Lecture 11: RTL Design
18-322 Fall 2003
Topics Today

- Switched from your Syllabus
- Today: Design Automation, RTL Design, and This Year’s Project
- Thursday: Interconnect II and Buffering Techniques
ASIC flow with Std Cells

Simulation (HDL Sim)

Simulation / STA (HDL Sim)

Back-annotation

“Synthesizable” HDL

Synthesis (Synopsys)

Place & Route

Gate-level Verilog

Interconnect Delay

*.db

*.v

*.sdf

*.lef

© Herman Schmit, 2003
RTL Design

- **Verilog or VHDL input**
  - Multi-level Simulation Languages
    - Not synthesis specifications
  - Subsets are Synthesizable

- **Register Transfer**
  - Specify registers
  - Specify data movement between registers
  - Cycle Accurate
Your Project

- You construct gate level design from RTL
  - You can actually modify RTL design if you want, but you don’t have to
- You construct transistor level design from gate level
- You layout components and put the whole system together

“Synthesizable” HDL

Gate-level Verilog

Transistor Level Design

FULL LAYOUT
What to do?

- Envision the design and partition
  - Registers
  - Combinational logic between registers
  - Finite State Machines
  - Tri-state busses

- Describe each with a “template”
  - Simulate correctly
  - Mapped correctly by tools

- You will not have to write RTL, you need to “synthesize” it to gates
Registers

- Edge-triggered:

```vhdl
always @(posedge clock)
  begin
    Q <= D;
  end
```
Latches

always @(clock or D)
  if (clock == 1)
    Q <= D;

- Why is “D” in the event list?
Register Reset behavior

always @(posedge clk or negedge reset)
begin
    if (reset == 0)  // Reset behavior
        Q <= 0;
    else
        Q <= D;       // Non-reset beh.
end
Digression: Block and Non-blocking Assignments

- “Blocking” Assignment:
  \[ Q = D; \]

- “Non-Blocking” Assignment:
  \[ Q \leq D; \]

Why do we need different assignments:
- Blocking assignment is what C-programmers know
- Non-blocking assignment is what HW designers know
Blocking vs. Nonblocking

- **Blocking:**

```verilog
classic always @(posedge clk or
             negedge rst_n)
begin
  Q1 = D1;
  Q2 = D2;
end
```

The update to Q1 happens, then we execute the next statement.

Then this statement is evaluated, and Q2 is updated.

**So what’s the problem??**
Blocking Problem 1

- The shift register:

```verilog
always @(posedge clk or negedge rst_n)
begin
    Q2 = Q1;
    Q1 = D1;
end
```

This is actually OK, but its correctness depends on the sequence in the begin/end

```verilog
begin
    Q1 = D1;
    Q2 = Q1;
end
```

This doesn’t work!
Why order doesn’t solve the problem

module wee_ff(clk, D, Q);
  ...
  always @(posedge clk)
    begin
      Q = D;
    end
  ...
endmodule

module top;
  ...
wee_ff x(clk, D1, Q1),
  y(clk, Q1, Q2);

Two instances of this DFF. Can’t say anything about which statement gets executed first!!!
Non-blocking Assignments: Semantics

- Non-Blocking:

```verilog
always @(posedge clk or negedge rst_n)
begin
    Q1 <= D1;
    Q2 <= D2;
end
```

Value of D1 is “sampled” after current instant. Proceed to next statement.

Do the same for Q2.

Current time + $\varepsilon$:
Update Q1 and Q2

Now, sequence of eval doesn’t matter
Nonblocking Example

always @(posedge clk)
begin
  Q1 <= D1;
  Q2 <= D2;
end

always @(posedge clk)
begin
  Q2 <= D2;
  Q1 <= D1;
end

These two are equivalent!
Combinational Logic

- Three techniques:
  - Continuous assignments
  - Always blocks (skipped)
  - Continuous assignments with verilog functions

- Continuous assignments:
  ```
  assign Y = A & B & C;
  assign Y = (A == 1) ? B : (C & D);
  ```

- Problems:
  - Why do the synthesis job?
  - This can become terribly unreadable…
    - Just try to comment the above
Building a 2-to-4 bit decoder

- Continuous Assignment:

```verbatim
assign dout = (din === 2'b00) ? 4'b0001 :
    ((din === 2'b01) ? 4'b0010 :
    ((din === 2'b10) ? 4'b0100 :
    ((din === 2'b11) ? 4'b1000:
        4'bxxxx)));
```
Hey: What is the ‘===‘?

- **Familiar C comparison operator: **==
- **Verilog interprets == with ‘x’ value liberally:**
  - 1’bx == 1’b1 is true!
  - 1’bx == 1’b0 is true!
- **Do you want less liberal comparison? Use ===**
  - 1’bx === 1’b1 is false!
  - 1’bx === 1’b0 is false!
More on the ===

assign dout = (din === 2'b00) ? 4'b0001 :
    ((din === 2'b01) ? 4'b0010 :
        ((din === 2'b10) ? 4'b0100 :
            ((din === 2'b11) ? 4'b1000:
                4'bxxxx)));

How will this behave differently than:

assign dout = (din == 2'b00) ? 4'b0001 :
    ((din == 2'b01) ? 4'b0010 :
        ((din == 2'b10) ? 4'b0100 :
            ((din == 2'b11) ? 4'b1000:
                4'bxxxx)));
Verilog Functions

- A third way to do combinational logic:
  - Combine continuous assignment with Verilog function

```verilog
function [3:0] DEC;
input [1:0] AIN;
begin
  case (AIN)
    2'b00:   DEC = 4'b0001;
    2'b01:   DEC = 4'b0010;
    2'b10:   DEC = 4'b0100;
    2'b11:   DEC = 4'b1000;
    default: DEC = 4'bxxxx;
  endcase
end
endfunction
assign dout = DEC24(din);
```
Mixing Logic and Storage (registers)

- Can use logic in register statements:

```verilog
always @(posedge clock)
begin
    Q <= (A == 0) ? X : Y;
end

OR

always @(posedge clock)
    if (A == 0) Q <= X;
    else Q <= Y;
```
THE PROJECT!

- After a break...
A On-Chip Network Router

- Even now, interconnect delay can exceed cycle time!
  - Needs N cycles to get from one side to the other
- At the same time, design is increasingly complex
  - Need to re-use designs that have been previously codes and verified

- Can we somehow fix both problems?
  - Provide a way to get data across the chip in multiple cycles
  - Provide a standard interface to components

- Yup! Use a network, except that its on chip
How an On-Chip Network Works
How an On-Chip Network Works

- "Packetize" the data
  - Packet has a two parts
  - Header (the address)
  - Payload (the data)

- Send packet to nearest network router
  - We’re using a “direct network” which means each node is associated with one router
  - Router moves the data closer to the destination
Our Network

- **A Torus**
  - Mesh with wrap arounds
  - Like a donut
Torus

- Nice thing about a torus:
  - You can get anywhere with two directions (e.g. NORTH and WEST)
  - Easier to build
  - There is a way to do “wrap around” connections, but we won’t worry about that
Our Router

- “Source Routing”
  - The header has explicit, step-by-step directions to get to the destination
  - Each step has three possibilities:
    - GO NORTH
    - GO WEST
    - EXIT THE NETWORK
Our Packet

- **Header: 12 bits**
  - Six 2-bit fields
  - Gets us up to six stages from origin

- **Payload: 4 bits**
  - Really small, to minimize extraction time
  - Still captures the behavior of the payload
Signaling Protocol

- **Situation**: what if all three inputs (S, E, and Pl) send to the North output?
  - Need to have some priority, tell one to go
  - Tell the other two NOT TO GO...

- **Somehow**:
  - we need to signal that the input data has/hasn’t been accepted

- **Situation**: what if no input has sent data to a particular output?

- **Somehow**:
  - we need to signal that an output is/isn’t valid
Our Solution

- Each output port has a “valid” signal (also an output)
  - This signal says “The data currently on the data pins is valid and hasn’t yet been transferred.”

- Each output port also has a “ready” signal (an input)
  - This signal tells the output port that the data on the bus can/can’t be accepted

- A data transfer happens iff valid == 1 and ready == 1
Input Ports

- Valid is an input
- Ready is an output
- This allows a symmetric setup…
Timing Diagram

Clk 0  Clk1  Clk2  Clk3  Clk4  Clk5  Clk6  Clk7

clk

ready

valid

data

0  1  2  4
This Project: Week 1 & 2

- We give you a working spec
- Using RTL description to create gate level design
  - Use the “templates” to design logic
- Testing of gate level design
  - We give you a testbench, which evaluates whether your design is working correctly. It works on the RTL
  - You have to get the gate level design to comply
Our Working Spec
(Baseline Design)
Transforming it to Gates

- **Example from: ocnet_fifo_in:**
  ```verilog
  always @(posedge clk or negedge rst)
  begin
    if (rst == 0)
      begin
        state <= EMPTY;
        packet_o <= 26'b0;
      end
    else
      begin
        state <= next_state;
        if (ready_i === 1'b1)
          begin
            packet_o <= packet_i;
          end
        else
          begin
            packet_o <= packet_o;
          end
      end // else: !if(rst == 0)
  end // always @(posedge clk or negedge rst)
  ```
Step 1: Just look at “state”

```verilog
always @(posedge clk or negedge rst)
    begin
        if (rst == 0)
            begin
                state <= EMPTY;
            end
        else
            begin
                state <= next_state;
            end // else: !if(rst == 0)
    end // always @(posedge clk or negedge rst)
```

![Diagram of state machine with inputs `rst`, `clk`, and output `state` connected to `next_state` and `state`]
Step 2: Look at “packet_o”

```verilog
always @(posedge clk or negedge rst)
    begin
        if (rst == 0)
            begin
                packet_o <= 26'b0;
            end
        else
            begin
                if (ready_i === 1'b1)
                    packet_o <= packet_i;
                else
                    packet_o <= packet_o;
            end // else: !if(rst == 0)
    end // always @ (posedge clk or negedge rst)
```

Diagram: 
- `rst` to `packet_o`
- `clk` to `packet_o`
- `ready_i` to `packet_o`
- `packet_i` to `packet_o`
Another Example

- Example from: ocnet_fifo_in:

```verilog
function [0:0] next_state_f;
  input valid_i, ready_o, state;
  begin
    case(state)
      EMPTY: begin
        if (valid_i === 1) next_state_f = FULL;
        else next_state_f = EMPTY;
      end
      FULL: begin
        if (ready_o === 1) next_state_f = EMPTY;
        else next_state_f = FULL;
      end
    endcase // case(state)
  end
endfunction // next_state_f

assign next_state = next_state_f(valid_i,ready_o,state);
```
Brute Force?

```verbatim
parameter [0:0] EMPTY = 1'b0,
    FULL = 1'b1;

function [0:0] next_state_f;
    input valid_i, ready_o, state;
    begin
        case(state)
            EMPTY: begin
                if (valid_i === 1) next_state_f = FULL;
                else next_state_f = EMPTY;
            end
            FULL: begin
                if (ready_o === 1) next_state_f = EMPTY;
                else next_state_f = FULL;
            end
        endcase // case(state)
    end
endfunction // next_state_f

assign next_state = next_state_f(valid_i,ready_o,state);
```

<table>
<thead>
<tr>
<th>state</th>
<th>valid_i</th>
<th>ready_o</th>
<th>next_state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</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>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

© Herman Schmit, 2003
What you need to do

- Generate new gate-level models of entire design from RTL
- Actually, you can change the way it works, if you want
- But it must still pass the testbench

Criteria will be:
- (How Many Cycles Required by Testbench) (as determined by Verilog)
  \[ \times \]
- (Achievable Clock Cycle Time) (as determined by SPICE)
Checkoff

- Show the TA your Gate Level Design
- Show your TA the testbench running the gate level design
- Legal Verilog:
  - built-in primitives: and, nand, or, nor, xor, xnor, buf, not
  - RTL registers:
    always @(posedge clk or negedge rst)
    if (rst == 0)
        Q <= 0;
    else
        Q <= D;
Week 3 & 4

- Circuit Level Design and verification
- Build transistor level models for all gates and FFs
- Use the testbench to continue verification of switch model
- Use SPICE, and your own test vectors to test electricals
Week 5

- Floorplanning: Figure out a rough plan of your layout
Weeks 6 & 7

- **Layout**
  - Verify thru LVS
Weeks 8 & 9

- Circuit Optimization
  - LONG INTERCONNECTS, LONG LOGIC CHAINS
  - Can you improve them?

- If all goes well, this is done before Thanksgiving!
Until the Final

- Create the final report