Verilog HDL语言

主讲：李榕

Email: lr@hust.edu.cn

QQ: 179825425

华中科技大学计算机科学与技术学院

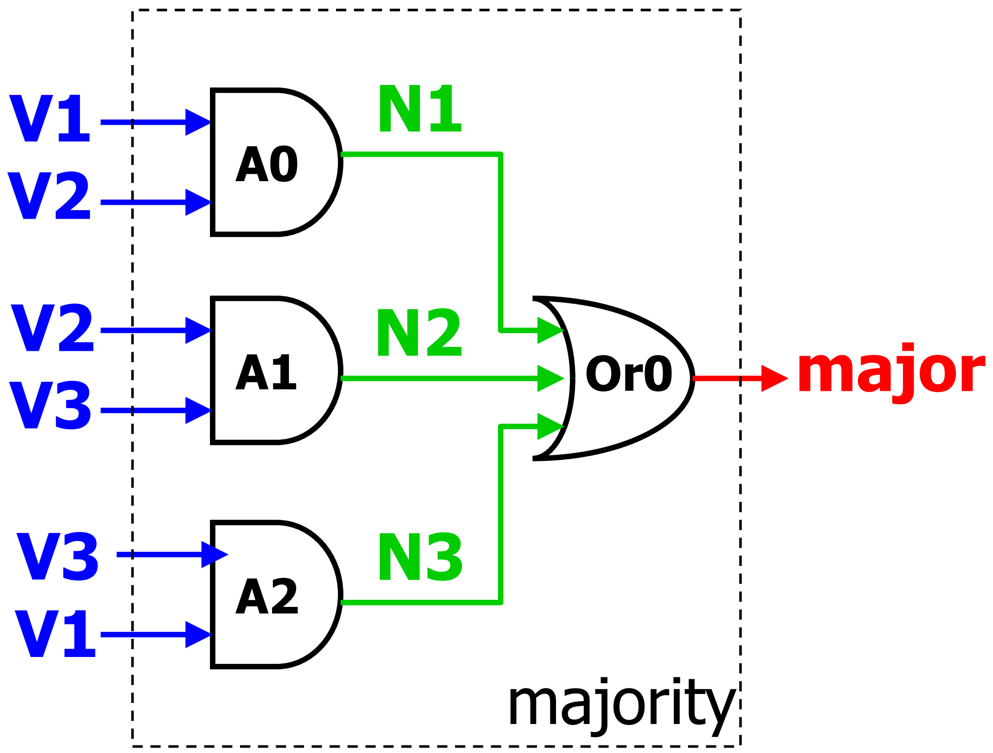
Verilog程序设计方法

Module Styles

- Modules can be specified different ways
  - Structural（结构描述） – connect primitives and modules
  - Dataflow（数据流描述）– use continuous assignments
  - Behavioral（行为描述） – use initial and always blocks
- A single module can use more than one method!
- What are the differences?

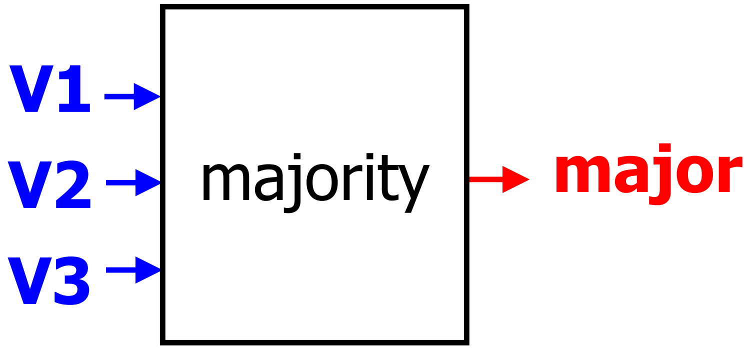
Structural Example

```
module majority (major, V1, V2, V3) ;

output major ;
input V1, V2, V3 ;

wire N1, N2, N3;

and A0 (N1, V1, V2),
    A1 (N2, V2, V3),
    A2 (N3, V3, V1);

or  Or0  (major, N1, N2, N3);

endmodule
```



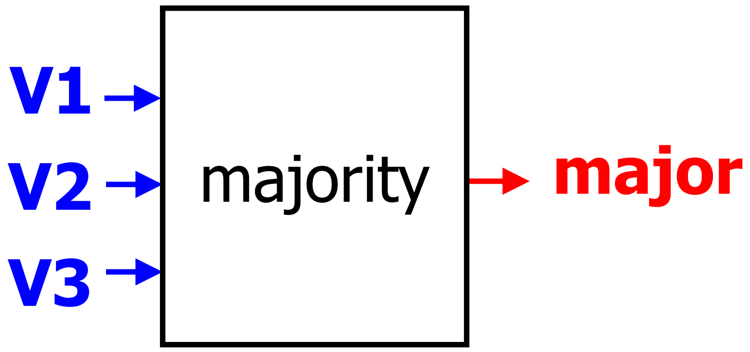
Dataflow Example

```
module majority (major, V1, V2, V3) ;

output major ;
input V1, V2, V3 ;

assign major = V1 & V2
               | V2 & V3
               | V1 & V3;
endmodule
```



Behavioral Example

```
module majority (major, V1, V2, V3) ;

output reg major ;
input V1, V2, V3 ;

always @(V1, V2, V3) begin
     if (V1 && V2 || V2 && V3
         || V1 && V3) major = 1;
     else major = 0;
end

endmodule
```



Full Adder: Structural

```
module half_add (X, Y, S, C);

input X, Y ;
output S, C ;

xor SUM (S, X, Y);
and CARRY (C, X, Y);

endmodule
```

```
module full_add (A, B, CI, S, CO) ;

input A, B, CI ;
output S, CO ;

wire S1, C1, C2;

// build full adder from 2 half-adders
half_add PARTSUM (A, B, S1, C1),
         SUM (S1, CI, S, C2);

// … add an OR gate for the carry
or CARRY (CO, C2, C1);

endmodule
```

Slide taken direct from Prof. Schulte

Full Adder: RTL/Dataflow

```
module fa_rtl (A, B, CI, S, CO) ;

input A, B, CI ;
output S, CO ;

// use continuous assignments
assign S = A ^  B ^ CI;
assign C0 = (A & B) | (A & CI) | (B & CI);

endmodule
```

Data flow Verilog is often very concise and still easy to read

Works great for most boolean and even datapath descriptions
（最适合组合逻辑设计）

Slide taken direct from Eric Hoffman

Full Adder: Behavioral

- Circuit “reacts” to given events (for simulation)
  - Actually list of signal changes that affect output

```
module fa_bhv (A, B, CI, S, CO) ;

input A, B, CI;
output S, CO;
reg S, CO;                    // assignment made in an always block
                              // must be made to registers
// use procedural assignments
always@(A or B or CI)
  begin
      S = A ^ B ^ CI;
      CO = (A & B) | (A & CI) | (B & CI);
   end
endmodule
```

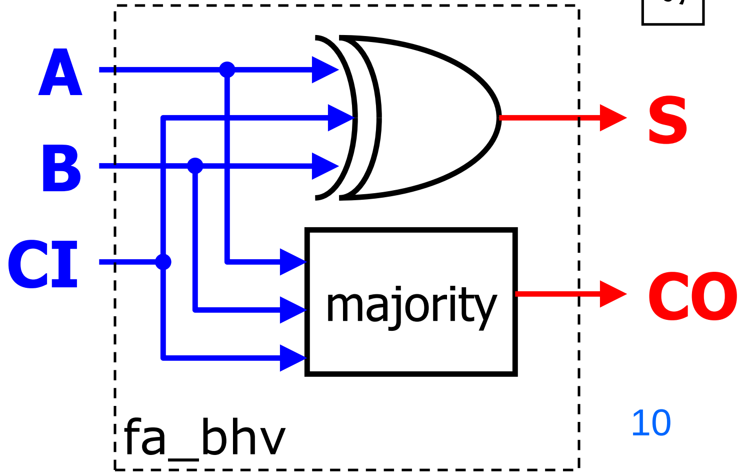
Slide taken direct from Prof. Schulte

Full Adder: Behavioral

- IN SIMULATION
  - When A, B, or C change, S and CO are recalculated
- IN REALITY
  - Combinational logic – no "waiting" for the trigger
  - *Constantly* computing - think transistors and gates!
  - Same hardware created for all three types of verilog

Slide taken direct from Prof. Schulte

```
always@(A or B or CI)
  begin
     S = A ^ B ^ CI;
    CO = (A & B) | (A & CI) | (B & CI);
  end
```



Structural Basics: Primitives

- Build design up from the gate/flip-flop/latch level
  - Structural verilog is a netlist of blocks and connectivity

- Verilog provides a set of gate primitives
  - and, nand, or, nor, xor, xnor, not, buf, bufif1, etc.
  - Combinational building blocks for structural design
  - Known “behavior”
  - Cannot access “inside” description

- Can also model at the transistor level
  - Most people don’t, we won’t

Primitives

- No declarations - can only be instantiated
- **Output port appears before input ports**
- Optionally specify: instance name and/or delay (discuss delay later)

```
and N25 (Z, A, B, C);     // name specified
and #10 (Z, A, B, X),
             (X, C, D, E);     // delay specified, 2 gates
and #10 N30 (Z, A, B); // name and delay specified
```

Syntax For Structural Verilog

- First declare the interface to the module
  - Module keyword, module name
  - Port names/types/sizes
- Next, declare any internal wires using “wire”
  - wire [3:0] partialsum;
- Then *instantiate* the primitives/submodules
  - Indicate which signal is on which port
- ModelSim Example with ring oscillator

Hierarchy

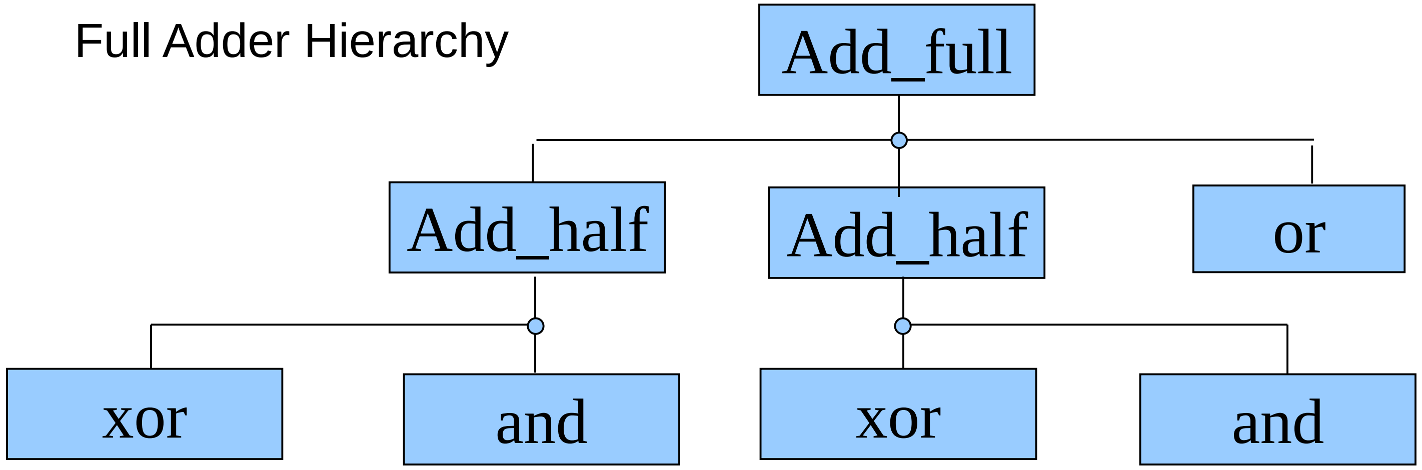
- Any Verilog design you do will be a module
- This includes testbenches!

- Interface (“black box” representation)
  - Module name, ports
- Definition
  - Describe functionality of the block
- Instantiation
  - Use the module inside another module

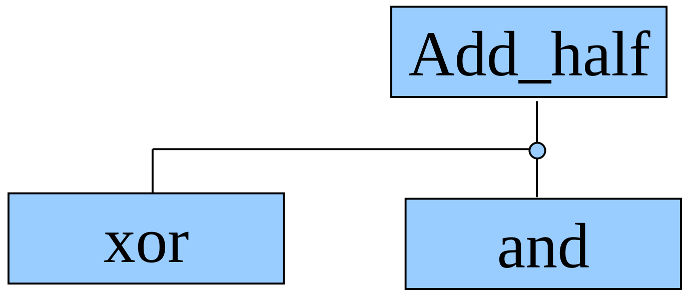
Hierarchy

- Build up a module from smaller pieces
  - Primitives
  - Other modules (which may contain other modules)
- Design: typically top-down
- Verification: typically bottom-up

Full Adder Hierarchy

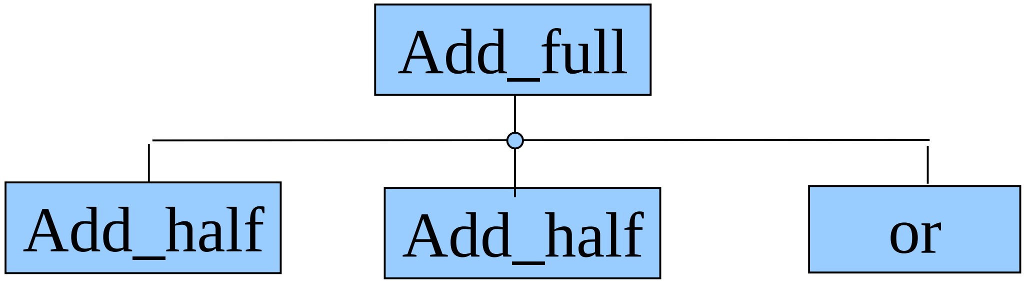


Add_half Module



```
module Add_half(c_out, sum, a, b);
    output sum, c_out;
    input a, b;

    xor sum_bit(sum, a, b);
    and carry_bit(c_out, a, b);
endmodule
```

Add_full Module



```
module Add_full(c_out, sum, a, b, c_in) ;
    output sum, c_out;
    input a, b, c_in;
    wire w1, w2, w3;

    Add_half AH1(.sum(w1), .c_out(w2), .a(a), .b(b));
    Add_half AH2(.sum(sum), .c_out(w3), .a(c_in), .b(w1));
    or carry_bit(c_out, w2, w3);
endmodule
```

Can Mix Styles In Hierarchy!

```
module Add_half_bhv(c_out, sum, a, b);
    output reg sum, c_out;
    input a, b;
    always @(a, b) begin
        sum = a ^ b;
        c_out = a & b;
    end
endmodule
```

```
module Add_full_mix(c_out, sum, a, b, c_in) ;
    output sum, c_out;
    input a, b, c_in;
    wire w1, w2, w3;
    Add_half_bhv AH1(.sum(w1), .c_out(w2),
                     .a(a), .b(b));
    Add_half_bhv AH2(.sum(sum), .c_out(w3),
                     .a(c_in), .b(w1));
    assign c_out = w2 | w3;
endmodule
```

Hierarchy And Scope

- Parent cannot access "internal" signals of child
- If you need a signal, must make a port!

Example:
Detecting overflow

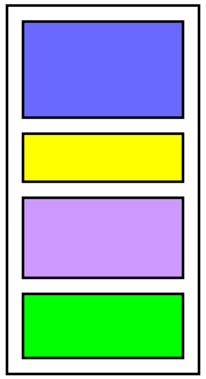
Overflow =
  cout XOR cout6

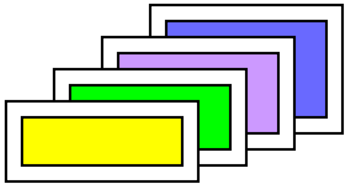
Must output
overflow or cout6!

```
module add8bit(cout, sum, a, b);
    output [7:0] sum;
    output cout;
    input [7:0] a, b;
    wire cout0, cout1,… cout6;
    FA A0(cout0, sum[0], a[0], b[0], 1'b0);
    FA A1(cout1, sum[1], a[1], b[1], cout0);
    …
    FA A7(cout, sum[7], a[7], b[7], cout6);
endmodule
```

Hierarchy And Source Code

- Can have all modules in a single file
  - Module order doesn't matter!
  - Good for <u>small</u> designs
  - Not so good for bigger ones
  - Not so good for module reuse (cut & paste)
- Can break up modules into multiple files
  - Helps with organization
  - Lets you find a specific module easily
  - Good for module reuse (add file to project)





Structural Verilog: Connections

- Positional or Connect by reference
  - Can be okay in some situations
    - Designs with very few ports
    - Interchangeable input ports (and/or/xor gate inputs)
  - Gets confusing for large #s of ports

```
module dec_2_4_en (A, E_n, D);

input [1:0] A;
input E_n;
output [3:0] D;
   .
   .
   .
```

```
wire [1:0] X;
wire W_n;
wire [3:0] word;

// instantiate decoder
dec_2_4_en DX (X,  W_n,  word);
```

Partial code instatiating decoder

Structural Verilog: Connections

- Explicit or Connect by name method
  - Helps avoid “misconnections”
  - Don’t have to remember port order
  - Can be easier to read
  - .<port name>(<signal name>)

```
module dec_2_4_en (A, E_n, D);

input [1:0] A;
input E_n;
output [3:0] D;
        .
        .
        .
```

```
wire [1:0] X;
wire W_n;
wire [3:0] word;

// instantiate decoder
dec_2_4_en DX
(.A(X), .E_n(W_n), .D(word));
```

Partial code instatiating decoder

Empty Port Connections

- Example: module dec_2_4_en(A, E_n, D);
  - dec_2_4_en DX (X[1:0], , word);   // E_n is high impedence (z)
  - dec_2_4_en DX (X[1:0], W_n ,);   // Outputs D[3:0] unused.

- General rules
  - Empty input ports => high impedance state (z)
  - Empty output ports => output not used
- Specify all input ports anyway!
  - Z as an input is very bad…why?

- Helps if no connection to output port name but leave empty:
  - dec_2_4_en DX(.A(X[3:2]), .E_n(W_n), .D());

Dataflow Verilog

- The continuous assign statement
  - It is the main construct of Dataflow Verilog
  - It is **deceptively** powerful & useful
- Generic form:

assign [drive_strength] [delay] list_of_net_assignments;

Where:
list_of_net_assignment ::= net_assignment [{,net_assignment}]
& Where:
Net_assignment ::= net_lvalue = expression

OK…that means just about nothing to me…how about some examples?

Continuous Assign Examples

- Simplest form:

```
// out is a net, a & b are also nets
assign out = a & b;      // and gate functionality
```

- Using vectors

```
wire [15:0] result, src1, src2;        // 3 16-bit wide vectors
assign result = src1 ^ src2;           // 16-bit wide XOR
```

- Can you implement a 32-bit adder in a single line?

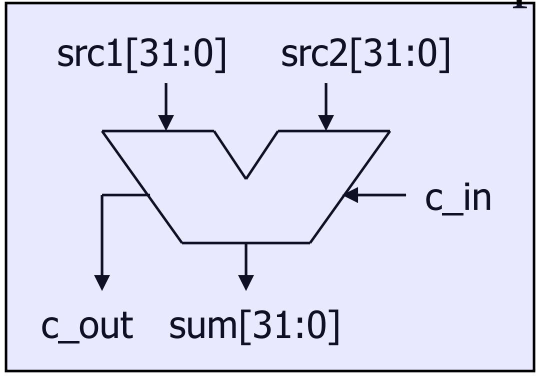
```
wire [31:0] sum, src1, src2;  // 3 32-bit wide vectors
assign {c_out,sum} = src1 + src2 + c_in; // wow!
```

Back to the Continuous Assign

- More basic generic form:

  assign <LHS> = <RHS expression>;

- If RHS result changes, LHS is updated with new value
  - Constantly operating (“continuous”)
  - It’s ***hardware!***
- RHS can use operators (i.e. +,-,&,|,^,~,>>,…)



➔

```
assign {c_out,sum} = src1 + src2 + c_in;
```

Lets Kick up the Horse Power

- You thought a 32-bit adder in one line was powerful. Lets try a 32-bit MAC…

Design a multiply-accumulate (MAC) unit that computes

    Z[31:0] = A[15:0]*B[15:0] + C[31:0]

It sets overflow to one, if the result cannot be represented using 32 bits.

```
module mac(output [31:0] Z, output overflow,
           input [15:0] A, B, input [15:0] C);
```

Lets Kick up the Horse Power

```
module mac(output [31:0] Z, output overflow,
           input [15:0] A, B, input [31:0] C);
  assign {overflow, Z} = A*B + C;
endmodule
```

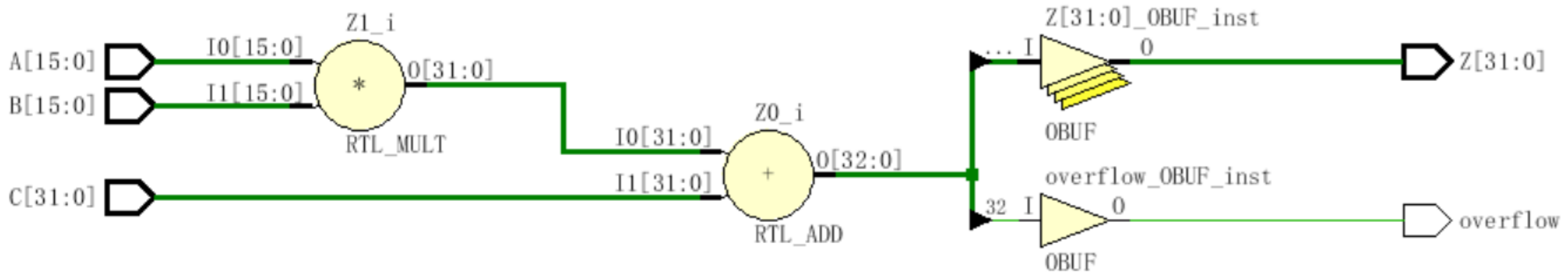
I am a brilliant genius. I am a HDL coder extraordinaire. I created a 32-bit MAC, and I did it in a single line.

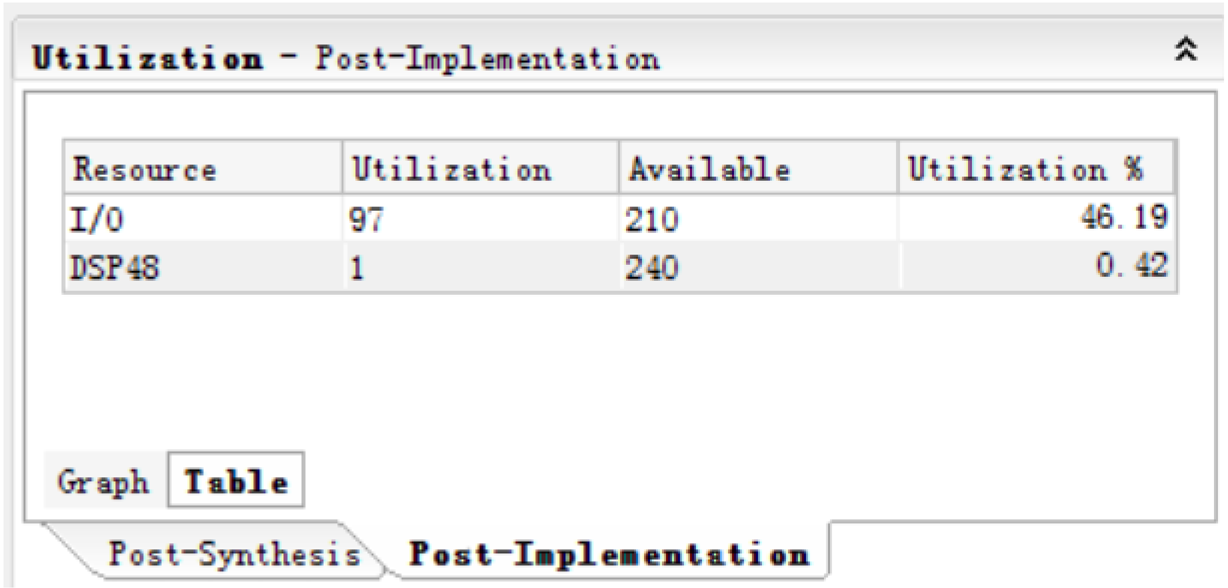
```
assign {overflow, Z} = A*B + C;
```

Synopsys

Oh my god, I've created a monster

```
module mac(output [31:0] Z, output overflow,
           input [15:0] A, B, input [31:0] C);
  assign {overflow, Z} = A*B + C;
endmodule
```



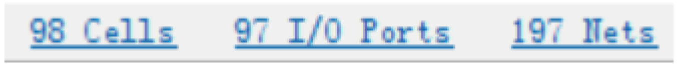


Utilization - Post-Implementation

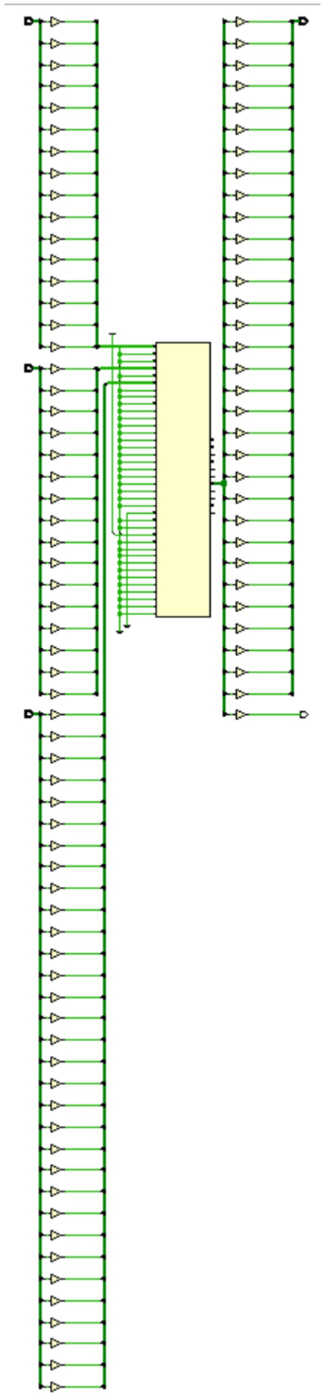
Resource	Utilization	Available	Utilization %
I/O	97	210	46.19
DSP48	1	240	0.42

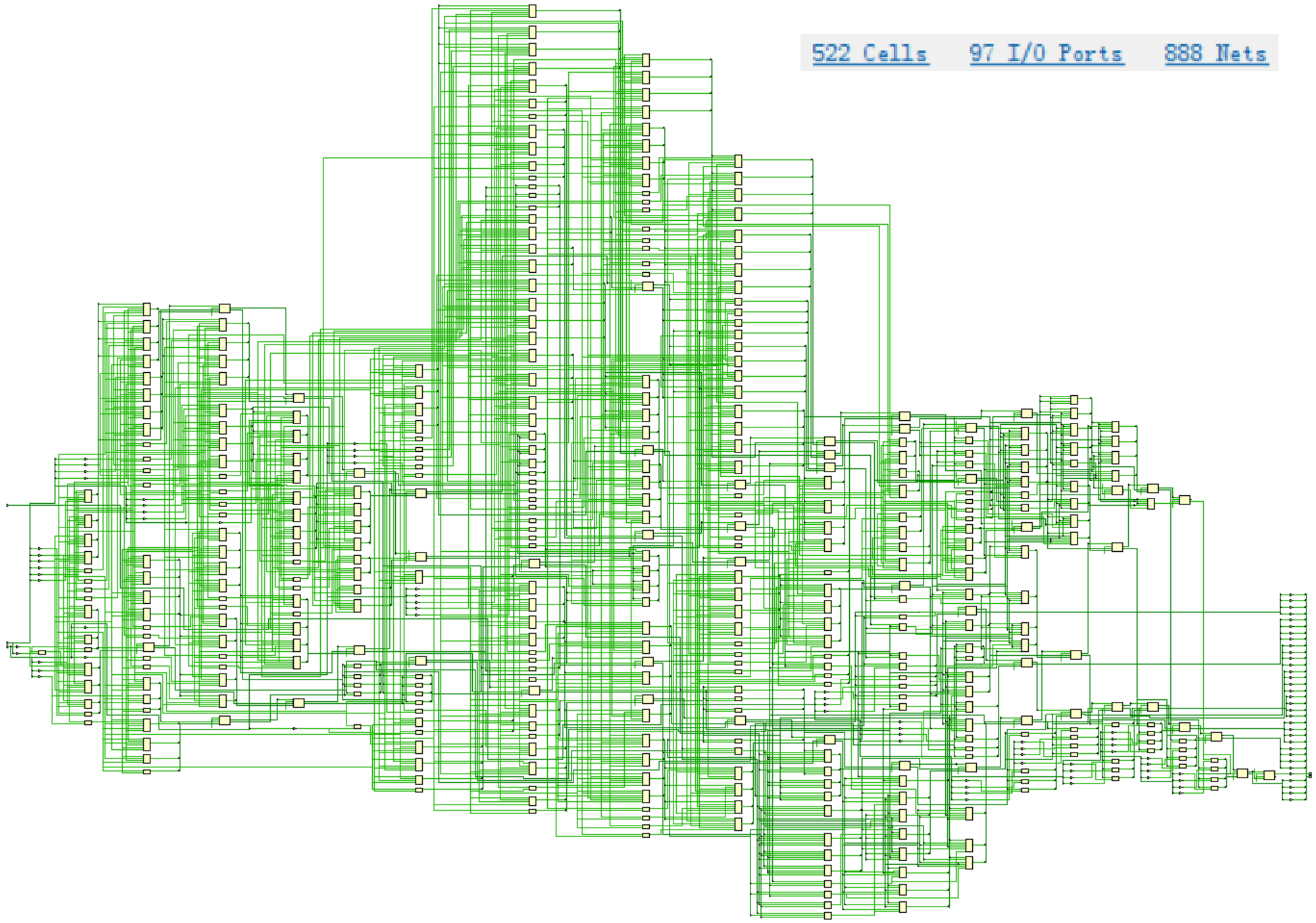
Graph | Table

Post-Synthesis | Post-Implementation



98 Cells 97 I/O Ports 197 Nets





522 Cells 97 I/O Ports 888 Nets

A Few Other Possibilities

- The implicit assign
  - Goes in the wire declaration

    ```
    wire [3:0] sum = a + b;
    ```

  - Can be a useful shortcut to make code succinct, but doesn't allow fancy LHS combos

    ```
    assign {cout, sum} = a + b + cin;
    ```

  - Personal choice
    - ✓You are welcome to use it when appropriate (**I never do!**)

Behavioral Verilog

- **initial** and **always** form basis of all behavioral Verilog
  - All other behavioral statements occur within these
  - **initial** and **always** blocks cannot be nested
  - All <LHS> assignments must be to type **reg**
- **initial** statements start at time 0 and execute once
  - If there are multiple **initial** blocks they all start at time 0 and execute independently. They may finish independently.
- If multiple behavioral statements are needed within the initial statement then the initial statement can be made compound with use of **begin/end**

More on **initial** statements

- Initial statement very useful for testbenches
- Initial statements **don’t** synthesize
- Don’t use them in DUT（Device Under Test，被测设备） Verilog (stuff you intend to synthesize)

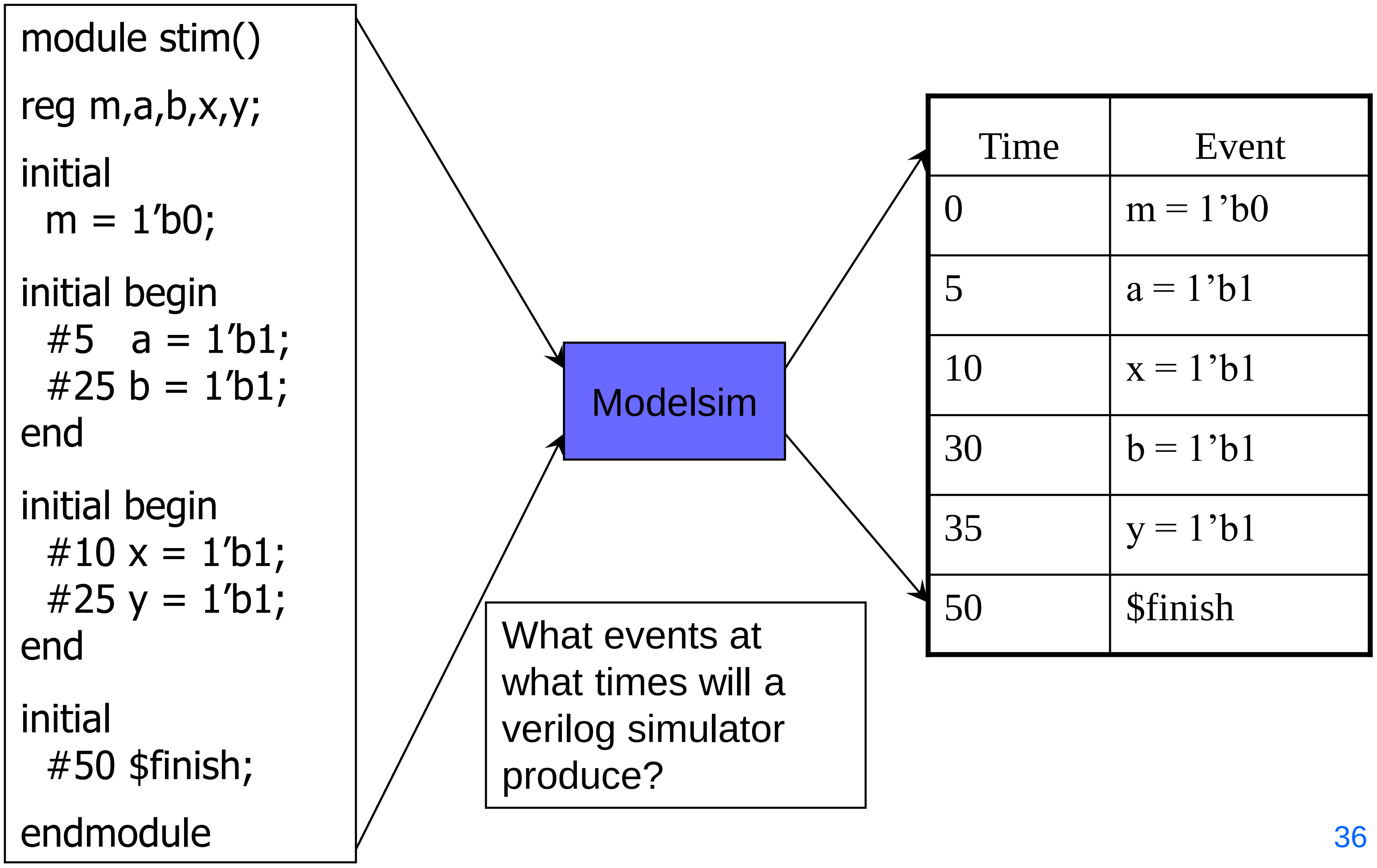
initial Blocks

```
`timescale 1 ns / 100 fs
module full_adder_tb;
    reg [3:0] stim;
    wire s, c;

    full_adder(sum, carry, stim[2], stim[1], stim[0]);        // instantiate DUT

    // monitor statement is special - only needs to be made once,
    initial $monitor("%t: s=%b c=%b stim=%b", $time, s, c, stim[2:0]);

    // tell our simulation when to stop
    initial #50 $stop;

    initial begin     // stimulus generation
        for (stim = 4'h0; stim < 4'h8; stim = stim + 1) begin
            #5;
        end
    end
endmodule
```

all initial blocks start at time 0

*multi-statement block enclosed by **begin** and **end***

Another **initial** Statement Example



```
module stim()

reg m,a,b,x,y;

initial
  m = 1'b0;

initial begin
  #5   a = 1'b1;
  #25 b = 1'b1;
end

initial begin
  #10 x = 1'b1;
  #25 y = 1'b1;
end

initial
  #50 $finish;

endmodule
```

Time	Event
0	m = 1'b0
5	a = 1'b1
10	x = 1'b1
30	b = 1'b1
35	y = 1'b1
50	$finish

What events at what times will a verilog simulator produce?

always statements

- Behavioral block operates CONTINUOUSLY
  - Executes at time zero but loops continuously
  - Can use a *trigger list* to control operation; @(a, b, c)
  - In absense of a trigger list it will re-evaluate when the last <LHS> assignment completes.

```
module clock_gen (output reg clock);

initial
  clock = 1'b0;                // must initialize in initial block

always                         // no trigger list for this always
  #10 clock = ~clock;          // always will re-evaluate when
                               // last <LHS> assignment completes
endmodule
```

always vs **initial**

```
reg [7:0] v1, v2, v3, v4;

initial begin
       v1 = 1;
    #2 v2 = v1 + 1;
       v3 = v2 + 1;
    #2 v4 = v3 + 1;
       v1 = v4 + 1;
    #2 v2 = v1 + 1;
       v3 = v2 + 1;
end
```

```
reg [7:0] v1, v2, v3, v4;

always begin
       v1 = 1;
    #2 v2 = v1 + 1;
       v3 = v2 + 1;
    #2 v4 = v3 + 1;
       v1 = v4 + 1;
    #2 v2 = v1 + 1;
       v3 = v2 + 1;
end
```

- **What values does each block produce?**
  - **Lets take our best guess**
  - **Then lets try it in a** simulator

Trigger lists (Sensitivity lists)

- Conditionally "execute" inside of **always** block
  - Any change on trigger (sensitivity) list, triggers block

    always @(a, b, c) **begin**

    …

    end

- Original way to specify trigger list

  always @ (X1 **or** X2 **or** X3)

- In Verilog 2001 can use **,** instead of **or**

  always @ (X1, X2, X3)

- Verilog 2001 also has ***** for *combinational only*

  always @ (*)

Some mixed simulation tools in use today still do not support Verilog 2001.

Do you know your design?

Example: Comparator

```
module compare_4bit_behave(output reg A_lt_B, A_gt_B, A_eq_B,
                           input [3:0] A, B);

  always@(      ) begin




  end

endmodule
```

Flush out this template with sensitivity list and implementation
Hint: a if...else if...else statement is best for implementation

- if-else 条件语句

```
if (条件表达式)          块语句1
else if (条件表达式2)   块语句2
……
else if (条件表达式n)   块语句n
else                    块语句n+1
```

- case 语句

```
case (敏感表达式)
   值1:        块语句1
   值2:        块语句2
   ……
   值n:        块语句n
   default:   块语句n+1
endcase
```

- for循环语句

```
for (表达式1；表达式2；表达式3）块语句
```

if…else if…else statement

- General forms…

```
If (condition) begin
   <statement1>;
   <statement2>;
end
```

```
If (condition)
  begin
    <statement1>;
    <statement2>;
  end
else
  begin
    <statement3>;
    <statement4>;
  end
```

```
If (condition)
  begin
    <statement1>;
    <statement2>;
  end
else if (condition2)
  begin
    <statement3>;
    <statement4>;
  end
else
  begin
    <statement5>;
    <statement6>;
  end
```

Of course the compound statements formed with **begin/end** are optional.

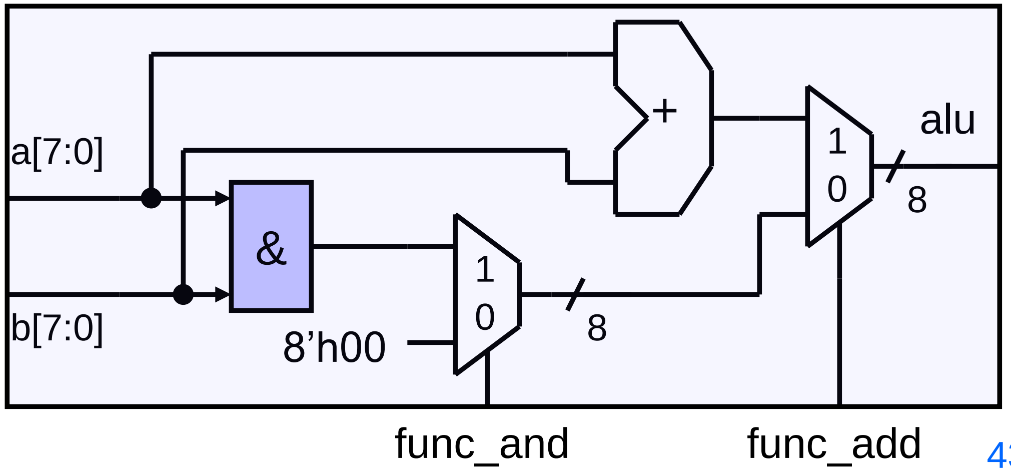
Multiple else if's can be strung along indefinitely

How does and if…else if…else statement synthesize?

- Does not conditionally “execute” block of “code”
- Does not conditionally create hardware!
- It makes a <u>multiplexer</u> or selecting logic
- Generally:
  - ✓Hardware for both paths is created
  - ✓Both paths “compute” simultaneously
  - ✓The result is selected depending on the condition

```
if (func_add)
   alu = a + b;
else if (func_and)
   alu = a & b;
else
   alu = 8'h00;
```

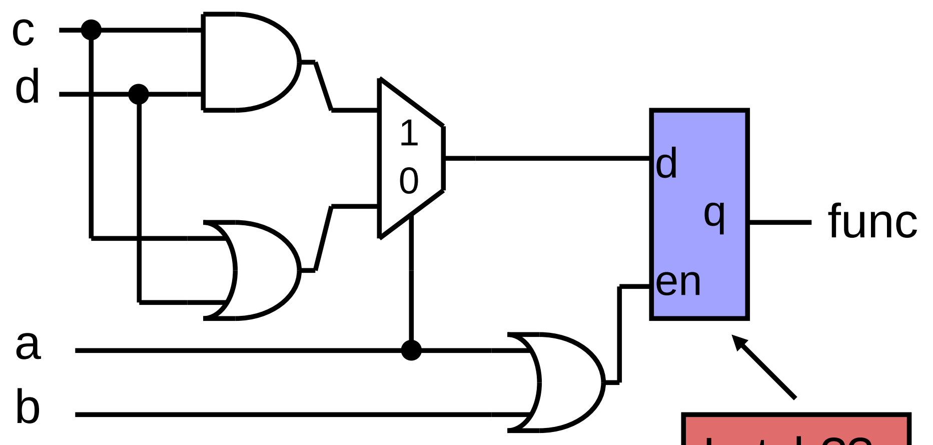
$\longrightarrow$



if statement synthesis (continued)

```
if (a)
  func = c & d;
else if (b)
  func = c | d;
```

How does this synthesize?



Latch??

What you ask for is what you get!

func is of type register. When neither **a** or **b** are asserted it didn't not get a new value.

That means it must have remained the value it was before.

That implies memory…i.e. a **latch!**

Always have an **else** to any **if** to avoid unintended latches.

More on **if** statements…

- Watch the sensitivity lists…what is missing in this example?

```
always @(a, b) begin
    temp = a – b;
    if ((temp < 8'b0) && abs)
            out = -temp;
    else out = temp;
end
```

```
always @  (posedge clk) begin
            if (reset) q <= 0;
            else if (set) q <= 1;
            else q <= data;
end
```

What is being coded here?

Is it synchrounous or asynch?

Does the reset or the set have higher priority?

case Statements

- Verilog has three types of case statements:
  - ➉ **case**, **casex**, and **casez**
- Performs bitwise match of expression and case item
  - Both <u>must</u> have same bitwidth to match!
- **case**
  - Can detect **x** and **z**! (good for testbenches)
- **casez**
  - Uses **z** and **?** as “don’t care” bits in case items and expression
- **casex**
  - Uses **x**, **z**, and **?** as “don’t care” bits in case items and expression

Slide taken direct from Eric Hoffman

Case statement (general form)

```
case (expression)
   alternative1 : statement1;         // any of these statements could
   alternative2 : statement2;         // be a compound statement using
   alternative3 : statement3;         // begin/end
   default : statement4               // always use default for synth stuff
endcase
```

```
parameter AND = 2'b00;
parameter OR = 2'b01;
parameter XOR = 2'b10;

case (alu_op)
   AND            : alu = src1 & src2;
   OR             : alu = src1 | src2;
   XOR            : alu = src1 ^ src2;
   default        : alu = src1 + src2;
endcase
```

Why always have a default?

Same reason as always having an else with an if statement.

All cases are specified, therefore no unintended latches.

Slide taken direct from Eric Hoffman

Using **case** To Detect **x** And **z**

- Only use this functionality in a testbench!
- Example taken from Verilog-2001 standard:

```
case (sig)
        1'bz:           $display("Signal is floating.");
        1'bx:           $display("Signal is unknown.");
        default:        $display("Signal is %b.", sig);
endcase
```

Slide taken direct from Eric Hoffman

casex Statement

- Uses **x**, **z**, and **?** as single-bit wildcards in case item and expression
- Uses first match encountered

```
always @ (code) begin
        casex (code)                              // case expression
                2'b0?: control = 8'b00100110; // case item1
                2'b10: control = 8'b11000010; // case item 2
                2'b11: control = 8'b00111101; // case item 3
        endcase
end
```

- What is the output for code = 2'b01?
- What is the output for code = 2'b1x?

Slide taken direct from Eric Hoffman

casez Statement

- Uses z, and ? as single-bit wildcards in case item and expression

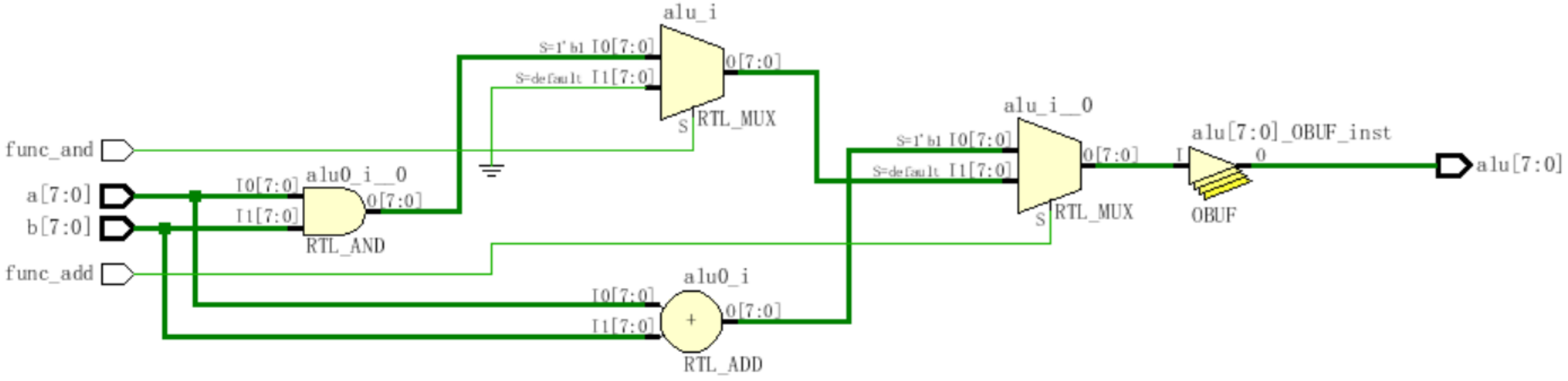
```
always @ (code) begin
    casez (code)
            2'b0?: control = 8'b00100110; // item 1
            2'bz1: control = 8'b11000010; // item 2
            default: control = 8b'xxxxxxxx; // item 3

    endcase
end
```

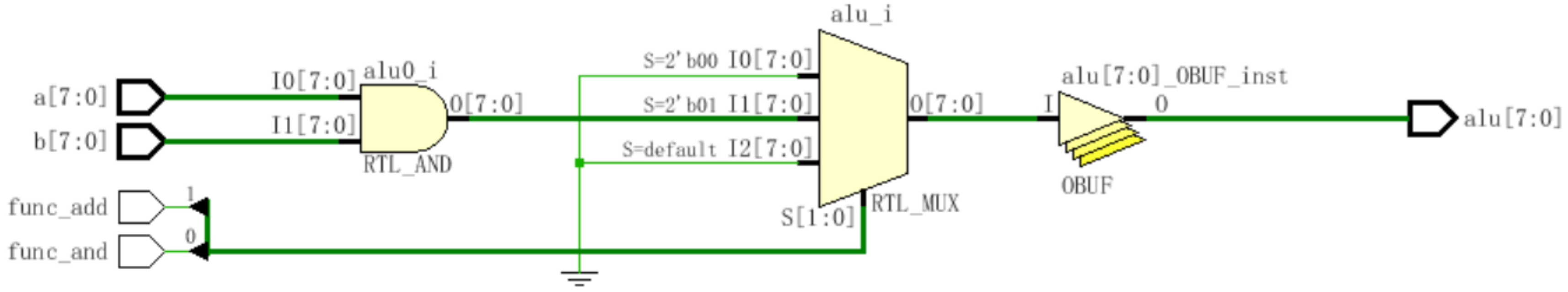
- What is the output for code = 2b'01?
- What is the output for code = 2b'zz?

Slide taken direct from Eric Hoffman

```
if (func_add)
   alu = a + b;
else if (func_and)
   alu = a & b;
else
   alu = 8’h00;
```

```
case({func_add, func_and})
    2'b00: alu = 8'h00;
    2'b01: alu = a & b;
    2'b1x: alu = func_add;
    default: alu = 8'h00;
endcase
```





Loops in Verilog

- We already saw the **for** loop:

```
reg [15:0] rf[0:15];          // memory structure for modeling register file
reg [5:0] w_addr;             // address to write to

for (w_addr=0; w_addr<16; w_addr=w_addr+1)
   rf[w_addr[3:0]] = 16’h0000;      // initialize register file memory
```

- There are 3 other loops available:
  - While loops
  - Repeat loop
  - Forever loop

while loops

- Executes until boolean condition is not true
  - ❿ If boolean expression false from beginning it will never execute loop

```
reg [15:0] flag;
reg [4:0] index;

initial begin
   index=0;
   found=1'b0;
   while ((index<16) && (!found)) begin
      if (flag[index]) found = 1'b1;
      else index = index + 1;
   end
   if (!found) $display("non-zero flag bit not found!");
   else $display("non-zero flag bit found in position %d",index);
end
```

Handy for cases where loop termination is a more complex function.

Like a search

repeat Loop

- Good for a fixed number of iterations
  - ➓ Repeat count can be a variable but…
    - ✓ It is only evaluated when the loops starts
    - ✓ If it changes during loop execution it won’t change the number of iterations
- Used in conjunction with @(posedge clk) it forms a handy & succinct way to wait in testbenches for a fixed number of clocks

```
initial begin
   inc_DAC = 1’b1;
   repeat(4095) @(posedge clk);   // bring DAC right up to point of rollover
   inc_DAC = 1’b0;
   inc_smpl = 1’b1;
   repeat(7)@(posedge clk);       // bring sample count up to 7
   inc_smpl = 1’b0;
end
```

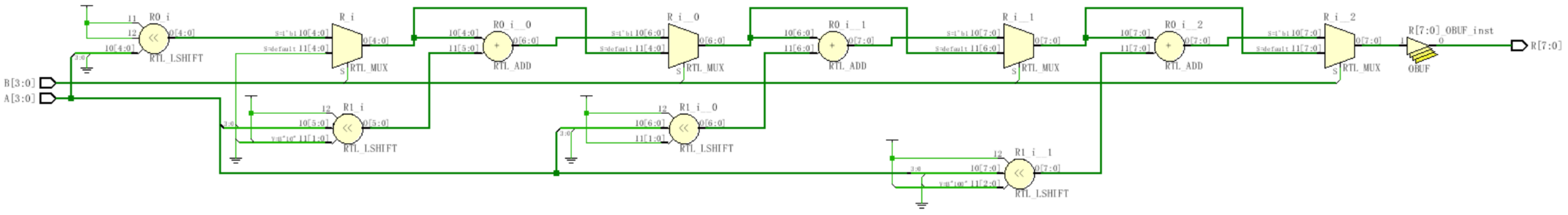
forever loops

- We got a glimpse of this already with clock generation in testbenches.
- Only a **$stop**, **$finish** or a specific **disable** can end a **forever** loop.

```
initial begin
  clk = 0;
  forever #10 clk = ~ clk;
end
```

Clock generator is by far the most common use of a forever loop

```
module mult4b(output reg [7:0] R, input [3:0] A, input [3:0] B);
    integer i;

    always @(A, B) begin
        R = 0;
        for(i=1; i<=4; i=i+1)
            if(B[i-1]) R = R + (A<<i);
        end
endmodule
```



- 赋值语句

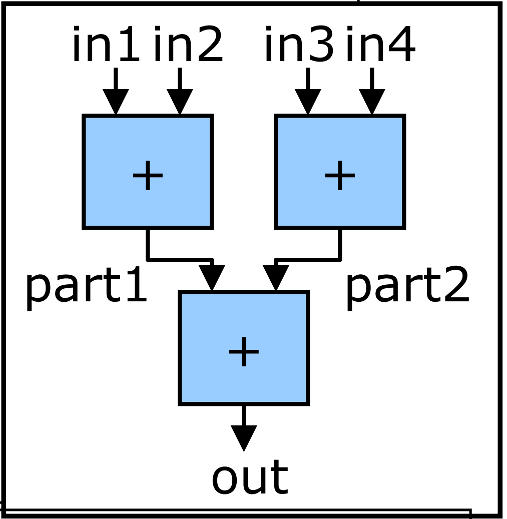
基本形式：<寄存器变量>=<表达式>;
外部模式：<定时控制> <寄存器变量>=<表达式>;
内部模式：<寄存器变量>= <定时控制> <表达式>;
定时控制分为两类：延时控制
　　　　　　　　　事件控制 @(事件控制敏感表）

- 阻塞型过程赋值
  - 赋值算符“=”：前一条语句没有完成赋值过程之前，后面的语句不可能被执行。
- 非阻塞型过程赋值
  - 赋值算符“<=”：一条非阻塞型赋值语句的执行，并不会影响块中其它语句的执行。
- 连续赋值语句
  - 只要输入端操作数的值发生变化，该语句就重新计算并刷新赋值结果。用关键词**assign**来区分。

Blocking vs non-Blocking

- Blocking “Evaluated” sequentially
- Works a lot like software **(danger!)**
- Used for combinational logic

```
module addtree(output reg [9:0] out,
                      input [7:0] in1, in2, in3, in4);

reg [8:0] part1, part2;
always @(in1, in2, in3, in4) begin
        part1 = in1 + in2;
        part2 = in3 + in4;
        out = part1 + part2;
end
endmodule
```

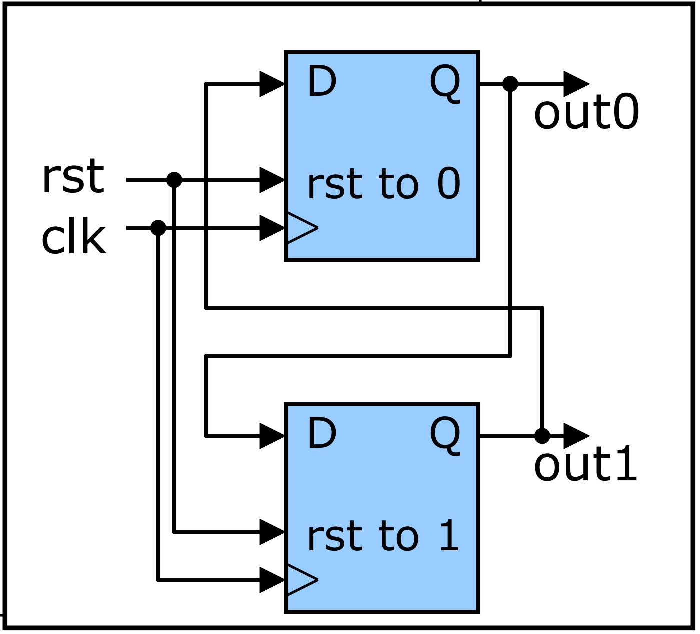


Slide taken direct from Eric Hoffman

Non-Blocking Assignments

- "Updated" <u>simultaneously</u> if no delays given
- Used for <u>sequential</u> logic

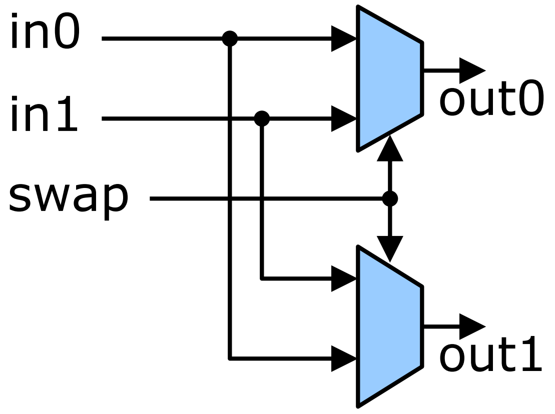
```
module swap(output reg out0, out1, input rst, clk);

always @(posedge clk) begin
        if (rst) begin
                out0 <= 1'b0;
                out1 <= 1'b1;
        end
        else begin
                out0 <= out1;
                out1 <= out0;
        end
end
endmodule
```



Swapping if done in Blocking

- In blocking, need a "temp" variable

```
module swap(output reg out0, out1, input in0, in1, swap);

reg temp;
always @(*) begin
        out0 = in0;
        out1 = in1;
        if (swap) begin
                temp = out0;
                out0 = out1;
                out1 = temp;
        end
end
endmodule
```

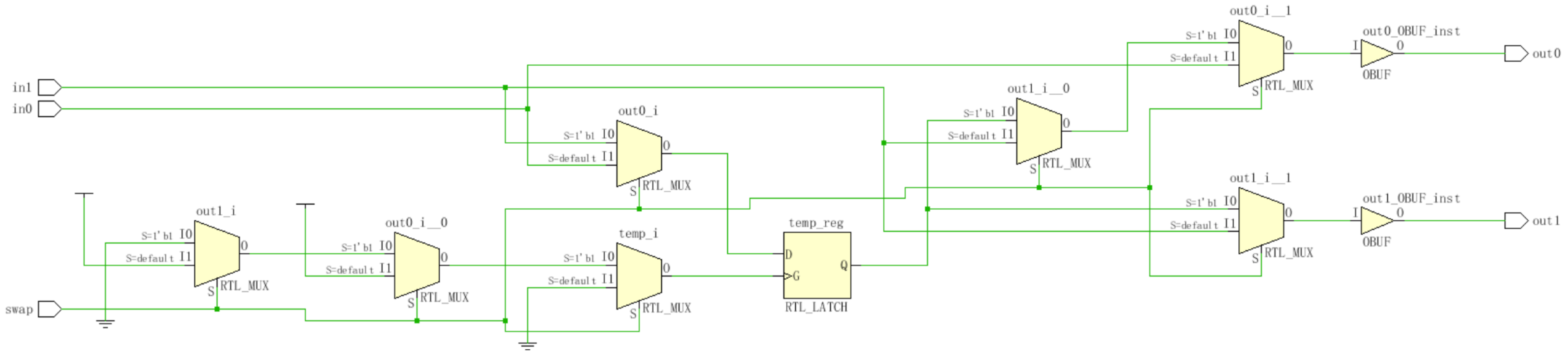


Which values get included on the sensitivity list from *?

Slide taken direct from Eric Hoffman

Swapping if done in Non-Blocking

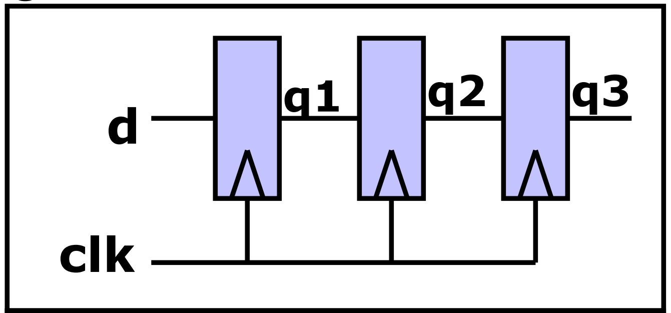
```
module swap(output reg out0, out1, input in0, in1, swap);
   reg temp;
   always @(*) begin
         out0 <= in0;
         out1 <= in1;
         if (swap) begin
                  temp <= out0;
                  out0 <= out1;
                  out1 <= temp;
         end
   end
endmodule
```



More on Blocking

- Called blocking because….
  - The evaluation of subsequent statements <RHS> are **blocked**, until the <LHS> assignment of the current statement is completed.

```
module pipe(clk, d, q);

input clk,d;
output q;
reg q;

always @(posedge clk) begin
  q1 = d;
  q2 = q1;
  q3 = q2;
end

endmodule
```



Lets code this

Simulate this in your head…

Remember blocking behavior of: <LHS> assigned before <RHS> of next evaluated.

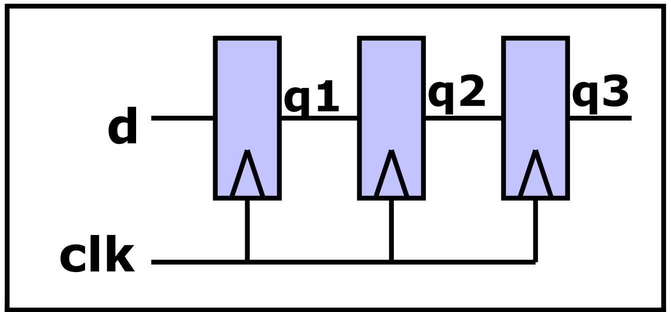
Does this work as intended?

Slide taken direct from Eric Hoffman

More on Non-Blocking

- Lets try that again

```
module pipe(clk, d, q);

input clk,d;
output q;
reg q;

always @(posedge clk) begin
  q1 <= d;
  q2 <= q1;
  q3 <= q2;
End

endmodule;
```



Lets code this

With non-blocking statements the <RHS> of subsequent statements are **not blocked**. They are all evaluated simultaneously.

The assignment to the <LHS> is then scheduled to occur.

This will work as intended.

Slide taken direct from Eric Hoffman

So Blocking is no good and we should always use Non-Blocking??

- Consider combinational logic

```
module ao4(z,a,b,c,d);

input a,b,c,d;
output z;

reg z,tmp1,tmp2;

always @(a,b,c,d) begin
  tmp1 <= a & b;
  tmp2 <= c & d;
  z <= tmp1 | tmp2;
end
endmodule
```

Does this work?

The inputs (a,b,c,d) in the sensitivity list change, and the always block is evaluated.

New assignments are scheduled for tmp1 & tmp2 variables.

A new assignment is scheduled for z using the **previous** tmp1 & tmp2 values.

Slide taken direct from Eric Hoffman

Why not non-Blocking for Combinational

- Can we make this example work?

```
module ao4(z,a,b,c,d);

input a,b,c,d;
output z;

reg z,tmp1,tmp2;

always @(a,b,c,d) begin
  tmp1 <= a & b;
  tmp2 <= c & d;
  z <= tmp1 | tmp2;
end
endmodule
```

$\xrightarrow{\text{Yes}}$

Put tmp1 & tmp2 in the trigger list

```
module ao4(z,a,b,c,d);

input a,b,c,d;
output z;

reg z,tmp1,tmp2;

always @(a,b,c,d,tmp1,tmp2) begin
  tmp1 <= a & b;
  tmp2 <= c & d;
  z <= tmp1 | tmp2;
end
endmodule
```

What is the downside of this?

Slide taken direct from Eric Hoffman

- 任务（task）
- 函数（function）
- 任务和函数结构之间的差异：
  - 一个任务块可以含有时间控制结构，而函数块则没有；
  - 一个任务块可以有输入和输出，而函数块必须有至少一个输入，没有任何输出；
  - 任务块的引发是通过一条语句，而函数块只有当它被引用在一个表达式中时才会生效。

```
task 任务名；
      端口与类型说明；
      局部变量说明；
      块语句
endtask
```

```
function <位宽说明> 函数名；
      输入端口与类型说明；
      局部变量说明；
      块语句
endfunction
```

functions

- Declared and referenced within a module
- Used to implement combinational behavior
  - Contain no timing controls or tasks
- Inputs/outputs
  - Must have at least one input argument
  - Has only one output (no inouts)
  - Function name is implicitly declared return variable
  - Type and range of return value can be specified (1-bit wire is default)

Slide taken direct from Eric Hoffman

When to use functions?

- Usage rules:
  - ➓May be referenced in any expression (RHS)
  - ➓May call other functions
- Requirements of procedure (implemented as function)
  - No timing or event control
  - Returns a single value
  - Has at least 1 input
  - Uses only behavioral statements
  - Only uses blocking assignments (combinational)
- Mainly useful for conversions, calculations, and selfchecking routines that return boolean. (testbenches)

Slide taken direct from Eric Hoffman

Function Example

```
module word_aligner (word_out, word_in);
  output          [7: 0]    word_out;
  input           [7: 0]    word_in;
  assign word_out = aligned_word(word_in);   // invoke function

  function        [7: 0]    aligned_word;        // function declaration
    input         [7: 0]    word;
    begin
      aligned_word = word;
      if (aligned_word != 0)
        while (aligned_word[7] == 0) aligned_word = aligned_word << 1;
    end
  endfunction
endmodule
```

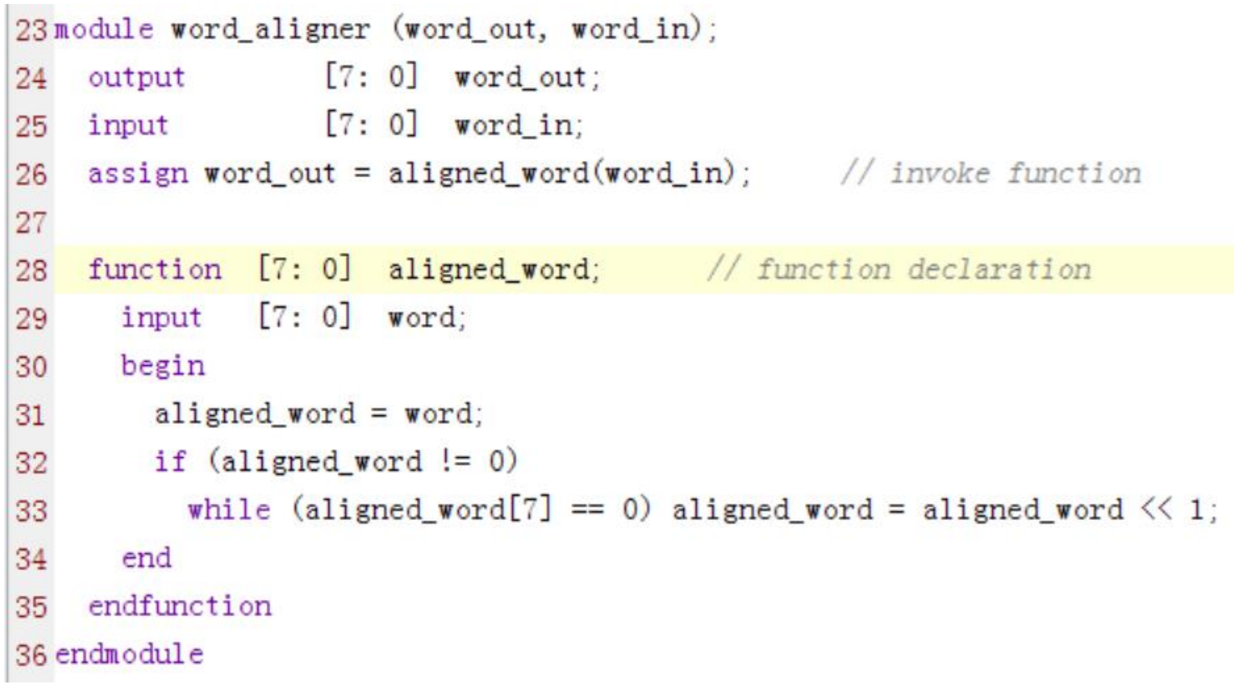
size of return value

input to function

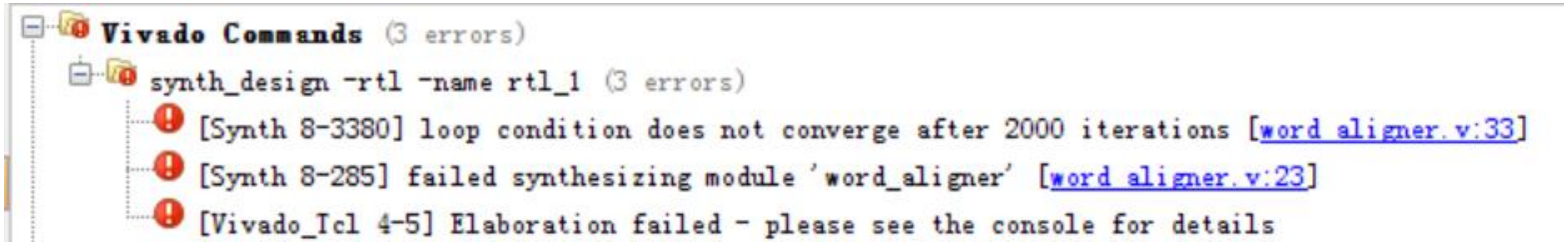
Does this synthsize?

Slide taken direct from Eric Hoffman

Function Example



```
23 module word_aligner (word_out, word_in);
24   output          [7: 0]   word_out;
25   input           [7: 0]   word_in;
26   assign word_out = aligned_word(word_in);      // invoke function
27
28   function  [7: 0]   aligned_word;        // function declaration
29     input    [7: 0]   word;
30     begin
31       aligned_word = word;
32       if (aligned_word != 0)
33         while (aligned_word[7] == 0) aligned_word = aligned_word << 1;
34     end
35   endfunction
36 endmodule
```



```
Vivado Commands (3 errors)
  synth_design -rtl -name rtl_1 (3 errors)
    [Synth 8-3380] loop condition does not converge after 2000 iterations [word_aligner.v:33]
    [Synth 8-285] failed synthesizing module 'word_aligner' [word_aligner.v:23]
    [Vivado_Tcl 4-5] Elaboration failed - please see the console for details
```

Function Example [2]

```
module arithmetic_unit (result_1, result_2, operand_1, operand_2,);
  output                    [4: 0] result_1;
  output          [3: 0] result_2;
  input           [3: 0] operand_1, operand_2;
  assign result_1 = sum_of_operands (operand_1, operand_2);
  assign result_2 = larger_operand (operand_1, operand_2);

  function [4: 0] sum_of_operands(input [3:0] operand_1, operand_2);
    sum_of_operands = operand_1 + operand_2;
  endfunction

  function [3: 0] larger_operand(input [3:0] operand_1, operand_2);
    larger_operand = (operand_1 >= operand_2) ? operand_1 : operand_2;
  endfunction
endmodule
```

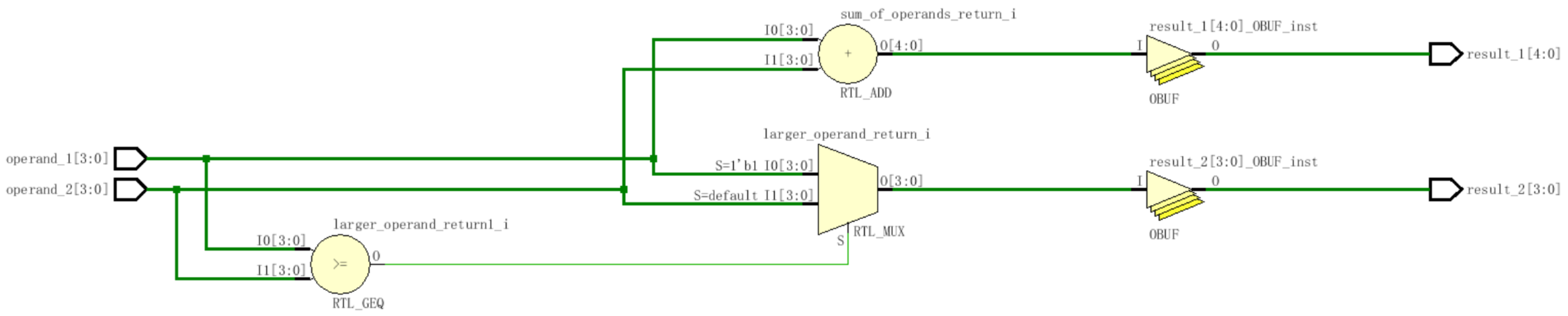
function call

function inputs

function output

Slide taken direct from Eric Hoffman

Function Example [2]



Re-Entrant (recursive) functions

- Use keyword **automatic** to enable stack saving of function working space and enable recursive functions.

```
module top;

//Define the factorial function
function automatic integer factorial;
input [31:0] oper;

begin
  if (oper>=2)
     factorial = factorial(oper-1)*oper;
  else
     factorial = 1;
end

endfunction
```

```
initial begin
  result = factorial(4);
  $display("Result is %d",result);
end

endmodule
```

Is this how you would do it?

KISS

Slide taken direct from Eric Hoffman

tasks (much more useful than functions)

Functions:	**Tasks:**
A function can enable another function, but not another task	A task can enable other tasks and functions
Functions must execute in zero delay	Tasks may execute in non-zero simulation time
Functions can have no timing or even control statements	Tasks may contain delay, event or timing control. (i.e. ➔ @, #)
Function must have at least one input argument.	Task may have zero or more arguments of type input, output, or inout
Fucntions always return a single value. The cannot have output or inout arguments	Task do not return a value, but rather pass multiple values through output and input arguments

Tasks can modify global signals too, perhaps naughty, but I do it all the time.

Slide taken direct from Eric Hoffman

Why use Tasks?

- Tasks provide the ability to
  - ❿ Execute common procedures from multiple places
  - ❿ Divide large procedures into smaller ones
- Local variables can be declared & used
- Personally, I only use tasks in testbenches, but they are very handy there.
  - Break common testing routines into tasks
    - ✓ Initialization tasks
    - ✓ Stimulus generation tasks
    - ✓ Self Checking tasks
  - Top level test then becomes mainly calls to tasks.

Slide taken direct from Eric Hoffman

Task Example [Part 1]

```
module adder_task (c_out, sum, clk, reset, c_in, data_a, data_b);
  output reg    [3: 0]   sum;
  output reg             c_out;
  input [3: 0]  data_a, data_b;
  input         clk, reset, c_in;

  always @(posedge clk or posedge reset) begin
    if (reset) {c_out, sum} <= 0;
    else add_values (sum, c_out, data_a, data_b, c_in); // invoke task
  end
// Continued on next slide
```

Calling of task

Slide taken direct from Eric Hoffman

Task Example [Part 2]

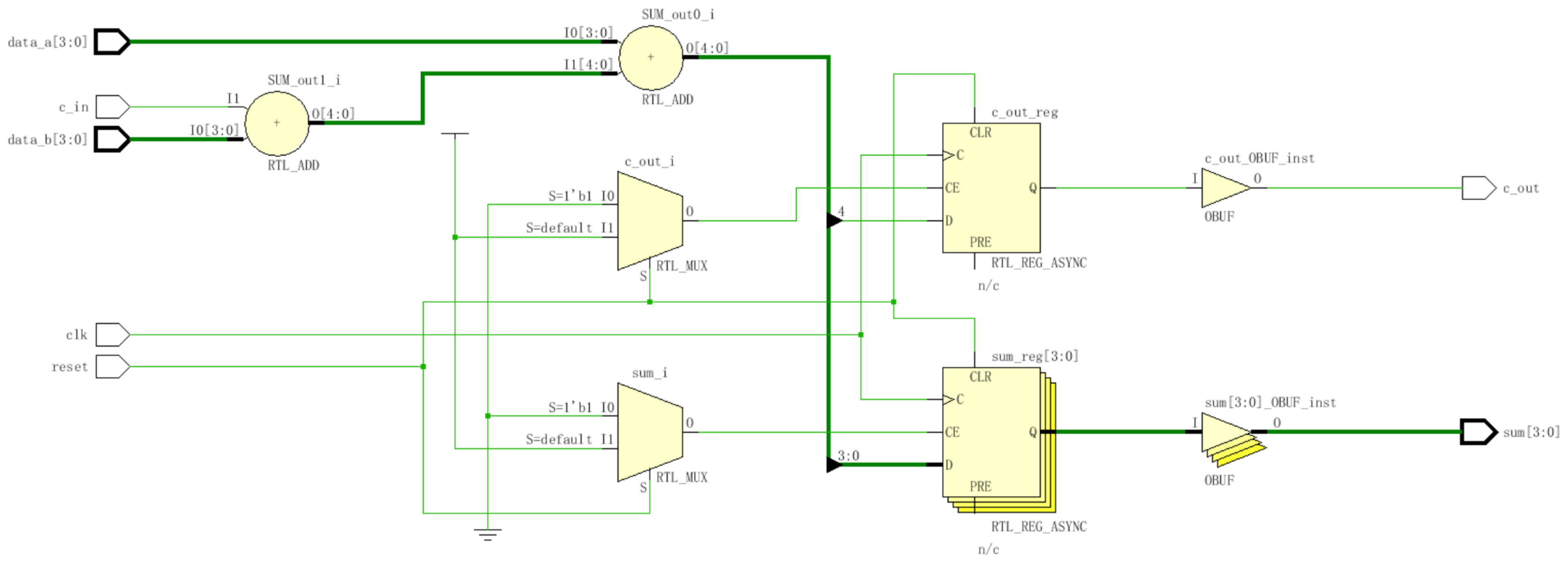
```
// Continued from previous slide
task add_values;          // task declaration
    output reg  [3: 0]  SUM;                    } task outputs
    output reg          C_OUT;
    input       [3: 0]  DATA_A, DATA_B;         } task inputs
    input               C_IN;
                {C_OUT, SUM} = DATA_A + (DATA_B + C_IN);
endtask
endmodule
```

- Could have instead specified inputs/outputs using a port list.

```
task add_values (output reg [3: 0] SUM, output reg C_OUT,
                         input [3:0] DATA_A, DATA_B, input C_IN);
```

Slide taken direct from Eric Hoffman

Task Example [Part 3]



Task Example [2]

```
task leading_1(output reg [2:0] position, input [7:0]
    data_word);
     reg          [7:0] temp;
     begin
           temp = data_word;
           position = 7;
           while (!temp[7]) begin
               temp = temp << 1;
               position = position - 1;
            end
      end
endtask
```

internal task variable

NOTE:
"while" loops usually not synthesizable!

- What does this task assume for it to work correctly?
- How do tasks differ from modules?
- How do tasks differ from functions?

Slide taken direct from Eric Hoffman

时序控制

- 由“#”符号引入的延迟控制。它将程序的执行过程中断一定的时间，时间的长度由<time>的值来确定。

```
# <time> <statement> ;
```

- 由“@”符号引入的事件控制。一个事件可以通过运行表达式“->event”变量被激发。

```
@ (<posedge>|<negedge>|<signals> ) <statement>;
```

- 等待语句。直到表达式计算为真之前，都延时下一个语句的执行。

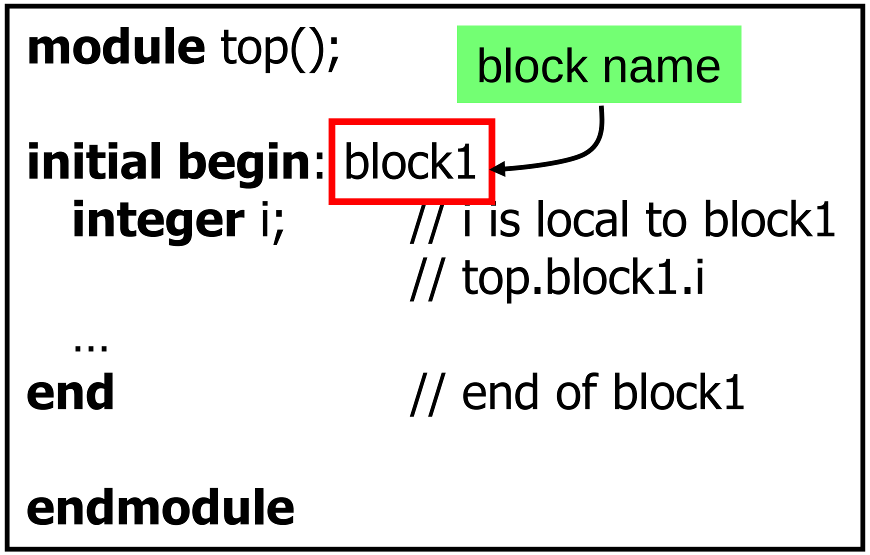
```
wait (<expression>) <statement> ;
```

- 延迟定义块。对模块中某一指定的路径进行延迟定义，这一路径连接模块的输入端口与输出端口（或双向端口）。

```
specify
    (<expression>) = <time>
    ……
endspecify
```

Named Blocks

- Blocks (**begin/end**) or (**fork/join**) can be named
  - Local variables can be declared for the named block
  - Variables in a named block can be accessed using hierarchical naming reference
  - Named blocks can be disabled (i.e. execution stopped)



```
module top();

initial begin: block1
   integer i;          // i is local to block1
                       // top.block1.i
   …
end                    // end of block1

endmodule
```

disable Statement

- Similar to the “break” statement in C
  - Disables execution of the current block (not permanently)

```
begin : break
   for (i = 0; i < n; i = i+1) begin : continue
      @(posedge clk)
      if (a == 0) // "continue" loop
         disable continue;
      if (a == b) // "break" from loop
         disable break;
      statement1
      statement2
   end
end
```

What occurs if (a==0)?

What occurs if (a==b)?

How do they differ?

File I/O – Why?

- If we don’t want to hard-code all information in the testbench, we can use input files

- Help automate testing
  - One file with inputs
  - One file with expected outputs

- Can have a software program generate data
  - Create the inputs for testing
  - Create “correct” output values for testing
  - Can use files to “connect” hardware/software system

Opening/Closing Files

- **$fopen** opens a file and returns an integer descriptor

  integer fd = **$fopen**(“filename”);

  integer fd = **$fopen**(“filename”, r);

  - If file cannot be open, returns a 0
  - Can output to more than one file simultaneously by writing to the OR (|) of the relevant file descriptors
    - ✓ Easier to have “summary” and “detailed” results

- **$fclose** closes the file

  $fclose(fd);

Writing To Files

- Output statements have file equivalents
  - ✓ **$fmonitor**()
  - ✓ **$fdisplay**()
  - ✓ **$fstrobe**()
  - ✓ **$fwrite**()   // write is like a display without the \n

- These system calls take the file descriptor as the first argument
  - ✓ **$fdisplay(fd, "out=%b in=%b", out, in);**

Reading From Files

- Read a binary file: $fread(destination, fd);
  - Can specify start address & number of locations too
  - Good luck! I have never used this.
- Very rich file manipulation (see IEEE Standard)
  - ✓ $fseek(), $fflush(), $ftell(), $rewind(), …
- Will cover a few of the more common read commands next

Using **$fgetc** to read characters

```
module file_read()

parameter EOF = -1;

integer file_handle,error,indx;
reg signed [15:0] wide_char;
reg [7:0] mem[0:255];
reg [639:0] err_str;

initial begin
  indx=0;
  file_handle = $fopen("text.txt","r");
  error = $ferror(file_handle,err_str);
  if (error==0) begin
    wide_char = 16'h0000;
    while (wide_char!=EOF) begin
      wide_char = $fgetc(file_handle);
      mem[indx] = wide_char[7:0];
      $write("%c",mem[indx]);
      indx = indx + 1;
    end
  end
  else $display("Can't open file…");
  $fclose(file_handle);
end
endmodule
```

The quick brown fox jumped
over the lazy dogs

text.txt

When finished the array *mem* will contain the characters of this file one by one, and the file will have been echoed to the screen.

Why wide_char[15:0] and why signed?

Using **$fgets** to read lines

```
module file_read2()

integer file_handle,error,indx,num_bytes_in_line;
reg [256*8:1] mem[0:255],line_buffer;
reg [639:0] err_str;

initial begin
   indx=0;
   file_handle = $fopen("text2.txt","r");
   error = $ferror(file_handle,err_str);
   if (error==0) begin
      num_bytes_in_line = $fgets(line_buffer,file_handle);
      while (num_bytes_in_line>0) begin
         mem[indx] = line_buffer;
         $write("%s",mem[indx]);
         indx = indx + 1;
         num_bytes_in_line = $fgets(line_buffer,file_handle);
      end
   end
   else $display("Could not open file text2.txt");
```

$fgets() returns the number of bytes in the line. When this is a zero you know you hit EOF.

Using **$fscanf** to read files

```
module file_read3()

integer file_handle,error,indx,num_matches;
reg [15:0] mem[0:255][1:0];
reg [639:0] err_str;

initial begin
   indx=0;
   file_handle = $fopen("text3.txt","r");
   error = $ferror(file_handle,err_str);
   if (error==0) begin
      num_matches = $fscanf(file_handle,"%h %h",mem[indx][0],mem[indx][1]);
      while (num_matches>0) begin
         $display("data is: %h %h",mem[indx][0],mem[indx[1]);
         indx = indx + 1;
         num_matches = $fscanf(file_handle,"%h %h",mem[indx][0],mem[indx][1]);
      end
   end
   else $display("Could not open file text3.txt");
```

```
12f3   13f3
abcd   1234
3214   21ab
```

text3.txt

Loading Memory Data From Files

- This is very useful (memory modeling & testbenches)
  - $readmemb("<file_name>",<memory>);
  - $readmemb("<file_name>",<memory>,<start_addr>,<finish_addr>);
  - $readmemh("<file_name>",<memory>);
  - $readmemh("<file_name>",<memory>,<start_addr>,<finish_addr>);

- **$readmemh** ➔ Hex data…**$readmemb** ➔ binary data
  - But they are reading ASCII files either way (just how numbers are represented)

```
// addr     data
@0000 10100010
@0001 10111001
@0002 00100011
```

example "binary" file

```
// addr     data
@0000   A2
@0001   B9
@0002   23
```

example "hex" file

```
//data
A2
B9
23
```

address is optional for the lazy

Example of **$readmemh**

```
module rom(input clk; input [7:0] addr; output [15:0] dout);

reg [15:0] mem[0:255];      // 16-bit wide 256 entry ROM
reg [15:0] dout;

initial
   $readmemh(“constants”,mem);

always @(negedge clk) begin
   ///////////////////////////////////////////////////////////
   // ROM presents data on clock low //
   /////////////////////////////////////////////////////////
   dout <= mem[addr];
end

endmodule
```

‘Include Compiler Directives

- **`include** filename
  - Inserts entire contents of another file at compilation
  - Can be placed anywhere in Verilog source
  - Can provide either relative or absolute path names

- Example 1:

```
module use_adder8(…);
`include “adder8.v”  // include the task for adder8
```

- Example 2:

```
module cppc_dig_tb();
`include “/home/ehoffman/ece551/project/tb_tasks.v”
```

- Useful for including tasks and functions in multiple modules