Verilog 2 - Design Examples
Verilog can be used at several levels

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

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

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 values.
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

```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 <= A_out + 1;
    C_out <= B_out + 1;
end
```

The order of non-blocking assignments does not matter!

Why? Because they generate registers!
Another way

```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;
```

Same behavior; we’ve just generated the combinational logic separately.
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

```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?

Yes. But why?

It generates the same underlying circuit.
Yet another style – blocking assignments

```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;
```

Will this synthesize?

→ Yes

Is it correct?

→ No; Do not use “blocking assignments” in @posedge clk blocks. It is forbidden in this class.
Verilog execution semantics

- Confusing
- Best solution is to write synthesizable verilog that corresponds exactly to logic you have already designed on paper, as described by the “verilog breakdown” slide.
If you treat verilog as a language for coding up hardware you have already designed on paper/whiteboard, you will not need to rely on this.

- **unordered (sequential right hand side)**
- **unordered (sequential left hand side)**

**Clock Edge**

**auto-ordered** (combinational)

```verilog
always @(posedge clk)
begin
  <= foo;
  <= bar;
end

always @(posedge clk)
begin
  foo = q;
  bar = r;
end

always @(posedge clk)
begin
  <= foo;
  <= bar;
end

always @(*)
begin
  b = f(a)
  c = d + 1;
end

assign d = b+2;

assign e = c+2;
```
Taylor’s Simplified Verilog Semantics

always @(*)
begin
    b = f(a);
    c = d + 1;
end

assign d = b + 2;

assign e = c + 2;

always @(posedge clk)
begin
    <= foo;
    <= bar;
end

always @(posedge clk)
begin
    foo = q;
    bar = r;
end

always @(posedge clk)
begin
    a <=
    x <=
end

Verilog Design Examples

- Greatest Common Divisor
- Unpipelined SMIPSv1 processor

Courtesy of Arvind
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.

We don’t spend much time on Behavioral Verilog because it is not a particularly good language and isn’t useful for hardware synthesis.
```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;
}
```

What does the RTL implementation need?

- State
- Less-Than Comparator
- Equal Comparator
- Subtractor
Step 1: Design an appropriate port interface

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

\[ A = \text{inA}; B = \text{inB}; \]

\[ \text{while} \ ( \neg \text{done} ) \]

\[ \text{begin} \]

\[ \text{if} \ ( A < B ) \]

\[ \text{swap} = A; \]

\[ A = B; \]

\[ B = \text{swap}; \]

\[ \text{else if} \ ( B \neq 0 ) \]

\[ A = A - B; \]

\[ \text{else} \]

\[ \text{done} = 1; \]

\[ \text{End} \]

\[ Y = A; \]

Courtesy of Arvind
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

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;
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 );

Courtesy of Arvind
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) );
```

Courtesy of Arvind
Connect the datapath modules …

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));

wire [W-1:0] B_out;
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));

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

Use structural verilog for datapath registers.

Not quite right: Continuous assignment combinational logic is fine for datapath operators BUT wrap them in a verilog module and instantiate structurally!
Control unit requires a state machine for valid/ready signals.

- **WAIT**: Waiting for new input operands
- **CALC**: Swapping and subtracting
  - (Done = 1)
- **DONE**: Waiting for consumer to take the result

State transitions:
- **reset** to **WAIT**
- **input_available** from **WAIT** to **CALC**
- **result_taken** from **CALC** to **DONE**

Courtesy of Arvind
Implementing the control logic FSM in Verilog

```verilog
localparam WAIT = 2'd0;
localparam CALC = 2'd1;
localparam DONE = 2'd2;

reg [1:0] state_next;
wire [1:0] state;

vcRDIFF_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.

Not quite right: For registers in control logic, use RTL: e.g.

```verilog
always @(posedge clk)
  if (reset)
    state <= WAIT;
  else
    state <= state_next;
```
Output signals for control logic

```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;
endcase

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 )
begin
    A_sel = A_SEL_B;
    A_en = 1'b1;
    B_sel = B_SEL_A;
    B_en = 1'b1;
else if ( !B_zero )
begin
    A_sel = A_SEL_SUB;
    A_en = 1'b1;
end
end

DONE:
result_rdy = 1'b1;
done

end
```

Courtesy of Arvind
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

reg [1:0] state;
always @(posedge clk)
    if (reset)
        state <= WAIT;
    else
        state <= state_next;

reset
input_available

WAIT

CALC

DONE

result_taken

( B = 0 )
RTL test harness requires proper handling of the ready/valid signals
Correctness: Compare behavioral and RTL implementations

Test Inputs

C Model

Test Outputs

RTL Model

Test Outputs

Identical?
Verilog Design Examples

- Greatest Common Divisor
- Unpipelined SMIPSv1 processor
SMIPS is a simple MIPS ISA which includes three variants

- **SMIPSv1**
  - 5 instructions
  - Lecture examples
## SMIPSv1 ISA

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

Courtesy of Arvind
First step: Design a port interface
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
module smipsProcDpath_pstr
    ( input clk, reset,
    // Memory ports
    output [31:0] imemreq_addr,
    output [31:0] dmemreq_addr,
    output [31:0] dmemreq_data,
    input [31:0] dmemresp_data,
    // Controls signals (ctrl->dpath)
    input pc_sel,
    input [4:0] rf_raddr0,
    input [4:0] rf_raddr1,
    input rf_wen,
    input [4:0] rf_waddr,
    input op0_sel,
    input op1_sel,
    input [15:0] inst_imm,
    input wb_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_out;
vcMux2#(32) pc_mux
    ( .in0 (pc_plus4),
    .in1 (branch_targ),
    .sel (pc_sel),
    .out (pc_out) );

wire [31:0] pc;
vcRDFF_pf#(32,32'h0001000) pc_pf
    ( .clk (clk),
    .reset_p (reset),
    .d_p (pc_out),
    .q_np (pc) );

assign imemreq_addr = pc;
vcInc#(32,32'd4) pc_inc4
    ( .in (pc),
    .out (pc_plus4) );

Courtesy of Arvind
Register file with 2 combinational read ports and 1 write port

```verilog
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

```
`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_data)
        `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

// Set the control signals based on the decoder output
wire br_type = cs[7];
assign pc_sel = ( br_type == br_pc4 ) ? 1'b0:
               ( br_type == br_neq ) ? ~branch_cond_eq:
                           1'bx;
assign op0_sel  = cs[6];
assign op1_sel  = cs[5];
assign wb_sel   = cs[4];
assign rf_wen   = ( reset ? 1'b0 : cs[3] );
assign dmemreq_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;

Courtesy of Arvind
Take away points

- Follow the simple guidelines to write synthesizable Verilog
- 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
Behavioral Verilog: For TestBenchs Only

Characterized by heavy use of sequential blocking statements in large always blocks

Many constructs are not synthesizable but can be useful for behavioral modeling and test benches

- Data dependent for and while loops
- Additional behavioral datatypes: `integer, real`
- Magic initialization blocks: `initial`
- Magic delay statements: `#<delay>
  - Except for clk generation, use `@(negedge clk)`;
- System calls: `$display, $assert, $finish`
System calls for test harnesses and simulation

```verilog
define 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
    // Stobe reset
    #0 reset = 1;
    #38 reset = 0;
end
```

Courtesy of Arvind
Some dangers in writing behavioral models

```
module GCDTestHarness_behav;
    reg [15:0] inA, inB;
    wire [15:0] out;

    GCD_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.