Verilog can be used at several levels

High-Level Behavioral

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

Register Transfer Level

automatic tools to synthesize a low-level gate-level model.

Gate Level
Writing Good Synthesizable Verilog

- Use only positive-edge triggered flip-flops for state
- Do not assign the same variable from more than one always block
- Describe combinational logic using continuous assignments (`assign`) and `always@(*)` blocks with blocking assignments (`=`)
  ```verilog
  assign C_in = B_out + 1;
  always @(*)
  begin
    out = 2’d0;
    if (in1 == 1)
      out = 2’d1;
    else if (in2 == 1)
      out = 2’d2;
  end
  ```
- Describe sequential logic using `always @(posedge clk)` and non-blocking assignments (`<=`)
  ```verilog
  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;
  ```
- Only leaf modules should have functionality; use higher-level modules only for wiring together sub-modules

An example

```verilog
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;
```

The order of non-blocking assignments does not matter!
Another style – multiple `always` blocks

```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;
```

Does it have the same functionality?

---

Execution semantics of Verilog

```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;
```

On clock edge all those events which are sensitive to the clock are added to the active event queue in any order!
Semantics of non-blocking assignment requires two event queues

```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

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

Behavioral Verilog is richer

- Characterized by heavy use of sequential blocking statements in large always blocks
- Many constructs are **not synthesizable** but can be useful for behavioral modeling
  - Data dependent for and while loops
  - Additional behavioral datatypes: `integer, real`
  - Magic initialization blocks: `initial`
  - Magic delay statements: `#<delay>`
  - System calls: `$display, $assert, $finish`
System tasks are used for test harnesses and simulation management

```verilog
reg [ 1023:0 ] exe_filename;

initial
begin

    // This turns on VCD (plus) output
    $vcdpluson(0);

    // This gets the program to load into memory from the command line
    if ( $value$plusargs("exe=%s", exe_filename) )
        $readmemh( exe_filename, mem.m );
    else
        begin
            $display("ERROR: No executable specified! (use +exe=<filename>)");
            $finish;
        end

    // Strobe reset
    #0 reset = 1;
    #38 reset = 0;

end
```

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

```verilog
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;
endmodule
```

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

Data dependent loop, “done”
You have to be careful in using such behavioral models

```verilog
testHarnessBehav;  
reg [15:0] inA, inB;  
wire [15:0] out;  
exGCD_behav#(16) gcd_unit( .inA(inA), .inB(inB), .out(out) );
initial
begin
  // 3 = GCD( 27, 15 )
  inA = 27;
  inB = 15;
  #10;
  if ( out == 3 )
    $display("Test ( gcd(27,15) ) succeeded, [ %x == %x ]", out, 3);
  else
    $display("Test ( gcd(27,15) ) failed, [ %x != %x ]", out, 3);
  $finish;
end
endmodule
```

without some delay
out is bogus

Deriving an RTL model for GCD

```verilog
gcdGCDUnit_behav#( parameter W = 16 )
(
  input [W-1:0] inA, inB,
  output [W-1:0] Y
);
reg [W-1:0] A, B, Y, 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
  Y = A;
end
endmodule
```

What does the RTL implementation need?

- **State**
- **Less-Than Comparator**
- **Equal Comparator**
- **Subtractor**
The first step is to carefully design an appropriate port interface.

Next develop a datapath which has the proper functional units.

```
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
Y = A;
```
Finally add the control unit to sequence the datapath

Datapath module interface

Control unit should be designed so that it is either busy or waiting for input or waiting for output to be picked up

```
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
Y = A;
```
Connect the modules

wire [W-1:0] B;
wire [W-1:0] sub_out;
wire [W-1:0] A_mux_out;
vcMux3#(W) A_mux
{
    .in0 (operands_bits_A),
    .in1 (B),
    .in2 (sub_out),
    .sel (A_mux_sel),
    .out (A_mux_out)
};
wire [W-1:0] A;
vcEDFF_pf#(W) A_pf
{
    .clk (clk),
    .en_p (A_en),
    .d_p (A_mux_out),
    .q_np (A)
};

wire [W-1:0] B_mux_out;
vcMux2#(W) B_mux
{
    .in0 (operands_bits_B),
    .in1 (A),
    .sel (B_mux_sel),
    .out (B_mux_out)
};
vcEDFF_pf#(W) B_pf
{
    .clk (clk),
    .en_p (B_en),
    .d_p (B_mux_out),
    .q_np (B)
};
assign B_zero = ( B == 0 );
assign A_lt_B = ( A < B );
assign sub_out = A - B;
assign result_bits_data = A;

Connect the modules ...
Control unit requires a simple state machine for valid/ready signals

![Control Unit State Diagram]

- **WAIT**: Waiting for new input operands
- **CALC**: Swapping and subtracting
- **DONE**: Waiting for consumer to take the result

Implementing the control logic finite state machine in Verilog

```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!
Implementing the control signal outputs for the finite state machine

```verilog
define cs
always @(*) begin
    // Default control signals
    A_mux_sel = A_MUX_SEL_X;
    A_en = 1'b0;
    B_mux_sel = B_MUX_SEL_X;
    B_en = 1'b0;
    input_available = 1'b0;
    result_rdy = 1'b0;
    case ( state )
        WAIT : begin
            // if input_available
            state_next = CALC;
        CALC : begin
            // if B_zero
            state_next = DONE;
        endcase
end
```

Implementing the state transitions for the finite state machine

```verilog
always @(*) begin
    // Default is to stay in the same state
    state_next = state;
    case ( state )
        WAIT : begin
            // if input_available
            state_next = CALC;
        CALC : begin
            // if B_zero
            state_next = DONE;
        endcase
end
```
RTL test harness requires proper handling of the ready/valid signals

We can compare the behavioral and RTL implementations to verify correctness.
Verilog Design Examples

- Greatest Common Divisor
- Unpipelined SMIPSv1 processor

SMIPS is a simple MIPS ISA which includes three variants

- SMIPSv1
  - 5 instructions
  - No exceptions/interrupts
  - Lecture examples
- SMIPSv2
  - 35 instructions
  - No exceptions/interrupts
  - ISA for lab assignments
- SMIPSv3
  - 58 instructions
  - Full system coproc with exceptions/Interrupts
  - Optional ISA for projects
### SMIPSv1 ISA

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
<th>Hardware Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addiu rt, rs, imm</code></td>
<td><code>R[rt] := R[rs] + sext(imm)</code></td>
<td>Needs adder, sext, 1w1r rf port</td>
</tr>
<tr>
<td><code>bne rs, rt, offset</code></td>
<td>if ( <code>R[rs] != R[rt]</code>) ( pc := pc + sext(offset) + 4 )</td>
<td>Needs adder, sext, comparator, 2r rf port</td>
</tr>
<tr>
<td><code>lw rt, offset(rs)</code></td>
<td><code>R[rt] := M[R[rs] + sext(offset)]</code></td>
<td>Needs adder, sext, memory read port, 1r1w rf port</td>
</tr>
<tr>
<td><code>sw rt, offset(rs)</code></td>
<td><code>M[R[rs] + sext(offset)] = R[rt]</code></td>
<td>Needs adder, sext, memory write port, 1r1w port</td>
</tr>
</tbody>
</table>

---

**The first step is to carefully design an appropriate port interface**

```
smipsTestHarness

smipsProc

imemreq_bits_addr
imemreq_bits_data
imemreq_val

dmemreq_bits_addr
dmemreq_bits_data
dmemreq_val

clk
reset

Magic Dual Port Memory

clk
reset
```
Identify memories, datapaths, and random logic

Step 1: Identify the memories
Step 2: Identify the datapaths
Step 3: Everything else is random logic

Identify the signals to interface with the controller
SMIPSv1 datapath

module smipsProcDpath_pstr
(input clk, reset,
// Memory ports
output [31:0] imemreq_bits_addr, output [31:0] dmemreq_bits_addr, output [31:0] dmemreq_bits_data, input [31:0] dmemresp_bits_data,
// Controls signals (ctrl->dpath)
input pc_mux_sel, input [ 4:0] rf_raddr0, input [ 4:0] rf_raddr1, input rf_wen, input [ 4:0] rf_waddr, input op0_mux_sel, input op1_mux_sel, input [15:0] inst_imm, input wb_mux_sel,
// Control signals (dpath->ctrl)
output branch_cond_eq, output [7:0] tohost_next);

wire [31:0] branch_targ;
wire [31:0] pc_plus4;
wire [31:0] pc_mux_out;
vcMux2#(32) pc_mux
(.in0 (pc_plus4),
 .in1 (branch_targ),
 .sel (pc_mux_sel),
 .out (pc_mux_out));
wire [31:0] pc;
vcRDFF_pf#(32,32'h0001000) pc_pf
(.clk (clk),
 .reset_p (reset),
 .d_p (pc_mux_out),
 .q_np (pc));
assign imemreq_bits_addr = pc;
vcInc#(32,32'd4) pc_inc4
(.in  (pc),
 .out (pc_plus4));

endmodule

Register file with two combinational read ports and one write port

module smipsProcDpathRegfile
(input clk,
 input [ 4:0] raddr0,  // Read 0 address (combinational input)
 output [31:0] rdata0,  // Read 0 data (combinational on raddr)
 input  [ 4:0] raddr1,  // Read 1 address (combinational input)
 output [31:0] rdata1,  // Read 1 data (combinational on raddr)
 input wen_p,   // Write enable (sample on rising clk edge)
 input [ 4:0] waddr_p, // Write address (sample on rising clk edge)
 input [31:0] wdata_p // Write data (sample on rising clk edge)
);
// We use an array of 32 bit register for the regfile itself
reg [31:0] registers[31:0];

// Combinational read ports
assign rdata0 = ( raddr0 == 0 ) ? 32'b0 : registers[raddr0];
assign rdata1 = ( raddr1 == 0 ) ? 32'b0 : registers[raddr1];

// Write port is active only when wen is asserted
always @(posedge clk)
if ( wen_p & (waddr_p != 5'b0) )
  registers[waddr_p] <= wdata_p;
endmodule
Verilog for SMIPSv1 control logic

```verilog
// Define instruction operands
`define LW 32'b100011_?????_?????_?????_?????_?????_?????
`define SW 32'b101011_?????_?????_?????_?????_?????_?????
`define ADDIU 32'b001001_?????_?????_?????_?????_?????_?????
`define BNE 32'b000101_?????_?????_?????_?????_?????_?????

localparam cs_sz = 8;
reg [cs_sz-1:0] cs;
always @(*)
begin
    cs = {cs_sz{1'b0}};
    casez ( imemresp_bits_data )
        //                      op0 mux op1 mux wb mux rfile mreq mreq tohost
        //              br type sel sel sel sel wen r/w val en
        `ADDIU : cs = { br_pc4, op0_sx,  op1_rd0, wmx_alu, 1'b1, mreq_x, 1'b0, 1'b0 };
        `BNE   : cs = { br_neq, op0_sx2, op1_pc4, wmx_x,   1'b0, mreq_x, 1'b0, 1'b0);
        `LW    : cs = { br_pc4, op0_sx,  op1_rd0, wmx_mem, 1'b1, mreq_r, 1'b1, 1'b0};
        `SW    : cs = { br_pc4, op0_sx,  op1_rd0, wmx_x,   1'b0, mreq_w, 1'b1, 1'b0};
        `MTC0  : cs = { br_pc4, op0_x,   op1_x,   wmx_x,   1'b0, mreq_x, 1'b0, 1'b1};
    endcase
end
```

casez performs simple pattern matching and can be very useful when implementing decoders

---

Verilog for SMIPSv1 control logic

```verilog
// Set the control signals based on the decoder output
wire br_type = cs[7];
assign pc_mux_sel = ( br_type == br_pc4 ) ? 1'b0 : ( br_type == br_neq ) ? ~branch_cond_eq : 1'bx;
assign op0_mux_sel = cs[6];
assign op1_mux_sel = cs[5];
assign wb_mux_sel = cs[4];
assign rf_wen = ( reset ? 1'b0 : cs[3]);
assign dmemreq_bits_rw = cs[2];
assign dmemreq_val = ( reset ? 1'b0 : cs[1]);
wire tohost_en = ( reset ? 1'b0 : cs[0]);

// These control signals we can set directly from the instruction bits
assign rf_raddr0 = inst[25:21];
assign rf_raddr1 = inst[20:16];
assign rf_waddr = inst[20:16];
assign inst_imm = inst[15:0];

// We are always making an imemreq
assign imemreq_val = 1'b1;
```
Take away points

• Parameterized models provide the foundation for reusable libraries of components
• Use explicit state to prevent unwanted state inference and to more directly represent the desired hardware
• Begin your RTL design by identifying the external interface and then move on to partition your design into the memories, datapaths, and control logic