module adder(
    input [3:0] A, B,
    output cout,
    output [3:0] S);
wire c0, c1, c2;
FA fa0( A[0], B[0], 1'b0, c0,   S[0] );
FA fa1( A[1], B[1], c0,   c1,   S[1] );
FA fa2( A[2], B[2], c1,   c2,   S[2] );
FA fa3( A[3], B[3], c2,   cout, S[3] );
endmodule
Verilog Review

- Data types
- Structural Verilog
  - Gate level
  - Register transfer level
  - Functional level
- Behavioral Verilog
  - Gate level
  - Register transfer level
  - Functional level
- Parameterized Static Elaboration
- Greatest Common Divisor
Primary Verilog data type is a bit-vector where bits can take on one of four values

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Logic zero</td>
</tr>
<tr>
<td>1</td>
<td>Logic one</td>
</tr>
<tr>
<td>X</td>
<td>Unknown logic value</td>
</tr>
<tr>
<td>Z</td>
<td>High impedance, floating</td>
</tr>
</tbody>
</table>

An X bit might be a 0, 1, Z, or in transition. We can set bits to be X in situations where we don’t care what the value is. This can help catch bugs and improve synthesis quality.
The Verilog keyword `wire` is used to denote a standard hardware net.

- `wire [15:0] instruction;`
- `wire [15:0] memory_req;`
- `wire [7:0] small_net;`

Absolutely no type safety when connecting nets!
Verilog includes ways to specify bit literals in various bases

- **Binary literals**
  - 8’b0000_0000
  - 8’b0xx0_1xx1

- **Hexadecimal literals**
  - 32’h0a34_def1
  - 16’haxxx

- **Decimal literals**
  - 32’d42

We’ll learn how to actually assign literals to nets a little later
Verilog Review

- Data types
- **Structural Verilog**
- Behavioral Verilog
  - Gate level
  - Register transfer level
  - Functional level
- Parameterized Static Elaboration
- Greatest Common Divisor
A Verilog module includes a module name and a port list

```verilog
module adder( A, B, cout, sum );
  input [3:0] A;
  input [3:0] B;
  output cout;
  output [3:0] sum;

  // HDL modeling of adder functionality
endmodule
```

Note the semicolon at the end of the port list!

Ports must have a direction (or be bidirectional) and a bitwidth.
A Verilog module includes a module name and a port list

Traditional Verilog-1995 Syntax

```verilog
module adder( A, B, cout, sum );
input  [3:0] A;
input  [3:0] B;
output cout;
output [3:0] sum;
```

ANSI C Style Verilog-2001 Syntax

```verilog
module adder( input [3:0] A,
              input [3:0] B,
              output cout,
              output [3:0] sum );
```
A module can instantiate other modules creating a module hierarchy

```verilog
module FA(
    input a, b, cin,
    output cout, sum
);

    // HDL modeling of 1 bit adder functionality

endmodule
```
A module can instantiate other modules creating a module hierarchy

```verilog
module adder( input [3:0] A, B,
              output cout,
              output [3:0] S );

wire c0, c1, c2;
FA fa0( ... );
FA fa1( ... );
FA fa2( ... );
FA fa3( ... );

endmodule
```
A module can instantiate other modules creating a module hierarchy

```verilog
module adder( input [3:0] A, B, 
              output cout, 
              output [3:0] S );
wire c0, c1, c2;
FA fa0( A[0], B[0], 1'b0, c0, S[0] );
FA fa1( A[1], B[1], c0, c1, S[1] );
FA fa2( A[2], B[2], c1, c2, S[2] );
FA fa3( A[3], B[3], c2, cout, S[3] );
endmodule
```

Carry Chain
Verilog supports connecting ports by position and by name

Connecting ports by ordered list

```verilog
FA fa0( A[0], B[0], 1'b0, c0, S[0] );
```

Connecting ports by name (compact)

```verilog
FA fa0( .a(A[0]), .b(B[0]),
       .cin(1'b0), .cout(c0), .sum(S[0]) );
```

Connecting ports by name

```verilog
FA fa0
  (  
    .a    (A[0]),
    .b    (B[0]),
    .cin  (1'b0),
    .cout (c0),
    .sum  (S[0])
  );
```

For all but the smallest modules, connecting ports by name yields clearer and less buggy code.
Let’s review how to turn our schematic diagram into structural Verilog

```
module adder(
    input  [3:0] A, B,
    output cout, 
    output [3:0] S
);

wire c0, c1, c2;
FA fa0( A[0], B[0], 1'b0, c0, S[0] );
FA fa1( A[1], B[1], c0, c1, S[1] );
FA fa2( A[2], B[2], c1, c2, S[2] );
FA fa3( A[3], B[3], c2, cout, S[3] );
endmodule
```
Verilog Review

- Data types
- Structural Verilog
- Behavioral Verilog
  - Gate level
  - Register transfer level
  - Functional level
- Parameterized Static Elaboration
- Greatest Common Divisor
Behavioral Verilog can roughly be divided into three abstraction levels:

1. **Algorithm** (Manual)
   - Abstract algorithmic description
   - Describes how data flows between state elements for each cycle

2. **Register Transfer Level**
   - Low-level netlist of primitive gates

3. **Gate Level**
   - Auto Place + Route
Gate-level Verilog uses structural Verilog to connect primitive gates

```
module mux4( input a, b, c, d, input [1:0] sel, output out );

    wire [1:0] sel_b;
    not not0( sel_b[0], sel[0] );
    not not1( sel_b[1], sel[1] );

    wire n0, n1, n2, n3;
    and and0( n0, c, sel[1] );
    and and1( n1, a, sel_b[1] );
    and and2( n2, d, sel[1] );
    and and3( n3, b, sel_b[1] );

    wire x0, x1;
    nor nor0( x0, n0, n1 );
    nor nor1( x1, n2, n3 );

    wire y0, y1;
    or or0( y0, x0, sel[0] );
    or or1( y1, x1, sel_b[0] );
    nand nand0( out, y0, y1 );

endmodule
```
Continuous assignment statements assign one net to another or to a literal

Explicit continuous assignment

```verilog
wire [15:0] netA;
wire [15:0] netB;
assign netA = 16'h3333;
assign netB = netA;
```

Implicit continuous assignment

```verilog
wire [15:0] netA = 16'h3333;
wire [15:0] netB = netA;
```
Using continuous assignments to implement an RTL four input multiplexer

```verilog
module mux4( input a, b, c, d
            input [1:0] sel,
            output out );

wire t0, t1;
assign t0  = ~( (sel[1] & c) | (~sel[1] & a) );
assign t1  = ~( (sel[1] & d) | (~sel[1] & b) );
assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );

endmodule
```

The order of these continuous assignment statements does not matter. They essentially happen in parallel!
Verilog RTL includes many operators in addition to basic boolean logic

// Four input multiplexer
module mux4 ( input a, b, c, d
            input [1:0] sel,
            output out );

    assign out = ( sel == 0 ) ? a :
                 ( sel == 1 ) ? b :
                 ( sel == 2 ) ? c :
                 ( sel == 3 ) ? d : 1'bx;
endmodule

// Simple four bit adder
module adder( input [3:0] op1, op2,
              output [3:0] sum );

    assign sum = op1 + op2;
endmodule

If input is undefined we want to propagate that information.
Verilog RTL operators

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>+ - * / % **</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>! &amp;&amp;</td>
</tr>
<tr>
<td>Relational</td>
<td>&gt; &lt; &gt;= &lt;=</td>
</tr>
<tr>
<td>Equality</td>
<td>== != === !===</td>
</tr>
<tr>
<td>Bitwise</td>
<td>~ &amp;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reduction</th>
<th>&amp; ~&amp;</th>
<th>~</th>
<th>^</th>
<th>^~</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>&gt;&gt;</td>
<td>&lt;&lt;</td>
<td>&gt;&gt;&gt;</td>
<td>&lt;&lt;&lt;</td>
</tr>
<tr>
<td>Concatenation</td>
<td>{ }</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td>? :</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

wire [ 3:0] net1 = 4'b00xx;
wire [ 3:0] net2 = 4'b1110;
wire [11:0] net3 = { 4'b0, net1, net2 };

wire equal = ( net3 === 12'b0000_1110_00xx );

Avoid ( / % ** ) since the usually synthesize poorly
Always blocks have parallel inter-block and sequential intra-block semantics

module mux4(
    input a, b, c, d,
    input [1:0] sel,
    output out );

reg out, t0, t1;

always @(a or b or c or d or sel)
begin
    t0  = ~( (sel[1] & c) | (~sel[1] & a) );
    t1  = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end
endmodule

The always block is reevaluated whenever a signal in its sensitivity list changes
Always blocks have parallel inter-block and sequential intra-block semantics

```verilog
module mux4(
  input  a, b, c, d,
  input [1:0] sel,
  output  out 
);

reg out, t0, t1;

always @ ( a or b or c or d or sel )
begin
  t0  = ~( (sel[1] & c) | (~sel[1] & a) );
  t1  = ~( (sel[1] & d) | (~sel[1] & b) );
  out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end
endmodule
```

The order of these procedural assignment statements does matter. They essentially happen in sequentially!
Always blocks have parallel inter-block and sequential intra-block semantics

```verilog
module mux4( input a, b, c, d, sel,
             input [1:0] sel,
             output out );

reg out, t0, t1;
always @( a or b or c or d or sel )
begin
    t0  = ~( (sel[1] & c) | (~sel[1] & a) );
    t1  = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end
endmodule
```

LHS of procedural assignments must be declared as a reg type. Verilog reg is not necessarily a hardware register!
Always blocks have parallel inter-block and sequential intra-block semantics

module mux4(
    input  a, b, c, d
    input [1:0] sel,
    output out);

    reg out, t0, t1;

    always @(a or b or c or d or sel)
    begin
        t0 = ~( (sel[1] & c) | (~sel[1] & a) );
        t1 = ~( (sel[1] & d) | (~sel[1] & b) );
        out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
    end

endmodule

What happens if we accidentally forget a signal on the sensitivity list?
Always blocks have parallel inter-block and sequential intra-block semantics

```verilog
module mux4( input a, b, c, d            
    input [1:0] sel, 
    output out );

reg out, t0, t1;

always @( *)
begin
    t0  = ~( (sel[1] & c) | (~sel[1] & a) );
    t1  = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end
endmodule
```

Verilog-2001 provides special syntax to automatically create a sensitivity list for all signals read in the always block.
Continuous and procedural assignment statements are very different

Continuous assignments are for naming and thus we cannot have multiple assignments for the same wire

```verilog
wire out, t0, t1;
assign t0  = ~( (sel[1] & c) | (~sel[1] & a) );
assign t1  = ~( (sel[1] & d) | (~sel[1] & b) );
assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
```

Procedural assignments hold a value semantically, but it is important to distinguish this from hardware state

```verilog
reg out, t0, t1, temp;
always @( *)
begin
    temp = ~( (sel[1] & c) | (~sel[1] & a) );
    t0   = temp;
    temp = ~( (sel[1] & d) | (~sel[1] & b) );
    t1   = temp;
    out  = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end
```
Always blocks can contain more advanced control constructs

```verilog
module mux4( input  a, b, c, d, input [1:0] sel, output out );

reg out;
always @( * )
begin
  if ( sel == 2'd0 )
    out = a;
  else if ( sel == 2'd1 )
    out = b;
  else if ( sel == 2'd2 )
    out = c;
  else if ( sel == 2'd3 )
    out = d;
  else
    out = 1'bx;
end
endmodule

module mux4( input  a, b, c, d, input [1:0] sel, output out );

reg out;
always @( * )
begin
  case ( sel )
    2'd0 : out = a;
    2'd1 : out = b;
    2'd2 : out = c;
    2'd3 : out = d;
    default : out = 1'bx;
  endcase
  end
endmodule
```
What happens if the case statement is not complete?

```verilog
module mux3(input a, b, c
            input [1:0] sel,
            output out );

reg out;

always @( * )
begin
  case ( sel )
    2’d0 : out = a;
    2’d1 : out = b;
    2’d2 : out = c;
  endcase
end

endmodule
```

If sel = 3, mux will output the previous value.

What have we created?
What happens if the case statement is not complete?

```verilog
module mux3(
    input  a, b, c,
    input [1:0] sel,
    output out );

    reg out;

    always @( *)
    begin
        case ( sel )
            2’d0 : out = a;
            2’d1 : out = b;
            2’d2 : out = c;
            default : out = 1’bx;
        endcase
    end

endmodule
```

We can prevent creating state with a default statement.
So is this how we make latches and flip-flops?

```verilog
defmodule latch
  (   input  clk,
      input  d,
      output reg q
  );

  always @(*) begin
    if ( clk )
      q = d;
  end
endmodule

defmodule flipflop
  (   input  clk,
      input  d,
      output reg q
  );

  always @(posedge clk) begin
    q = d;
  end
endmodule
```

Edge-triggered always block
To understand why we need to know more about Verilog execution semantics

```verilog
type A_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
A_out = A_in;

always @(posedge clk)
B_out = A_out;

assign C_in = B_out + 1;

always @(posedge clk)
C_out = C_in;
```
To understand why we need to know more about Verilog execution semantics

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

always @(posedge clk)
A_out = A_in;

always @(posedge clk)
B_out = A_out;

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!
To understand why we need to know more about Verilog execution semantics

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

always @(posedge clk) 
  A_out = A_in;

always @(posedge clk) 
  B_out = A_out;

assign C_in = B_out + 1;

always @(posedge clk) 
  C_out = C_in;
```
To understand why we need to know more about Verilog execution semantics

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

always @(posedge clk)  
  A_out = A_in;

always @(posedge clk)  
  B_out = A_out;

assign C_in = B_out + 1;

always @(posedge clk)  
  C_out = C_in;
```
To understand why we need to know more about Verilog execution semantics

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

always @(posedge clk) begin
    A_out = A_in;
end

always @(posedge clk) begin
    B_out = A_out;
end

assign C_in = B_out + 1;

always @(posedge clk) begin
    C_out = C_in;
end
```

Active Event Queue

A evaluates and updates A_out
To understand why we need to know more about Verilog execution semantics

```verilog
type: wire, reg
wire A_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk )
A_out = A_in;

always @(posedge clk )
B_out = A_out;

assign C_in = B_out + 1;

always @(posedge clk )
C_out = C_in;
```

Active Event Queue

B evaluates and reads new value of A_out and updates B_out

As a consequence 1 is added to event queue
To understand why we need to know more about Verilog execution semantics

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

always @(posedge clk )
    A_out = A_in;

always @(posedge clk )
    B_out = A_out;

assign C_in = B_out + 1;

always @(posedge clk )
    C_out = C_in;
```

Event queue is emptied before we go to next clock cycle.
To understand why we need to know more about Verilog execution semantics

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

always @(posedge clk) A_out = A_in;

always @(posedge clk) B_out = A_out;

assign C_in = B_out + 1;

always @(posedge clk) C_out = C_in;
```
Non-blocking procedural assignments add an extra event queue

```verilog
global wires and registers

always @(posedge clk) begin
    A_out <= A_in;
end

always @(posedge clk) begin
    B_out <= A_out;
end

assign C_in = B_out + 1;

always @(posedge clk) begin
    C_out <= C_in;
end
```

Put evaluation of RHS of non-blocking assignments in AEQ and actual update into NBQ
Non-blocking procedural assignments add an extra event queue

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

always @(posedge clk )
    A_out <= A_in;

always @(posedge clk )
    B_out <= A_out;

assign C_in = B_out + 1;

always @(posedge clk )
    C_out <= C_in;
```

Once RHS of non-blocking assignments have been evaluated and saved results internally, move updated events from NBQ into AEQ
Non-blocking procedural assignments add an extra event queue

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

always @(posedge clk)
A_out <= A_in;

always @(posedge clk)
B_out <= A_out;

assign C_in = B_out + 1;

always @(posedge clk)
C_out <= C_in;

Updates trigger other events (1) to be added to active event queue, but B does not see the new value from A since B has already evaluated its RHS.
Non-blocking procedural assignments add an extra event queue

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

always @(posedge clk)
    A_out <= A_in;

always @(posedge clk)
    B_out <= A_out;

assign C_in = B_out + 1;

always @(posedge clk)
    C_out <= C_in;
Non-blocking procedural assignments add an extra event queue

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

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

assign C_in = B_out + 1;

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

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

assign C_in = B_out + 1;

The order of non-blocking assignments does not matter!
Common patterns for latch and flip-flop inference

```verilog
always @(*)
begin
  if ( clk )
    Q <= D;
end

always @(posedge clk)
begin
  Q <= D;
end

always @(posedge clk)
begin
  if ( enable )
    Q <= D;
end
```

![Diagram of latch and flip-flop inference patterns](image-url)
Five guidelines for using combinational and sequential always blocks

1. Clearly distinguish always blocks meant for combinational logic from those meant for sequential logic, do not mix combinational logic and sequential logic in the same always block.

2. Only use blocking assignments (=) in combinational always blocks.

3. Only use non-blocking assignments (<=) in sequential always blocks.

4. Try to put as little logic in a sequential always block as possible.

5. Do not assign to the same variable from more than one always block.
Functional Verilog is used to model the abstract function of a hardware module

- 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>
Verilog can be used to model the high-level function of a hardware block

```
module factorial( input [ 7:0] in, output reg [15:0] out );

integer num_calls;
initial num_calls = 0;

integer multiplier;
integer result;
always @(*)
begin
  multiplier = in;
  result = 1;
  while ( multiplier > 0 )
    begin
      result = result * multiplier;
      multiplier = multiplier - 1;
    end

  out = result;
  num_calls = num_calls + 1;
end
endmodule
```
Delay statements should only be used in test harnesses

```verilog
module mux4
  (input a,
   input b,
   input c,
   input d,
   input [1:0] sel,
   output out);

wire #10 t0 = ~( (sel[1] & c) | (~sel[1] & a) );
wire #10 t1 = ~( (sel[1] & d) | (~sel[1] & b) );
wire #10 out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
endmodule
```

Although this will add a delay for simulation, these are ignored in synthesis.
Even synthesizable blocks can be more functional in nature.

```verilog
module ALU
(
    input [31:0] in0,
    input [31:0] in1,
    input [1:0] fn,
    output [31:0] out
);

assign out = ( fn == 2'd0 ) ? ( in0 + in1 ) :
              ( fn == 2'd1 ) ? ( in0 - in1 ) :
              ( fn == 2'd9 ) ? ( in1 >> in0 ) :
              ( fn == 2'd10 ) ? ( in1 >>> in0 ) :
                              32'bx;
endmodule
```

Although this module is synthesizable, it is unlikely to produce the desired hardware.
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

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

end
```
Which abstraction is the right one?

Designers usually use a mix of all three! Early on in the design process they might use mostly behavioral models. As the design is refined, the behavioral models begin to be replaced by dataflow models. Finally, the designers use automatic tools to synthesize a low-level gate-level model.
Verilog Review

• Data types
• Structural Verilog
• Behavioral Verilog
  – Gate level
  – Register transfer level
  – Functional level
• Parameterized Static Elaboration
• Greatest Common Divisor
Static elaboration enables generation of hardware at synthesis time

We will look at two forms of static elaboration: (1) parameters and (2) generate blocks
Parameters are bound during static elaboration creating flexible modules

```verilog
module vcMux2
#( parameter WIDTH = 1 )
(
    input [WIDTH-1:0] in0, in1,
    input [1:0] sel,
    output reg [WIDTH-1:0] out
);

always @(*)
begin
    case ( sel )
    1'd0 : out = in0;
    1'd1 : out = in1;
    default : out = {WIDTH{1'bX}};
    endcase
end
endmodule
```

Instantiation Syntax

```verilog
vcMux2#(32) alu_mux
(
    .in0 (op1),
    .in1 (bypass),
    .sel (alu_mux_sel),
    .out (alu_mux_out)
);
```
Parameters are bound during static elaboration creating flexible modules

```verilog
module vcERDFF_pf
#( parameter WIDTH = 1,  
    parameter RESET_VALUE = 0 )
(
    input      clk,  
    input      reset,  
    input      [WIDTH-1:0] d,  
    input      en,  
    output reg [WIDTH-1:0] q
);

always @(posedge clk )
    if ( reset )
        q <= RESET_VALUE;
    else if ( en )
        q <= d;
endmodule
```

Instantiation Syntax

```verilog
vcERDFF_pf#(32,32'h10) pc_pf
(
    .clk   (clk),
    .reset (reset),
    .en    (pc_enable),
    .d     (pc_mux_out),
    .q     (pc)
);
```
Generate blocks can execute loops and conditionals during static elaboration

```verilog
module adder ( input [3:0] op1, op2,
              output cout,
              output [3:0] sum );

wire [4:0] carry;
assign carry[0] = 1'b0;
assign cout = carry[4];

genvar i;
generate
    for ( i = 0; i < 4; i = i+1 )
    begin : ripple
        FA fa( op1[i], op2[i], carry[i], carry[i+1] );
    end
endgenerate

endmodule
```

All genvars must be disappear after static elaboration

Generated names will have ripple[i]. prefix
Combining parameters + generate blocks enables more powerful elaboration

module adder#( parameter WIDTH = 1 )
(
    input  [WIDTH-1:0] op1,op2,
    output cout,
    output [WIDTH-1:0] sum
);

wire [WIDTH:0] carry;
assign carry[0] = 1'b0;
assign cout = carry[WIDTH];

genvar i;
genenerate
    for ( i = 0; i < WIDTH; i = i+1 )
        begin : ripple
            FA fa( op1[i], op2[i], carry[i], carry[i+1] );
        end
endgenerate

endmodule

Use parameter for loop bounds
Generate statements are useful for more than just module instantiation

```verilog
module adder#( parameter WIDTH = 1 )
(
    input [WIDTH-1:0] op1, op2,
    output cout,
    output [WIDTH-1:0] sum
);

wire [WIDTH:0] carry;
assign carry[0] = 1'b0;
assign cout = carry[WIDTH];

genvar i;
generate
    for ( i = 0; i < WIDTH; i = i+1 )
    begin : ripple
        assign {carry[i+1],sum[i]} = op1[i] + op2[i] + carry[i];
    end
endgenerate
endmodule
```

Statically elaborating many continuous assignments
Traditionally designers have resorted to behavioral inference for elaboration

module adder#( parameter WIDTH = 1 )
(
    input [WIDTH-1:0] op1, op2,
    output cout,
    output reg [WIDTH-1:0] sum
);

reg [WIDTH:0] carry;
assign cout = carry[WIDTH];

integer i;
always @(*)
begin
    assign carry[0] = 1'b0;
    for ( i = 0; i < WIDTH; i = i+1 )
        {carry[i+1],sum[i]} = op1[i] + op2[i] + carry[i];
end
endmodule

Although similar to generate block, this code has very different semantics!
Verilog Review

- Data types
- Structural Verilog
- Behavioral Verilog
  - Gate level
  - Register transfer level
  - Functional level
- Parameterized Static Elaboration
- Greatest Common Divisor
Functional GCD model is written within a single always block with C like structure

```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;
  end
endmodule
```

Test harness will simply set the input operands and check the output.
module exGCDTestHarness_behav;

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 );
extelse
        $display( "Test ( gcd(27,15) ) failed, [ %x != %x ]", out, 3 );

$finish;
end
endmodule
Behavioral GCD model is written within a single always block with C like structure

```
module 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
```

Our goal now is to design an RTL hardware block which implements this high-level behavior. What does the RTL implementation need?

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

- operands_rdy
- operands_val
- operands_bits_A
- operands_bits_B
- result_bits_data
- result_rdy
- result_val
- clk
- reset
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;
Next develop a datapath which has the proper functional units

\[
A = \text{inA}; 
B = \text{inB}; 
\]
\[
\text{while} \ ( \ !\text{done} \ ) 
\begin{align*}
&\text{if} \ ( \ A < B ) \\
&\quad \text{swap} = A; \\
&\quad A = B; \\
&\quad B = \text{swap}; \\
&\quad \text{else if} \ ( \ B \neq 0 ) \\
&\quad A = A - B; \\
&\quad \text{else} \\
&\quad \text{done} = 1; \\
&\end{align*}
\]
\[
Y = A;
\]
Next develop a datapath which has the proper functional units

\[
A = \text{inA}; \quad B = \text{inB}; \\
\text{while} \ ( \ !\text{done} \ ) \\
begi
\text{if} \ ( A < B ) \\
\quad \text{swap} = A; \\
\quad A = B; \\
\quad B = \text{swap}; \\
\text{else if} \ ( B != 0 ) \\
\quad A = A - B; \\
\text{else} \\
\quad \text{done} = 1; \\
\end{bega

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

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

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

    // Data signals
    input [W-1:0] operands_bits_A,
    input [W-1:0] operands_bits_B,
    output [W-1:0] result_bits_data,

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

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

Datapath module interface
Try to contain all functionality in leaf modules

```verilog
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)
);
```

A < B
B = 0
zero?
Try to contain all functionality in leaf modules

```verilog
code
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;
```

Using explicit state helps eliminate issues with non-blocking assignments

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

- WAIT
  - Waiting for new input operands
  -operands\_val

- CALC
  - Swapping and subtracting
  - !( A < B ) & ( B = 0 )

- DONE
  - Waiting for consumer to take the result
  - result\_rdy

- reset
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
reg [6:0] cs;

always @(*)
beg

// 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;
operands_rdy = 1'b0;
result_val   = 1'b0;

case ( state )

WAIT :
    ...
CALC :
    ...
DONE :
    ...
endcase
end

WAIT :
begin
    A_mux_sel    = A_MUX_SEL_IN;
    A_en         = 1'b1;
    B_mux_sel    = B_MUX_SEL_IN;
    B_en         = 1'b1;
    operands_rdy = 1'b1;
end

CALC :
if ( A_lt_B )
    A_mux_sel = A_MUX_SEL_B;
    A_en      = 1'b1;
    B_mux_sel = B_MUX_SEL_A;
    B_en      = 1'b1;
else if ( !B_zero )
    A_mux_sel = A_MUX_SEL_SUB;
    A_en      = 1'b1;
end

DONE :
result_val = 1'b1;
```
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 :
            if ( operands_val )
                state_next = CALC;
        CALC :
            if ( !A_lt_B && B_zero )
                state_next = DONE;
        DONE :
            if ( result_rdy )
                state_next = WAIT;
    endcase
end
```

[Diagram of the finite state machine with states WAIT, CALC, and DONE, and transitions based on operands_val, !A < B & B = 0, and result_rdy.]
RTL test harness requires properly handling the ready/valid signals
We can compare the functional and RTL implementations to verify correctness.
Take away points

- Structural Verilog for describing a hw schematic textually
- Verilog can model hardware at three levels of abstraction: gate level, register transfer level, and behavioral
- Understanding the Verilog execution semantics is critical for understanding blocking + non-blocking assignments
- Designers must have the hardware they are trying to create in mind when they write their Verilog
- Parameterized models provide the foundation for reusable libraries of components
- Begin your RTL design by identifying the external interface and then move on to partition your design into datapaths and control logic