good practice to update each output set register on each clock, when no change is needed just do `qout <= qout`
Maintaining Cycle-Accurate Specification

• Different realizations of state machines are possible based on same cycle-accurate specification code
  • Synthesis tools can attempt to optimize the design while maintaining the same behavior
  • Level of possible optimization depends on the tools and the “effort” level for the tool

```verilog
module sampleModule
  ( input [7:0] i, j, k,
    output reg [7:0] f, h
  );

  reg [7:0] g, q, r, s;
  always begin
    // ...
    @ (posedge clock);
    f <= i + j;
    g = j * 23;
    @ (posedge clock);
    h <= f + k;
    @ (posedge clock);
    f <= f * g;
    q = r * s;
    // ...
  end
endmodule
```

Possible to reduce design size by using only one multiplier in each cycle.
Reset Function

- A reset function can be used to initialize the state into a known state
  - Can be achieved with reset logic in an `initial` statement, only used for simulation
  - Can be achieved with `posedge` or `negedge` of reset signal in `always` blocks
Mealy and Moore State Machines

- Mealy state machine – outputs depend on both current state and the inputs
- Moore state machine – outputs depend only on the current state

- Mealy vs. Moore:
  - For Moore machines, registers in output set are not updated with input set registers, but only depend on internal register set
  - For Mealy machines, there are usually fewer states, but logic may be more complex

Moore machine for a FIR (finite impulse response) filter

Mealy machine for a FIR filter
Mealy and Moore State Machines

Textbook examples of FIR filters as Moore and Mealy machines:

```
module firFiltMealy
    input clock, reset,
    input [7:0] x,
    output reg [7:0] y;
    reg [7:0] coef_array [7:0];
    reg [7:0] x_array [7:0];
    reg [7:0] acc;
    reg [2:0] index, start_pos;

    initial forever
    @(negedge reset)
    begin
        disable firmain;
        start_pos = 0;
    end

    always begin firmain
        wait (reset);
    end

    @(posedge clock); // State A;
    x_array[start_pos] = x;
    acc = x * coef_array[start_pos];
    index = start_pos + 1;

    begin loop1
        forever
        begin
            @(posedge clock); // State B;
            acc = acc + x_array[index] * coef_array[index];
            index = index + 1;
            if (index == start_pos) disable loop1;
        end
    end // loop1

    y <= acc;
    start_pos = start_pos + 1;
endmodule
```

```
module firFilt
    input clock, reset,
    input [7:0] x,
    output reg [7:0] y;

    reg [7:0] coef_array [7:0];
    reg [7:0] x_array [7:0];
    reg [7:0] acc;
    reg [2:0] index, start_pos;

    initial forever
    @(negedge reset)
    begin
        disable firmain;
        start_pos = 0;
        index = 0;
    end

    always begin firmain
        begin loop1
            forever
                @(posedge clock); // State 1, the only state
                if (index == start_pos)
                    begin
                        x_array[index] = x;
                        acc = x * coef_array[index];
                        index = index + 1;
                    end
                    else
                        begin
                            acc = acc + x_array[index] * coef_array[index];
                            index = index + 1;
                            if (index == start_pos) disable loop1;
                        end
                end // loop1
        end

        y <= acc;
        start_pos = start_pos + 1;
    end
endmodule
```
When designing for FPGAs, the behavioral code is synthesized into the hardware using vendor-specific tools:

- Tools have to determine how to generate hardware logic based on the behavioral description.
- Use different standard modules, e.g., multiplier, to implement user’s logic.
- Can have different optimizations and optimization effort levels.
- Target area, power, or performance.

E.g. Xilinx or Intel/Altera tools.
References