Verilog HDL语言

Verilog程序设计方法

主讲：卢　萍

华中科技大学计算机科学与技术学院

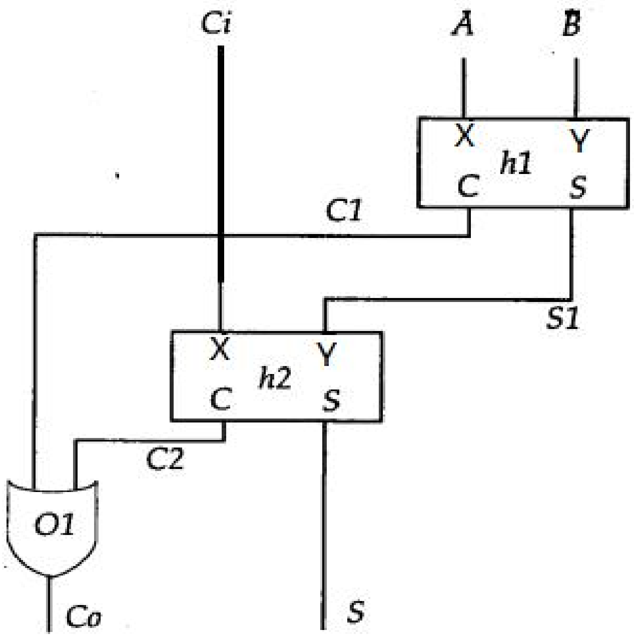
Module Styles

- Modules can be specified different ways
  - Structural – connect primitives and modules
  - Dataflow– use **assign** continuous assignments

    assign <LHS net> = <RHS expression>;
  - Behavioral – use **initial** and **always** blocks
- A single module can use more than one method!

- What are the differences?

Full Adder: Structural



使用两个半加器模块构造全加器

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

input X, Y ;
output S, C ;

xor SUM (S, X, Y);  // 异或门实例语句
and CARRY (C, X, Y);  // 与门实例语句

endmodule
```

```
module full_add ( input A, B, CI, output S, CO) ;
  wire S1, C1, C2;

  half_add  h1 (A, B, S1, C1),
            h2 (S1, CI, S, C2);     //两个 半加器模块实例语句
  or CARRY (CO, C2, C1);            // 或门实例语句
endmodule
```

Full Adder: 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 CO = (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

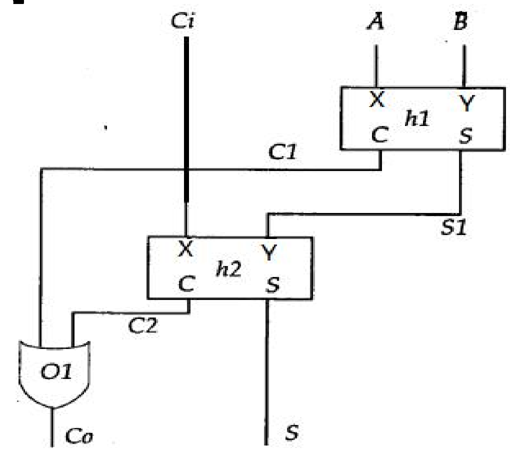
Full Adder: Behavioral

- 只要有事件发生（列表中任何 信号有变化），就执行begin...end 的语句 (for simulation)

  ---- 实际上，列表中信号的改变影响输出

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

input A, B, CI;
output S, CO;
reg S, CO;            // 只有reg型能够在 always 语句中被赋值

always@(A or B or CI)  // 由事件控制的always语句
  begin
     S = A ^ B ^ CI;          // 过程性赋值语句,，必须在 always 过程块中
     CO = (A & B) | (A & CI) | (B & CI);
  end
endmodule
```

Mix Styles




```
module Add_half_bhv(C, S, X, Y);
    output reg S, C;
    input X, Y;
    always @(X, Y) begin
        S = X ^ Y;
        C = X & Y;
    end
endmodule
```

```
module Add_full_mix(CO, S, A, B, Ci) ;
    output S, CO;
    input A, B, Ci;
    wire s1, c1, c2;
    Add_half_bhv h1(.S(s1), .C(C1), .X(A), .Y(B));
    Add_half_bhv h2(.S(S), .C(C2), .X(Ci), .Y(S1));
    assign CO = C1 | C2;
endmodule
```

原语(Primitives)

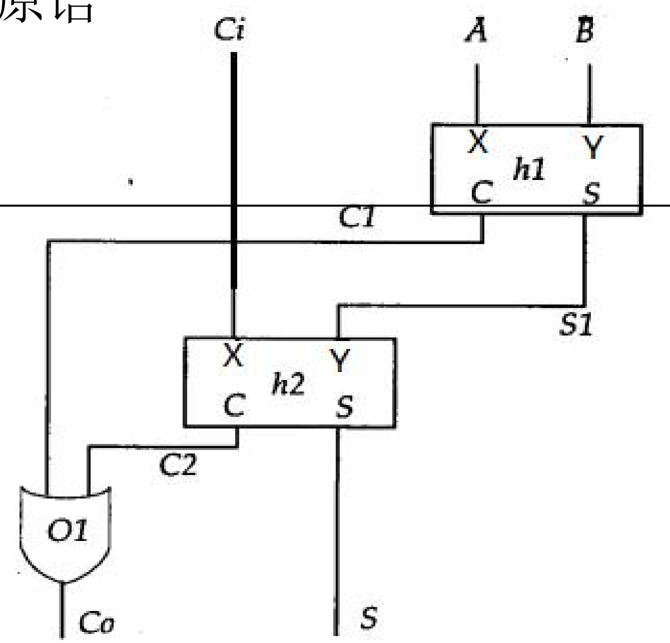
- 不用声明，直接实例化
- 输出端口在前，输入端口在后
- 实例名和时延可选

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

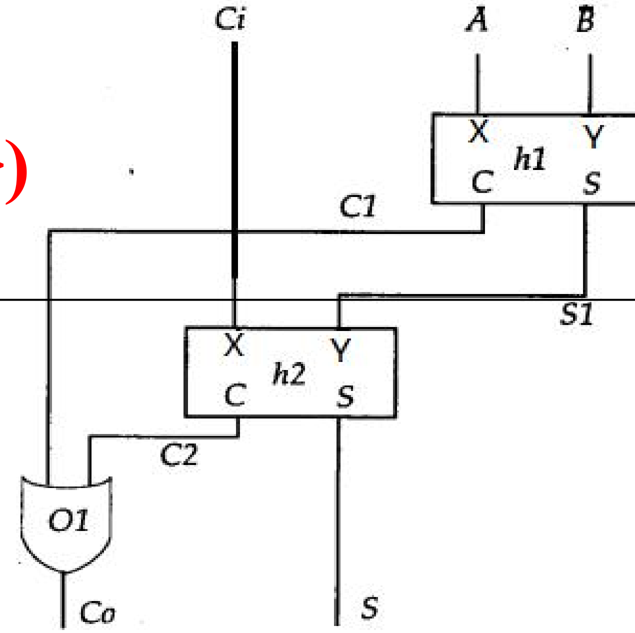
Syntax For Structural Verilog

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

 wire S1, C1, C2;

 half_add  h1 (A, B, S1, C1),   //实例化两个 half_add半加器子模块
                 h2 (CI, S1, S, C2);   //   端口位置关联
 or CARRY (CO, C2, C1);          // 实例化或门原语

endmodule
```

```
module half_add (X, Y, S, C);
  input X, Y ;
  output S, C ;
  ....
endmodule
```



.<port name>(<signal name>)




```
module Add_full(Co,S, A, B, Ci) ;
    output Co,S;
    input A, B, Ci;
    wire S1, C1, C2;

    Add_half h1( .C(C1), .S(S1), .X(A), .Y(B)); /* 端口名称关联 */
    Add_half h2( .C(C2), .S(S), .X(Ci), .Y(S1));
    or carry_bit(Co, C1, C2);   /* 端口位置关联 */
endmodule
```

```
module half_add (X, Y, S, C);
  input X, Y ;
  output S, C ;
  ....
endmodule
```

Empty Port Connections

- Example: **module** Add_full(Co,S, A, B, Ci) ;
  - Add_full AF(Cout,Sum, a, , cin) ; // B is high impedence (z)
  - Add_full AF(,Sum, a, b , cin) ; // Outputs Co 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?

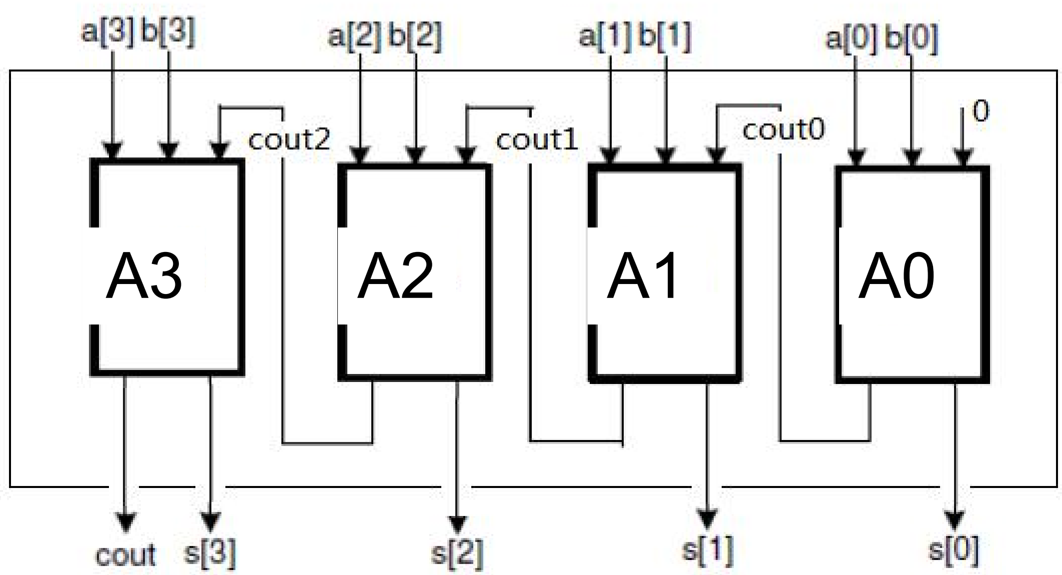
- Helps if no connection to output port name but leave empty:
  - Add_full AF(**.Co( )**,.S(Sum),.A(a),.b(B) , .Ci(cin)) ;

```
module Add4bit(cout, s, a, b);
    output [3:0] s;
    output cout;
    input [3:0] a, b;
    wire cout0, cout1, cout2; //  内部信号
    Add_full A0(cout0, s[0], a[0], b[0], 1'b0);
    Add_full A1(cout1, s[1], a[1], b[1], cout0);
    Add_full A2(cout2, s[2], a[2], b[2], cout1);
    Add_full A3(cout, s[3], a[3], b[3], cout2);
endmodule
```

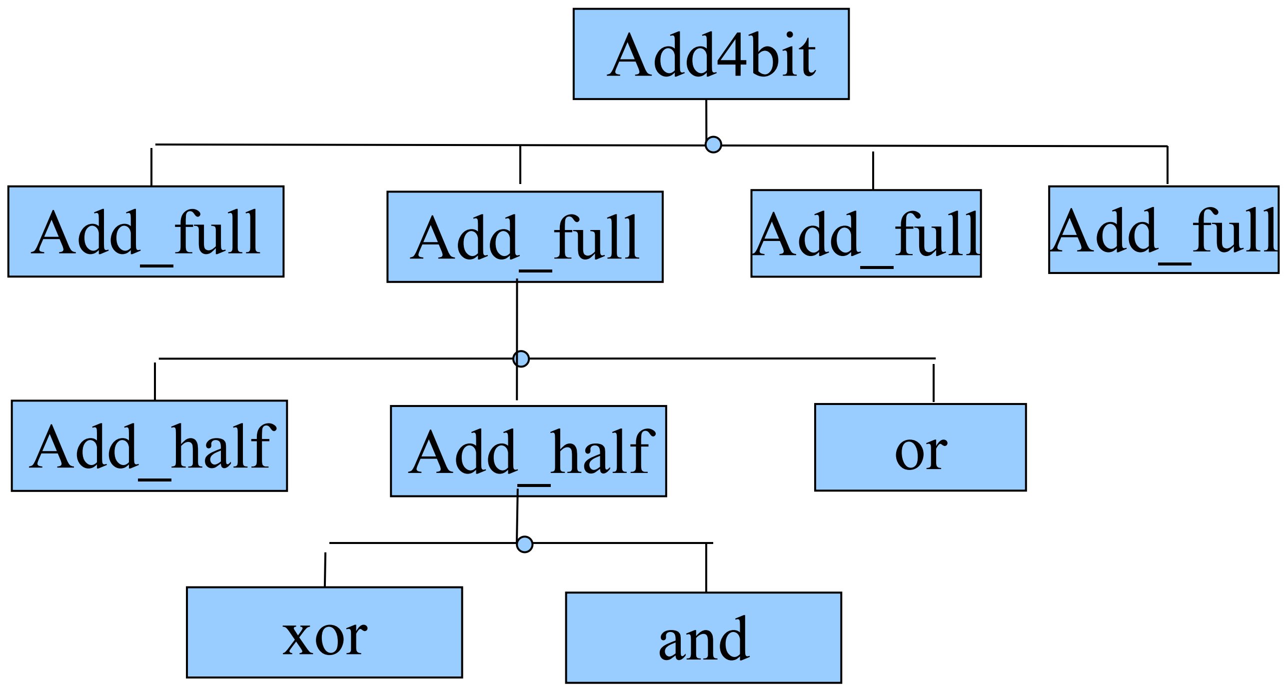
Example:
Detecting cout2

Must output
cout2

```
module Add_full (Co,S, A, B, Ci);
  input A, B, Ci ;
  output S, Co ;
  ....
endmodule
```



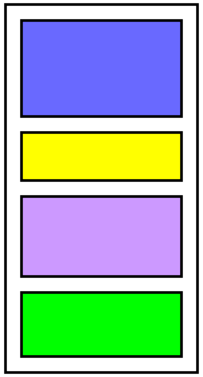
层次结构



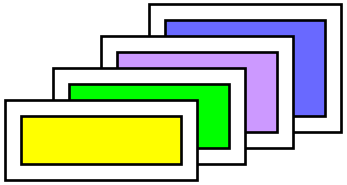
4位加法器的层次结构

层次化 与 源文件

- 多个module是放在同一个文件还是分开放？
- 可以放在同一个文件中
  - 模块次序自由
  - 适合于小设计
  - 不太适合模块重用（剪切和粘贴）



- 可以将模块分解成多个文件
  - 良好习惯：每个文件只包含一个module
  - 有助于组织模块
  - 便于找到一个模块
  - 有利于模块重用（将文件添加到项目中）



Dataflow Verilog

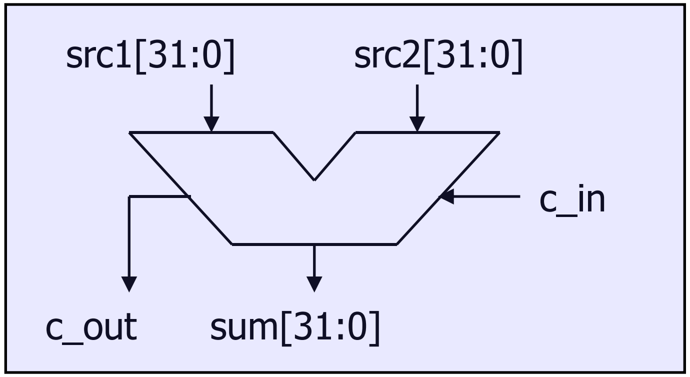
- The continuous assign statement
  - It is the main construct of Dataflow Verilog
  - It is **deceptively** powerful & useful
- More basic generic form:

  assign [dalay] <LHS_net> = <RHS_expression>;

- If RHS（right hand side） result changes, LHS is updated with new value
  - Constantly operating (“continuous”)
  - It’s ***hardware!***

Back to the Continuous Assign

- RHS can use operators (i.e. +,-,&,|,^,~,>>,…)




```
wire [31:0] sum, src1, src2;
wire  c_in,c_out;
assign {c_out,sum} = src1 + src2 + c_in;
```

Verilog运算符（9类）

- 算术运算符： +、-、*、/、%
- 位运算符： ~、&、|、^、^~ 或 ~^（异或非）
- 缩位运算符（单目）： &、~&(与非)、|、~|、^、^~、~^
- 逻辑运算符： !、&&、||
- 关系运算符（双目）： <、>、<=、>=
- 相等与全等运算符： ==（逻辑相等）、!=（逻辑不等）、===（全等）、!==（非全等）
- 逻辑移位运算符： <<、>>
- 连接运算符： { }
- 条件运算符： ?:

Behavioral Verilog

- 行为建模的主要机制：**initial** and **always**
  - 所有其他的行为语句包含在这两个语句中
  - **initial** and **always** 块不能嵌套
  - 左边变量必须是 **reg 类型**

- **initial** 语句从0时刻开始执行且只执行一次
  - 如果有多个**initial**块，它们都从0时刻开始并行独立执行，独立完成执行.

- 如果在**initial**语句中有多条行为语句，那么需用 **begin/end** 组合在一起。

More on **initial** statements

- Initial statement very useful for testbenches
- Initial statements **don't** synthesize
- Don't use them in DUT（DUT：device under test）Verilog (stuff you intend to synthesize)

initial Blocks

```
`timescale 1 ns / 100 fs
module full_adder_tb;
    reg [2:0] stim;
    wire s, c;

    add_full   DUT (c, s, 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 = 3’h0; stim < 3’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
```

Modelsim

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

- **always**语句重复执行
  - 从0时刻开始不断循环执行；
  - 可以使用触发/敏感事件列表来控制操作; **@(a or b or c)**
  - 没有触发事件，将在0时刻无限循环执行。

```
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;            // 带有时序控制
                                 //产生周期为20时间单位的波形
endmodule
```

always的事件控制方式

- always是基于事件执行，有两种类型的事件控制方式
  - 边沿触发事件控制:用来描述时序逻辑电路
  - 电平敏感事件控制:用来描述组合逻辑电路

- **always @(posedge** clk) // clk从低电平->高（正沿）

  cur_state =next_state; // 就执行赋值语句

- **always @(negedge** reset) // reset从高->低（负沿）

  count =0; // 就执行赋值语句

- **always @ (posedge** clear or **negedge** reset)

  Q=0;

- 不可以同时包括同一个信号的上升沿和下降沿

电平敏感信号列表

- 只要列表内有信号发生电平变化，就执行该always结构中的内容。

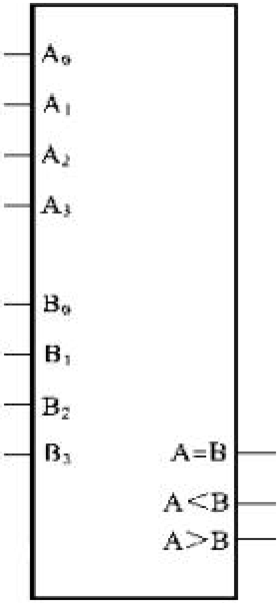
- Original way to specify trigger list
  always @ (X1 **or** X2 **or** X3)
- In Verilog 2001 can use , instead of **or**
  always @ (X1, X2, X3)
- Verilog 2001 also has * for *combinational only*
  always @ (*) //"*"代表所有输入信号，可防止遗漏

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

- 不可以同时包括电平敏感事件和边沿敏感事件

Example: 比较器

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

  always@( A or B) begin

    if ( A > B )
          { A_gt_B, A_lt_B, A_eq_B } = 3'b100;
     else if ( A == B)
          { A_gt_B, A_lt_B, A_eq_B } = 3'b001;
      else
          { A_gt_B, A_lt_B, A_eq_B } = 3'b010;

  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
```

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

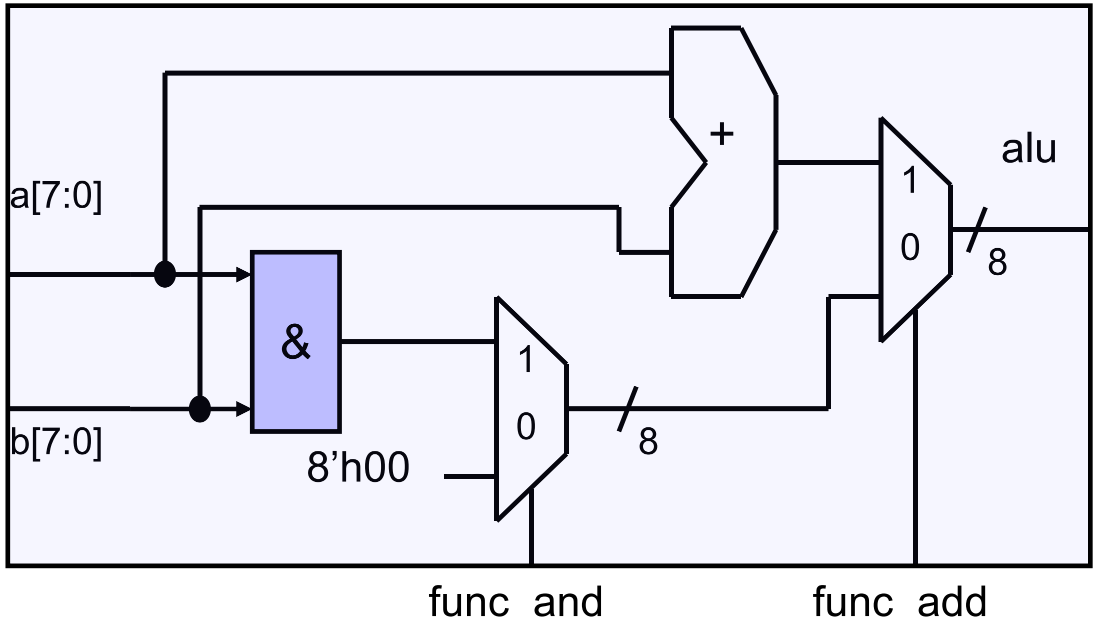
```
If (condition)
  begin
    <statement1>;
    <statement2>;
  end
else
  begin
    <statement3>;
    <statement4>;
  end
```

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

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

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

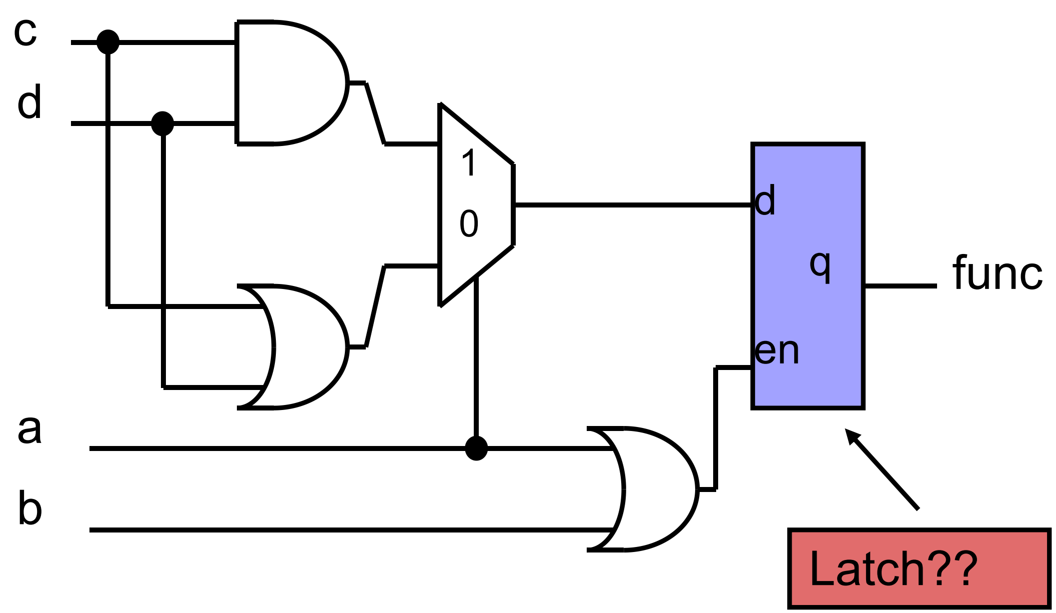
→



if statement synthesis (continued)

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

How does this synthesize?



if 要加上else，以防止锁存器的发生

语句不完整即有某些情况的输入对输出无任何影响，根据锁存器的特征，反映到硬件电路即会产生锁存器。

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
```

注意敏感信号的完备性：
必须将所有的输入信号和条件判断信号都列在信号列表中。

```
//带同步清0、同步置1的D触发器
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?

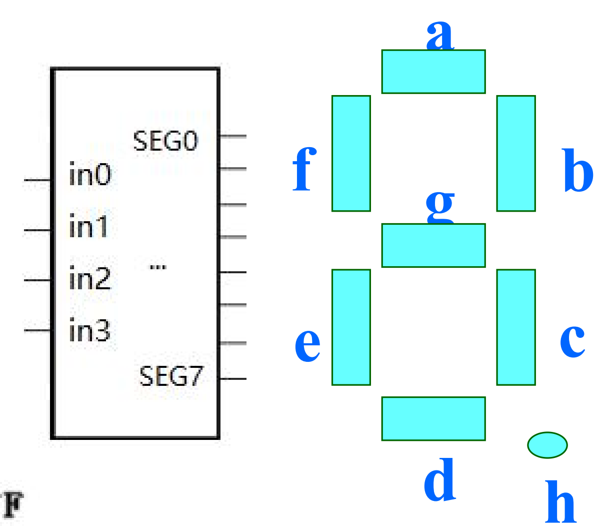
Does the reset or the set have higher priority?

Example: 7段数码管的显示译码器

```
module _7seg_display( in,SEG);
  input [3:0] in;      // 二进制输入
  output reg[7:0] SEG;   // 7-seg display 输出。0--ON，1--OFF
            // 对应关系，SEG[0]->a,SEG[1]->b, ...

  always@(   in    ) begin
     case (in[3:0])
         4'b0000 : SEG = 8'b11000000;     // 0
         4'b0001 : SEG = 8'b11111001;     // 1
         4'b0010 : SEG = 8'b10100100;     // 2
         4'b0011 : SEG = 8'b10110000;     // 3
         4'b0100 : SEG = 8'b10011001;     // 4
         4'b0101 : SEG = 8'b10010010;     // 5
         4'b0110 : SEG = 8'b10000010;     // 6
         4'b0111 : SEG = 8'b11111000;     // 7
         4'b1000 : SEG = 8'b10000000;     // 8
         4'b1001 : SEG = 8'b10010000;     // 9
         4'b1010 : SEG = 8'b10000110;     // E
         default : SEG = 8'b11111111;     // All OFF
     endcase
  end
endmodule
```

h g f e d c b a

显示0： 1 1 0 0 0 0 0 0

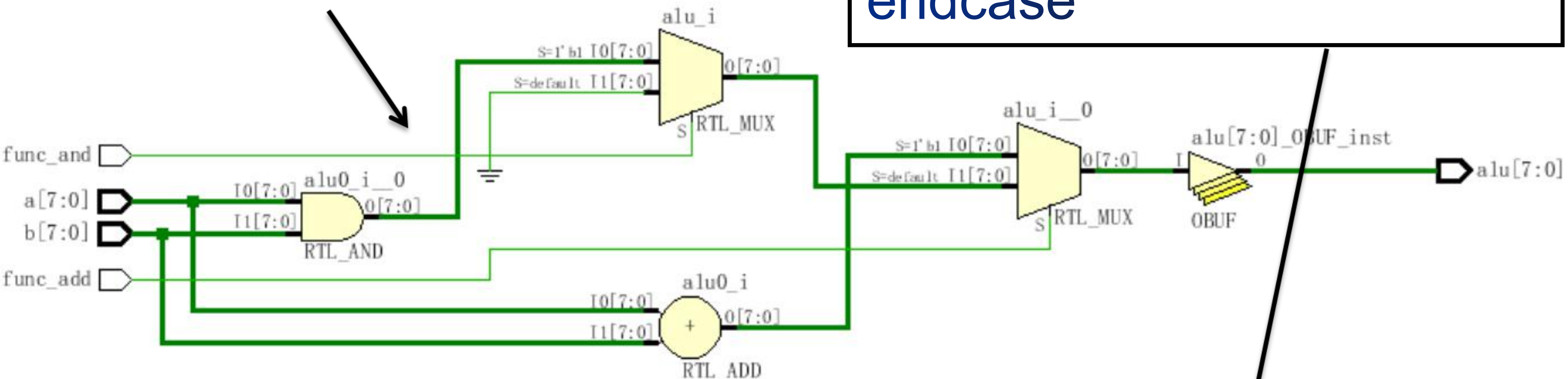


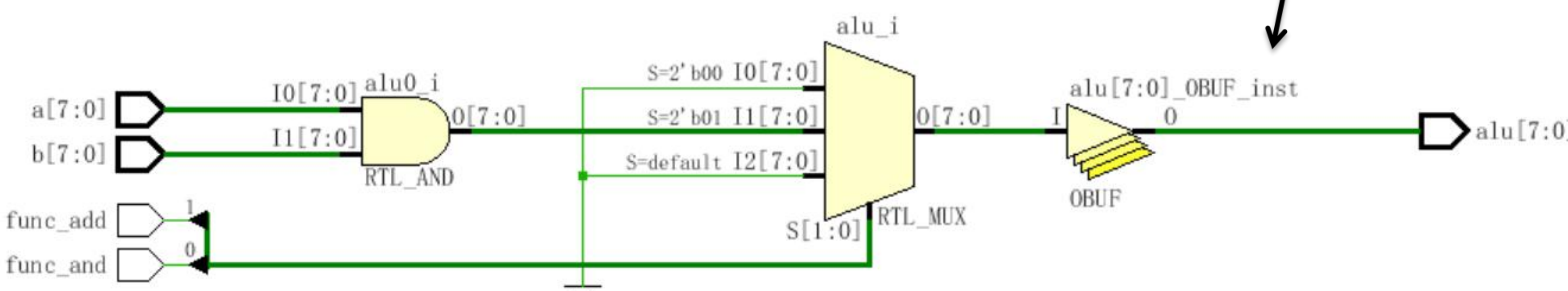
case Statements

- Verilog 有3种形式的cese语句:
  ⑩ **case**, **casex**, and **casez**
- 表达式按顺序与各分支项值按位匹配
  - 两者位宽相同!
- **case**：是一种全等比较
  - Can detect **x** and **z**！(good for testbenches)
- **casez**
  - 比较双方有一方的某些位的值是z，那么这些位的比较就不予考虑，而只关注其他位的比较结果
- **casex**
  - 比较双方有一方的某些位的值是x和z，就不予考虑
- **？可用来代替z，表示无关位**

```
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 = a + b;~~
    default: alu = 8'h00;
endcase





case的index列表里的x和z，都被综合工具认为是不可达状态,被去掉了。

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
  - ➓ If boolean expression false from beginning it will never execute loop

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

initial begin   // 找非0标志位
  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

- 执行指定的循环次数
  - ❿ Repeat的循环次数可以是一个变量，但
    - ✓其值在开始循环时得到计算，从而得以事先确定循环次数
    - ✓如果其值在循环执行期间发生更改，也不会更改迭代次数
- 在testbench，常与 @(posedge clk) 事件控制一起使用，用来等待固定数量的时钟。

```
initial begin
   inc_DAC = 1'b1;                      // 等待clk的4095个正沿
   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

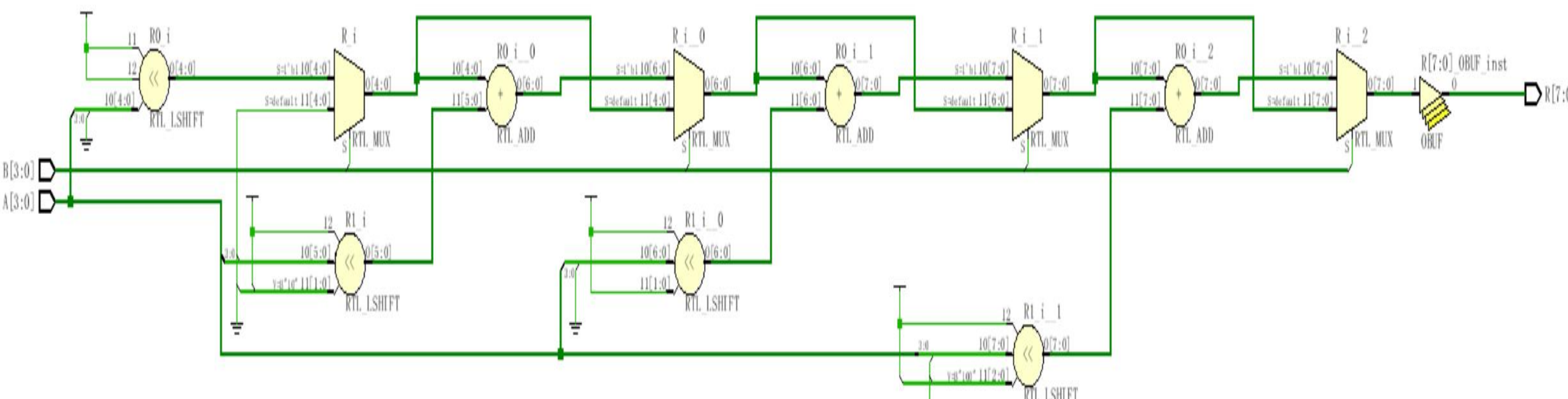
- 无限循环。常用于产生周期性的波形，用来作为仿真测试信号。它与always的不同之处是不能独立在程序中，必须写在initial块中
- Only a **$stop**, **$finish** or a specific **disable** can end a **forever** loop.

```
initial begin
  clk = 0;
  forever    //每隔10个时间单位clk反相一次
     #10 clk = ~ clk;
end
```

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

```verilog
// 4b乘法器，R=A*B
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=0; i<=3; i=i+1)
        if(B[i]) R = R + (A<<i);
      end

endmodule
```

```
A:        1101
B:        1011
* ----------------
          1101
         1101
       1101
+ ----------------
      10001111
```



HDL里的for是并行展开，代码看似简洁，电路规模扩大了n倍。

过程赋值语句

- 阻塞型过程赋值：=

  前一条语句没有完成赋值过程之前，后面的语句不能被执行

- 非阻塞型过程赋值：<=

  一条非阻塞赋值语句的执行，不会影响块中其它语句的执行

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>；//定时满足，**RHS**被计算和赋值

内部模式：<寄存器变量>= <定时控制> <表达式>；//计算**RHS**，定时满足后再赋值

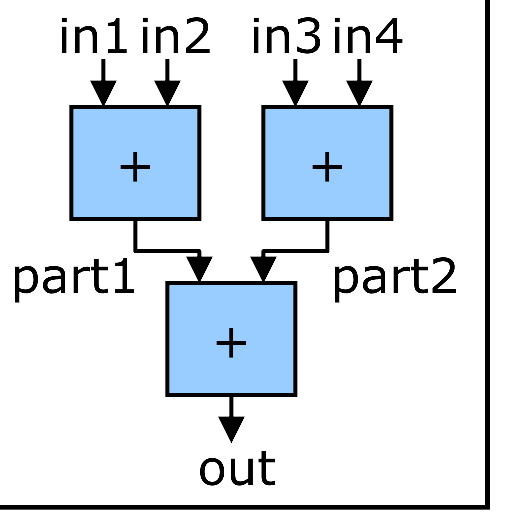
定时控制分为两类：延时控制 **#delay**

事件控制 @(事件控制敏感表）

阻塞性赋值

- 当前的赋值语句阻断了其后的语句，即后面的语句必须等到当前的赋值语句执行完毕才能执行
- Works a lot like software **(danger!)**
- 用于组合逻辑

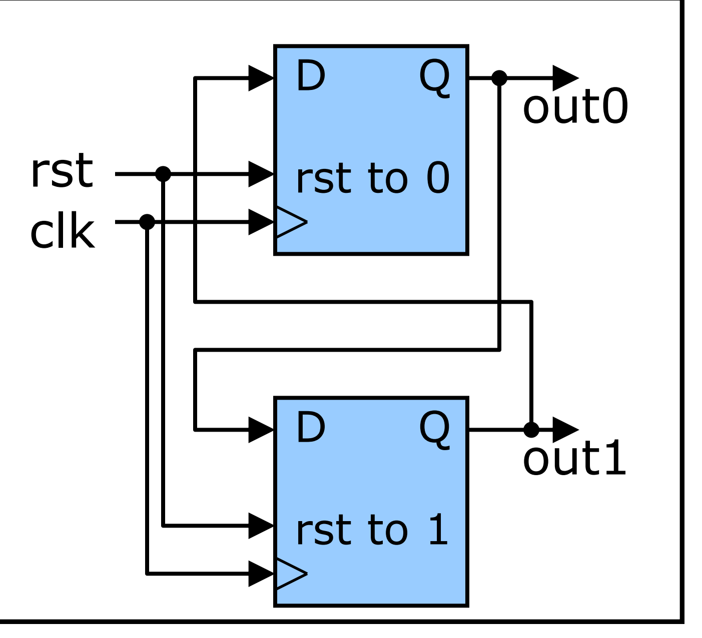
```
module addtree(output reg [9:0] out,
                          input [7:0] in1, in2, in3, in4);

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



非阻塞性赋值

- 如果没有定时控制，则同时计算RHS和更新LHS
- 用于时序逻辑
- 只能用在"initial"和"always"中

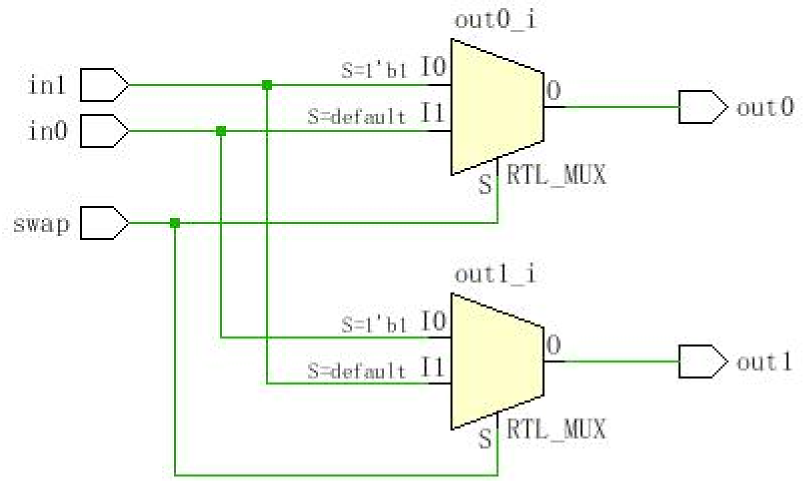
```
module swap(output reg out0, out1, input rst, clk);
   always @(posedge clk) begin
         if (rst) begin
               out0 <= 1'b0;
               out1 <= 1'b1;
         end
         else begin
               out0 <= out1;
               out1 <= out0;
         end
end
endmodule
```



Swapping if done in Blocking

- In blocking, need a "temp" variable

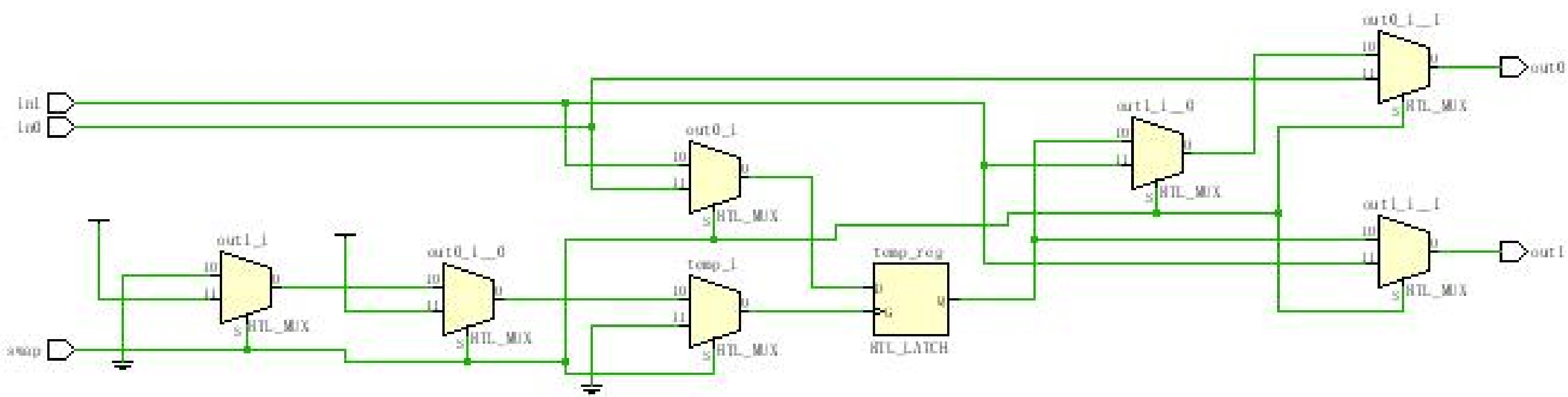
```
module swap(output reg out0, out1, input in0, in1, swap);

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 *?

Swapping if done in Non-Blocking

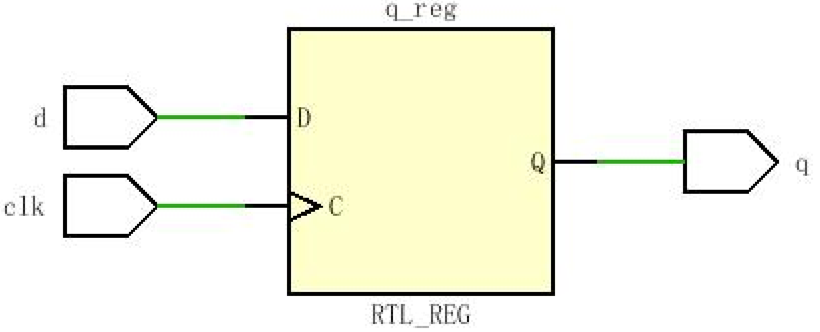
```
module swap(output reg out0, out1, input in0, in1, swap);
    reg temp;
    always @(*) begin
            out0 <= in0;
            out1 <= in1;
            if (swap) begin
                        temp <= out0;
                        out0 <= out1;
                        out1 <= temp;
          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

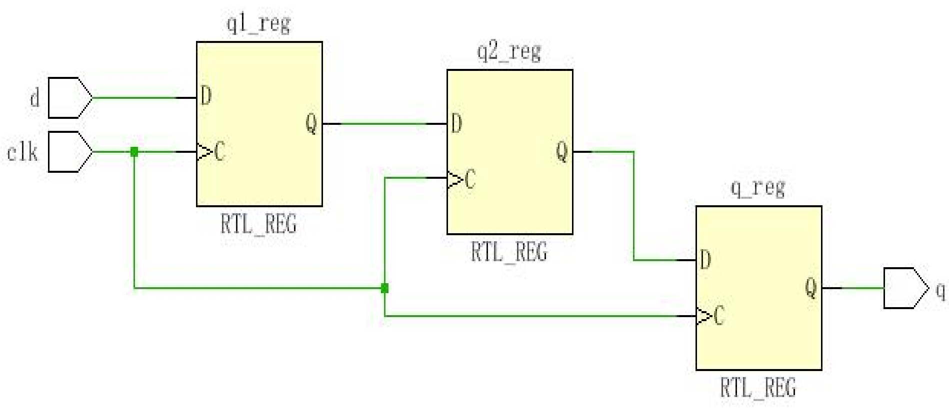
```
module pipe(clk, d, q);

input clk,d;
output q;
reg q,q1,q2;

always @(posedge clk) begin
  q1 = d;
  q2 = q1;
  q  = q2;
end  //在一个时钟沿，d传递到q
endmodule
```



More on Non-Blocking

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

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

input clk,d;
output q;
reg q,q1,q2;

always @(posedge clk) begin
  q1 <= d;
  q2 <= q1;
  q <= q2;
end //d传递到q需要间隔三个时钟

endmodule;
```



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.

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
```

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?

任务(task)和 函数(function)

```
module  Name（port list）;

  ……

task 任务名;
    参数与类型说明;
    局部变量说明; //静态的
    过程语句
endtask

function <位宽说明> 函数名;
    输入参数与类型说明;
    局部变量说明; //静态的
    过程语句
endfunction

endmodule
```

可以在模块不同位置执行共同代码

任务(task)和 函数(function)

- 差异
  - 任务可以含有时序控制，而函数则没有；
  - 任务可以有输入和输出参数，而函数至少一个输入参数，没有输出参数；
  - 任务的调用是通过调用语句，而函数调用出现在表达式中。

functions

- 函数的定义和调用都包括在一个module内
  - 函数调用在表达式中 (RHS)
- 用于实现组合逻辑
  - 不能包含任何时延、时序、事件控制
  - 可以调用其他函数，不能调用任务
  - 只使用阻塞赋值，行为语句
- 输入/输出
  - 至少一个输入参数，不能有输出或者双向(inout)参数
  - 返回一个值，**与函数同名的寄存器变量在函数中被隐式地声明**，通过对该寄存器赋值来返回函数值，可以指定返回值的宽度 (缺省1-bit)

Function Example

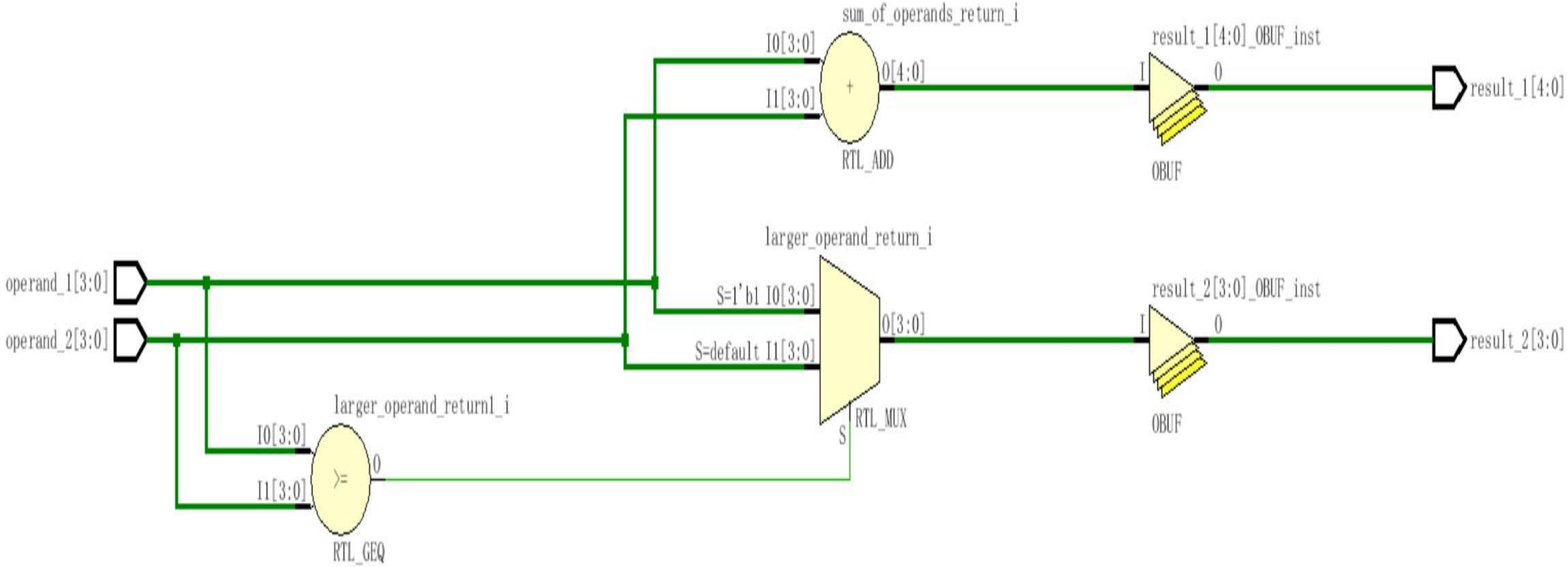
```
module arithmetic_unit (result_1, result_2, operand_1, operand_2,);
  output          [4: 0] result_1;
  output          [3: 0] result_2;
  input           [3: 0] operand_1, operand_2;
  assign result_1 = sum_of_operands (operand_1, operand_2);
  assign result_2 = larger_operand (operand_1, operand_2);

  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 output

function inputs

Function Example



tasks (much more useful than functions)

Functions	Tasks
函数只能调用其他函数，不能调用任务	任务可以调用其他任务和函数
函数不能包含任何延迟，事件或者时序控制	任务可以包含延迟，事件或者时序控制 (i.e. ➔ @, #)
函数至少要有一个输入参数，也可以有多个输入参数	任务可以没有或者有多个输入，输出，双向参数
函数只能返回一个值，不能有输出或者双向参数	任务不返回任何值，但是返回多个输出或双向参数值

Why use Tasks?

- 把大程序分成小程序
- 在测试台上使用任务非常方便
  - 将常见的测试例程分解为任务
    - ✓初始化任务
    - ✓激励信号产生任务
    - ✓自检任务
  - 顶层测试主要是对任务的调用

Task Example [Part 1]

```
module adder_task (c_out, sum,
  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); //任务调用
  end
  // Continued on next slide
```

任务调用:

*（1）任务调用语句是过程性语句，只能出现在**always** 语句或**initial** 语句中；*

（2）任务调用语句中的参数列表顺序和类型必须和任务定义中的一致；

（3）输出参数必须是寄存器类型。

Task Examp

任务定义:

（1）任务的参数不受限制

（2）在任务中可以调用其他的任务或函数。

（3）任务定义内不能出现 ***initial*** *和* ***always*** *过程块。*

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

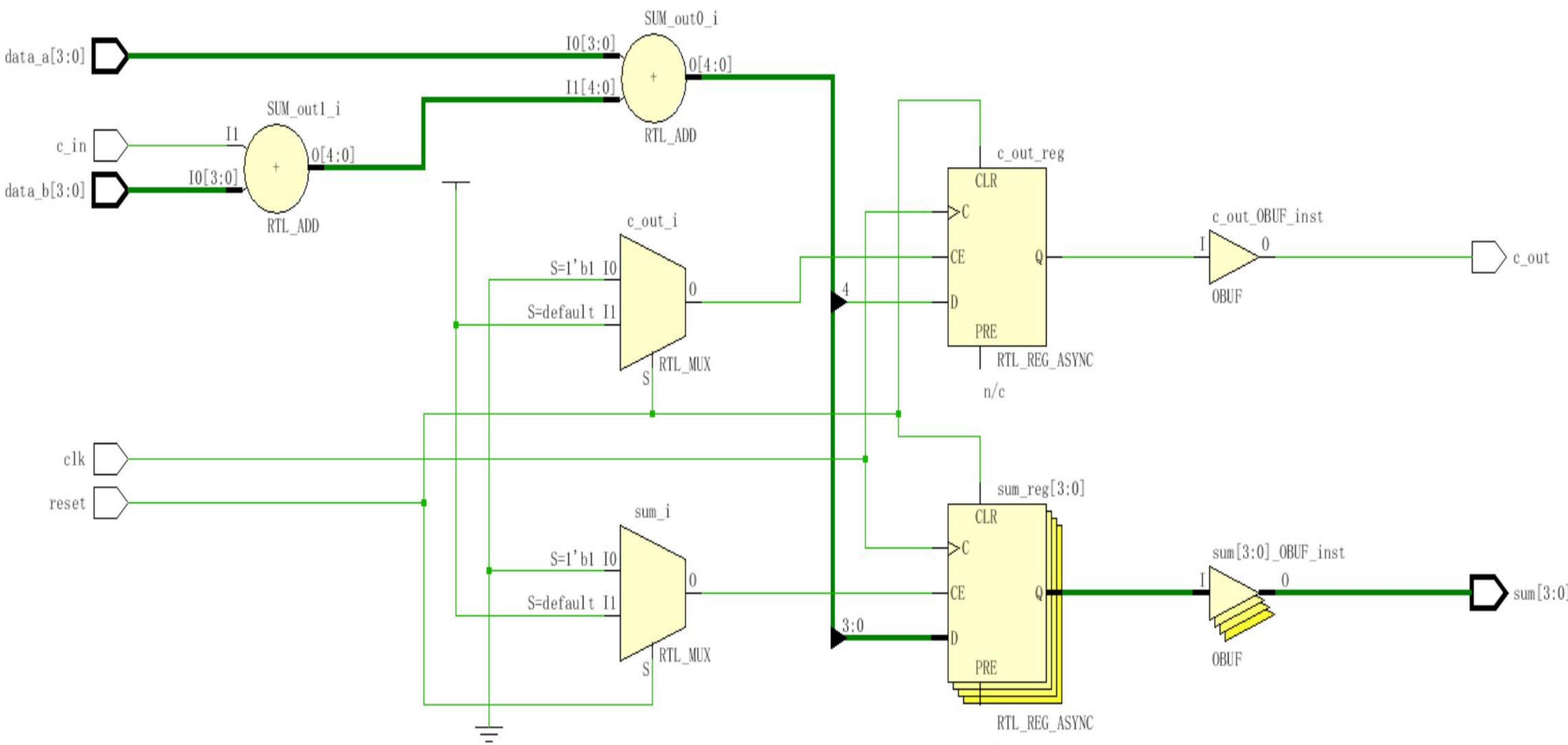
task outputs

task inputs

- 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);
```

Task Example [Part 3]



时序控制

- "#"符号引入的延迟控制，时间的长度由<time>值确定。

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

- "@"符号引入的事件控制

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

- 等待语句。表达式为真之前，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
```

(a=>out)=9;

//从输入a到out的每位延迟为9

命名程序块

- Blocks (begin/end) or (fork/join) can be named
  - 命名块中可以声明局部变量;
  - 命名块中声明的变量可以通过层次名引用进行访问;
  - 命名块可以被禁用（例如停止其执行）

block name

```
module top();

initial begin: block1          //名字为block1的过程命名块
  integer i;                   // i 是block1命名块的静态局部变量
                               //可以用层次名top.block1.i 被其他模块访问
  …
end                            // end of block1

endmodule
```

disable Statement

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

block name

block name

```
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

- 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
```

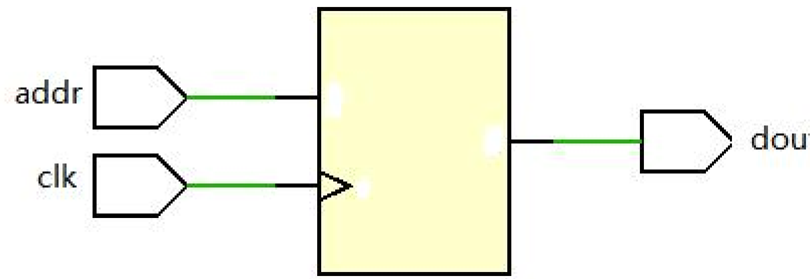
example "hex" file

```
//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