Verilog 2 - Design Examples

6.375 Complex Digital Systems
Arvind
February 9, 2009

February 9, 2009

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
Verilog can be used at several levels

A common approach is to use C/C++ for initial behavioral modeling, and for building test rigs.

- High-Level Behavioral
- Register Transfer Level
- Gate Level

Automatic tools to synthesize a low-level gate-level model.
Writing synthesizable Verilog: Combinational logic

- Use continuous assignments (**assign**)
  ```verilog
  assign C_in = B_out + 1;
  ```

- Use **always@(*)** blocks with blocking assignments (**=**)
  ```verilog
  always @(*)
  begin
    out = 2’d0;
    if (in1 == 1)
      out = 2’d1;
    else if (in2 == 1)
      out = 2’d2;
  end
  ```

- Every variable should have a default value to avoid inadvertent introduction of latches
- Do not assign the same variable from more than one **always** block — ill defined semantics

**always** blocks allow more expressive control structures, though not all will synthesize default
Writing synthesizable Verilog:
Sequential logic

- Use `always @(posedge clk)` and non-blocking assignments (`<=`)
  ```verilog
  always @(posedge clk)
  C_out <= C_in;
  ```

- Use only positive-edge triggered flip-flops for state

- Do not assign the same variable from more than one always block – ill defined semantics

- Do not mix blocking and non-blocking assignments

- Only leaf modules should have functionality; use higher-level modules only for wiring together sub-modules
An example

wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk) begin
    A_out <= A_in;
    B_out <= A_out + 1;
    C_out <= B_out + 1;
end

The order of non-blocking assignments does not matter!

The effect of non-blocking assignments is not visible until the end of the “simulation tick”
Another way

wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @(posedge clk)
begin
  A_out <= A_in;
  B_out <= B_in;
  C_out <= C_in;
end

assign B_in = A_out + 1;
assign C_in = B_out + 1;

B_in and C_in are evaluated as needed
An example:
Some wrong solutions

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @(posedge clk)
begin
  A_out <= A_in;
  B_out <= B_in;
  C_out <= C_in;
  assign B_in = A_out + 1;
  assign C_in = B_out + 1;
end
```

Syntactically illegal
Another style – multiple 
always blocks

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;

Does it have the same functionality?

Yes. But why?

Need to understand something about Verilog execution semantics.

 Courtesy of Arvind http://csg.csail.mit.edu/6.375/
Yet another style – blocking assignments

wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
begin
  A_out = A_in;
  B_out = B_in;
  C_out = C_in;
end

assign B_in = A_out + 1;
assign C_in = B_out + 1;

Does it have the same functionality?

Not even close!
Verilog execution semantics

- Driven by simulation
- Explained using event queues
Execution semantics of Verilog - 1

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;

Active Event Queue

On clock edge all those events which are sensitive to the clock are added to the active event queue in any order!
Execution semantics of Verilog - 2

```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;
```

Active Event Queue

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>B</td>
</tr>
</tbody>
</table>

A evaluates and as a consequence 1 is added to the event queue
Execution semantics of Verilog -3

```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;
```

Active Event Queue

Event queue is emptied before we go to next clock cycle
Non-blocking assignment

Within a “simulation tick” all RHS variables are read first and all the LHS variables are updated together at the end of the tick

Consequently, two event queues have to be maintained – one keeps the computations to be performed while the other keeps the variables to be updated
Non-blocking assignments require two event queues

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;

Variables in RHS of always blocks are not updated until all inputs (e.g. LHS + dependencies) are evaluated.
Verilog Design Examples

- Greatest Common Divisor
- Unpipelined SMIPSv1 processor
GCD in C

```c
int GCD( int inA, int inB)
{
    int done = 0;
    int A = inA;
    int B = inB;
    while ( !done )
    {
        if ( A < B )
        {
            swap = A;
            A = B;
            B = swap;
        }
        else if ( B != 0 )
        {
            A = A - B;
        }
        else
        {
            done = 1;
        }
    }
    return A;
}
```

Such a GCD description can be easily written in Behavioral Verilog

It can be simulated but it will have nothing to do with hardware, i.e. it won’t synthesize.
Behavioral GCD in Verilog

module GCD_behav#( parameter W = 16 )
( input [W-1:0] inA, inB,
  output [W-1:0] out );
reg [W-1:0] A, B, out, swap;
integer done;
always @(*)
begin
  done = 0; A = inA; B = inB;
  while ( !done )
  begin
    if ( A < B )
      swap = A;
      A = B; B = swap;
    else if ( B != 0 )
      A = A - B;
    else
      done = 1;
  end
  out = A;
end endmodule

User sets the input operands and checks the output; the answer will appear immediately, like a combinational circuit.

Note data dependent loop, "done"
Deriving an RTL model for GCD

```
module gcdGCDUnit_behav#( parameter W = 16 )
  ( input [W-1:0] inA, inB, 
  output [W-1:0] out ), 
reg [W-1:0] A, B, out, swap; 
integer done; 
always @(*) 
begin
  done = 0; A = inA; B = inB; 
  while ( !done ) 
  begin
    if ( A < B ) 
      swap = A; 
      A = B; B = swap; 
    else if ( B != 0 ) 
      A = A - B; 
    else
      done = 1; 
  end
  out = A; 
end 
endmodule
```

What does the RTL implementation need?
- State
- Less-Than Comparator
- Equal Comparator
- Subtractor

Courtesy of Arvind http://csg.csail.mit.edu/6.375/

February 9, 2009
Step 1: Design an appropriate port interface

- input_available
- idle
- operand_A
- operand_B
- result_data
- result_rdy
- result_taken
- clk
- reset
Step 2: Design a datapath which has the functional units

\[
\text{A} = \text{inA}; \ \text{B} = \text{inB}; \\
\text{while} \ (\neg \text{done}) \ \\
\text{begin} \\
\text{if} \ (\text{A} < \text{B}) \\
\text{swap} = \text{A}; \\
\text{A} = \text{B}; \\
\text{B} = \text{swap}; \\
\text{else if} \ (\text{B} \neq 0) \\
\text{A} = \text{A} - \text{B}; \\
\text{else} \\
\text{done} = 1; \\
\text{End} \\
\text{Y} = \text{A};
\]
Step 3: Add the control unit to sequence the datapath

Control unit should be designed to be either busy or waiting for input or waiting for output to be picked up.

\[
\begin{align*}
A &= \text{inA}; B = \text{inB}; \\
\text{while} \ ( \neg \text{done} ) \ &\text{begin} \\
&\quad \text{if} \ ( A < B ) \\
&\quad \quad \text{swap} = A; \\
&\quad \quad A = B; \\
&\quad \quad B = \text{swap}; \\
&\quad \quad \text{else if} \ ( B \neq 0 ) \\
&\quad \quad \quad A = A - B; \\
&\quad \quad \text{else} \\
&\quad \quad \quad \text{done} = 1; \\
\text{End} \\
Y &= A;
\end{align*}
\]
module GCDdatapath#( parameter W = 16 )
( input clk,

  // Data signals
  input [W-1:0] operand_A,
  input [W-1:0] operand_B,
  output [W-1:0] result_data,

  // Control signals (ctrl->dpath)
  input A_en,
  input B_en,
  input [1:0] A_sel,
  input B_sel,

  // Control signals (dpath->ctrl)
  output B_zero,
  output A_lt_B
);

February 9, 2009

Courtesy of Arvind http://
csg.csail.mit.edu/6.375/
Connect the modules

```vhdl
wire [W-1:0] B;
wire [W-1:0] sub_out;
wire [W-1:0] A_out;

vcMux3#(W) A_mux
(  .in0 (operand_A),
   .in1 (B),
   .in2 (sub_out),
   .sel (A_sel),
   .out (A_out)  );

wire [W-1:0] A;

vcEDFF_pf#(W) A_pf
(  .clk  (clk),
   .en_p (A_en),
   .d_p  (A_out),
   .q_np (A)      );
```

February 9, 2009

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
Connect the modules ... 

```verilog
wire [W-1:0] B;
wire [W-1:0] sub_out;
wire [W-1:0] A_out;

vcMux3#(W) A_mux
( .in0 (operand_A),
  .in1 (B),
  .in2 (sub_out),
  .sel (A_sel),
  .out (A_out) );

wire [W-1:0] A;
vcEDFF_pf#(W) A_pf
( .clk (clk),
  .en_p (A_en),
  .d_p (A_out),
  .q_np (A) );

assign B_zero  = (B==0);
assign A_lt_B  = (A < B);
assign sub_out = A - B;
assign result_data = A;

vcMux2#(W) B_mux
( .in0 (operand_B),
  .in1 (A),
  .sel (B_sel),
  .out (B_out) );

vcEDFF_pf#(W) B_pf
( .clk (clk),
  .en_p (B_en),
  .d_p (B_out),
  .q_np (B) );
```

Using explicit state helps eliminate issues with non-blocking assignments.

Continuous assignment combinational logic is fine.
Control unit requires a state machine for valid/ready signals

- **WAIT**: Waiting for new input operands
- **CALC**: Swapping and subtracting
  - \( B = 0 \)
- **DONE**: Waiting for consumer to take the result

- **reset**
- **input_available**
- **result_taken**
Implementing the control logic FSM in 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) );

Localparams are not really parameters at all. They are scoped constants.

Explicit state in the control logic is also a good idea!
Control signals for the FSM

```verilog
reg [6:0] cs;
always @(*)
begin
    //Default control signals
    A_sel    = A_SEL_X;
    A_en     = 1'b0;
    B_sel    = B_SEL_X;
    B_en     = 1'b0;
    input_available = 1'b0;
    result_rdy   = 1'b0;
    case ( state )
        WAIT : ...;
        CALC : ...;
        DONE : ...
    endcase
end

WAIT: begin
    A_sel    = A_SEL_IN;
    A_en     = 1'b1;
    B_sel    = B_SEL_IN;
    B_en     = 1'b1;
    input_available = 1'b1;
end

CALC: if ( A lt B )
    A_sel = A_SEL_B;
    A_en   = 1'b1;
    B_sel = B_SEL_A;
    B_en   = 1'b1;
else if ( !B_zero )
    A_sel = A_SEL_SUB;
    A_en   = 1'b1;
end

DONE: result_rdy = 1'b1;
```

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
always @(*)
begin
  // Default is to stay in the same state
  state_next = state;

  case ( state )
    WAIT :
      if ( input_available )
        state_next = CALC;
    CALC :
      if ( B_zero )
        state_next = DONE;
    DONE :
      if ( result_taken )
        state_next = WAIT;
  endcase
end
RTL test harness requires proper handling of the ready/valid signals

Generic Test Source

A sel  A
B sel  B

B = 0  A < B

zero?  It

sub

Generic Test Sink

Courtesy of Arvind http://csg.csail.mit.edu/6.375/