Introduction to Verilog design

Lecture 2

Design flow (from the book)

Hierarchical Design

- A chip contain many modules
- A module may contain other modules
  - no recursion
- A module uses predefined cells
- Everything is based on primitives
Hierarchical design flow

- Multi-Core system
  - SW/HW co-design, multiple cores
    (SW/HW system verification)

- Full chip
  - Processor, DSP, MPEG, etc.
    (Functionality is specified in books)

- Functional units
  - ALU, Float Point Unit, Filter, Memory Management, Scheduler, etc.
    (RTL verification can be hard)

- Block designs
  - Arithmetic blocks, common bus structures, Cross-bar switches, decoders, etc.
    (May be custom designed)

- Cell Library
  - Certify that layouts can be manufactured with high yield
    (This may be Fab dependent)

Advantages

- When working at a particular level of design, we don’t need to worry about all detail below
  - When putting a system together, you don’t need to worry about how a processor is designed
  - Optimization can be done independently at each level
- Since no detail is involved, it is easier to replace the designs with other implementations
  - You can easily change the processor to another implementation without changing the system
  - You can change the technology from 0.45 micron to 0.32 micron without changing the Verilog RTL

Primitives and design model

- Verilog includes 26 predefined models of logic gates called primitives
- Primitives are the most basic functional components that can be used to build (model) a design
  - Their functions are built into the language by internal truth table
- The output port of a primitive is the first in the list

![Diagram of a primitive and its Verilog description of a half adder.](image)
### Verilog primitives for combinational logic

<table>
<thead>
<tr>
<th>n-input</th>
<th>n-output, 3-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>buf</td>
</tr>
<tr>
<td>nand</td>
<td>not</td>
</tr>
<tr>
<td>or</td>
<td>bufif0</td>
</tr>
<tr>
<td>nor</td>
<td>bufif1</td>
</tr>
<tr>
<td>xor</td>
<td>notif0</td>
</tr>
<tr>
<td>xnor</td>
<td>notif1</td>
</tr>
</tbody>
</table>

- n-input: Any number of inputs and 1 output
- n-output: Any number of outputs and 1 input

### List of Verilog primitives

- **Gates**
  - and, nand, or, nor, xor, xnor, buf, not
- **Tri-State**
  - bufif0, bufif1, notif0, notif1
- **MOS**
  - nmos, pmos, nmnos, rpmos
- **CMOS**
  - cmos, rcmos
- **Bi-directional**
  - tran, tranif0, tranif1, rtran, rtranif0, rtranif1
- **Pull**
  - pullup, pulldown

### Standard cells vs. Primitives

- Verilog primitives are abstract building blocks used to model your design
  - They are not real
- Standard cells are real building blocks used to implement your design
  - Each has a corresponding layout
  - Each has a timing model
  - Cells are stored in a cell library
  - Cell library is NOT free; you need to buy
Verilog Behavior of D FF

```verilog
module Flip_flop (q, data_in, clk, rst);
    input data_in, clk, rst;
    output reg q;
    always @ (posedge clk)
    begin
        if (rst == 1) q = 0;
        else q = data_in;
    end
endmodule
```

Block diagram view
Comparison

- Block vs. Implementation
  - Block view
    - Specify inputs and outputs
    - No internal detail
  - Implementation
    - Include specific detail
    - One block view can have many implementations

- Structural vs. Behavior
  - Structural model
    - Specify how a function is achieved
  - Behavior model
    - Specify only the function
    - One behavior can have many structural models

Hierarchical design example

An half adder (behavior/structural)
A full adder (structural)

```verilog
module Add_full (sum, c_out, a, b, c_in);
    input a, b, c_in;
    output sum, c_out;
    wire w1, w2, w3;
    Add_half M1 (w1, w2, a, b);
    Add_half M2 (sum, w3, w1, c_in);
endmodule
```

4-bit adder

```verilog
module Add_full(sum, c_out, a, b, c_in);
    input a, b, c_in;
    output sum, c_out;
    wire w1, w2, w3;
    Add_half M1 (w1, w2, a, b);
    Add_half M2 (sum, w3, w1, c_in);
endmodule
```

Behavior model of a 4-bit adder

```verilog
module adder_4_RTL (a, b, c_in, sum, c_out);
    output [3:0] sum;
    output c_out;
    input [3:0] a, b;
    input c_in;
    assign {c_out, sum} = a + b + c_in;
endmodule
```
The concept of *Design Entry*

- **Design Entry**
  - The entry point to do your design
  - Select design data representation(s)
    - Behavior, RTL, Gate-Transistor Schematics
- **Schematic**
  - Draw your design with gates/transistors/lines
  - For high-performance blocks
- **Hardware Description Language**
  - Describe your design with Verilog/VHDL
  - Just like writing a program
  - No need to worry about structural detail
Unit level verification

- Each block or unit should be verified through extensive testbench simulation

![Diagram showing organization of a testbench for verifying a unit under test](image)

A testbench example

```verilog
module UUT_name();
  reg ...;
  wire ...;
  parameter ...
  initial ...
  initial ...
  initial ...
  always @(...)
    begin ...
    end
endmodule
```

- New version of Verilog has a lot more features to support testbench development

![Example testbench code](image)

Generic testbench design

```
module Testbench();
  reg in, out;
  wire in, out;
  integer i;
  initial ...
  initial ...
  always @(...)
    begin ...
    end
endmodule
```

- New version of Verilog has a lot more features to support testbench development
Delay model in verilog

module Add_half_1 (sum, c_out, a, b);
  input a, b;
  output sum, c_out;
  wire c_out_bar;
  xor #2 (sum, a, b);
  nand #2 (c_out_bar, a, b);
  not #1 (c_out, c_out_bar);
endmodule

These delays have no physical meaning
  – They are used only for simulation
  – The ordering of events is what being specified
  • Not the actual timing
  – Relative delays are more important than absolute numbers

Cell Delay Characterization

module and123 (o, a, b);
  input a, b;
  output o;
  and o;
  specify
    specparam
      Tpd_0_1 = 1.13, 3.09, 7.75
      Tpd_1_0 = 0.93, 2.50, 7.34
      (a => o) = (Tpd_0_1, Tpd_1_0);
      (b => o) = (Tpd_0_1, Tpd_1_0);
  endspecify
endmodule

The delays are only estimations
  – Maybe from SPICE simulation of the actual layout
  • This can only be done for very small cells

Inertial delay concept

In the upper example, the event of rising transition is de-scheduled by the simulator because the pulse width is less than the propagation delay
User-defined primitives

- Basic constructs in your design
- Contain detailed table-like description
- Use the keywords
  - `primitive`
  - `endprimitive`
- Differentiate themselves from Modules
  - In test, verification, and simulation, never goes inside primitives (black-boxes)

Instantiated in the same manner as `and`, `or`, `not`, `nor`, `nand`, etc.

- Always has one output port!
- Output port must be declared as `net` in a combinational primitive
- Output port must be declared as `reg` in a sequential primitive
- No `inout`
- Specify everything (anything left unspecified will be treated as `X`)

2-bit multiplexer

- Z input is treated as `X`!

```
primitive mux_prim (mux_out, select, a, b);
output mux_out;
input select, a, b;
table
//   select a b :  mux_out
0 0 0 :        0;
0 0 1 :        0;
0 0 x :        0;
0 1 0 :        1;
0 1 1 :        1;
0 1 x :        1;
1 0 0 :        0;
1 0 1 :        1;
1 1 0 :        0;
1 1 1 :        1;
x 0 0 :        0;
x 0 1 :        1;
x 1 0 :        1;
x 1 1 :        1;
endtable
endprimitive
```

Note: Combinations not explicitly specified will drive "X" under simulation.
Shorthand notation “?”

<table>
<thead>
<tr>
<th>select</th>
<th>a</th>
<th>b</th>
<th>mux_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>?</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>?</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Primitive notations

- 0, 1, x:
  - ? : \{0, 1, x\}
  - b : \{0, 1\}
  - - : no change
  - (vw) : v \in \{0, 1, x\} \mapsto w \in \{0, 1, x\}
  - * : \{0, 1, x\} \mapsto \{0, 1, x\}
  - r : (01)
  - f : (10)
  - p : (01), (0x), (x1), (0z), (z1)
  - n : (10), (1x), (x0), (1z), (z0)

Sequential Primitives

- n inputs
- 1 state and 1 output
- n+2 columns
  - <n inputs>, <state>, <output>=<next state>

- Two types:
  - Level sensitive
    - Achieve transparent latch (a latch can be made "invisible")
  - Edge sensitive
A Transparent Latch

primitive latch (q_out, enable, data);

output q_out;
input enable, data;
reg q_out;

// enable q_out data state: q_out/next state

<table>
<thead>
<tr>
<th>en</th>
<th>data</th>
<th>state:</th>
<th>q_out/next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>?</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>?</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
<td>?</td>
<td>-</td>
</tr>
</tbody>
</table>

// Note: '-' denotes "no change."
// The state is the residual value of q_out
endtable
// Note: Combinations not explicitly specified will drive "x" under simulation
endprimitive

Rising Edge-Sensitive Latch

primitive d_prim1 (q_out, clock, data);

output q_out;
input clock, data;
reg q_out;

// clock data state: q_out/next state

<table>
<thead>
<tr>
<th>clock</th>
<th>data</th>
<th>state:</th>
<th>q_out/next state</th>
</tr>
</thead>
</table>
| (01)  | 0    | ?      | 0;              // Rising clock edge
| (0a)  | ?    | ?      | X;              // Falling clock edge
| (0b)  | ?    | ?      | -               // Steady clock,
| (??)  | ?    | ?      | -               // Steady clock,
| (??)  | ?    | ?      | -               // Steady clock,

ignore data
endtable
endprimitive

Level-S Dominates Edge-S
Summary

- Each row specify a transition on only 1 input
- All no-effect transitions should be stated or the results will be X
- Only 0,1,x,- are allowed for outputs (no ?)
- Z is X in a primitive
- Input order = specification order

State initialization

- You can use the “initial” command
- In reality, it won’t initialize itself
- You better not count on it in real implementation

```vhdl
primitive d_prim2 (q_out, clock, data);
output q_out;
input clock, data;
reg q_out;
initial q_out = 0; // Optional -- Declares // initial value of q_out
table
//    clk data state :    q_out/next state
(01) 0 :        ? :               0; // Rising clock edge
(01) 1 :        ? :               1; 
(0?) 1 :        1 :               1; // Falling clock edge
(1?) 1 :        1 :               1; // Steady clock, ignore data
endtable
endprimitive
```