Verilog HDL使用中该注意的问题及一些模块代码

一、采用模块化、层次化设计

1、假设有三个模块top（顶层模块），bottom1（底层模块），bottom2（底层模块）分别存在于三个文件top.v, bottom1.v, bottom2.v中。

2、在Verilog HDL中，调用底层模块的语法结构为：

底层模块名 实例名(参数定义表)

或

底层模块名 你给它起的名(.模块端口名(连线名),……);

其中，模块端口名就是模块内参数名；连线名是顶层模块中定义的信号名

3、例子如下：

如果底层模块文件（*.v）没有加到和顶层模块文件（*.v）同一个项目中，就必须在顶层模块文件（*.v）的开头部分，使用如下方法且指定路径和文件名，如：
’include "bottom1.v"。在bottom1.v文件前面指定路径。

顶层模块文件（top.v）中与底层模块的关系：

```
module  top(clk, reset, rec, send, data); // 顶层模块
    input  clk, reset, rec;
    output  send;
    output [15:0]  data;

    bottom1  my_bottom1(         // 底层模块1
    .rec(rec), .clk(clk), .reset(reset), .data(data)
    );
    bottom2  my_bottom2(         // 底层模块2
    .data(data), .clk(clk), .reset(reset), .send(send)
    );
endmodule
```

二、wire 与reg类型的使用

1、wire类型变量用于assign连续赋值语句中被赋值。

例如，wire A;

assign A = B; 其中，B变量可为wire或reg类型。

2、reg类型变量用于always块内或initial 语句（initial 语句只执行一次，仿真开始时执行，即在0时开始执行，不能综合。一般只用于初始化一些常量。）中被赋值。

例如，

```
reg A,B,C;
initial begin
    A = 0;              // A、B 必须为reg类型
    B = 0;
end
always@(*)begin
    if(A) C = 1 else C = 0;  // C 必须为reg类型
end
```

3、不能这样使用，assign 语句不能出现在always块中

```
reg A,B,C
always@(*)begin
    assign A = B;                // 用法错误，综合（Synthesize）通不过！
    if(A) C = 1 else C = 0;  // C 必须为reg类型
end
```

三、always块中敏感信号的使用注意

1、时序信号触发（时钟边沿触发）（PC模块例子）

```
module PC(input clk,reset,input [31:0]nextPC,output [31:0] curPC);
      always@( posedge clk or negedge reset ) begin  // posedge（上升沿）、negedge（下降沿）
         if( reset == 1'b0 )   curPC <= 0;        // PC初值
         else if (PCWre)       curPC <= nextPC;  // 下条指令的地址
      end
endmodule
```

2、电平信号触发 （存储器模块例子）

```
module RAM(input nRD,nWR,input [15:0] DAddr,input [7:0] Datain,output [7:0] outData);
      reg [7:0] dataMem[0:99];  // 存储单元8位长度，共100地址单元
      assign outData = (nRD==0)?dataMem:8`bzzzzzzzz;   // 读
      always@(nWR or DAddr or Datain)begin // nWR、DAddr、Datain为电平信号
         if(nWR == 0) dataMem[DAddr] = Datain;  / 写
      end
endmodule
```

3、不能混用。敏感信号表中若有时序信号触发时，不能还有电平信号触发。以下用法错误！

```
      always@( posedge clk or PCWre ) begin      // 综合（Synthesize）通不过！
         if( reset == 1'b0 )   curPC <= 0;        // PC初值
         else if (PCWre)       curPC <= nextPC;  // 下条指令的地址
      end
```

四、阻塞赋值（=）与非阻塞赋值（<=）

1、阻塞赋值（=），用于组合逻辑建模。

```
always@(ALUop or A or B)begin    // ALUop、A、B为电平信号
   case(ALUop)
         2'b00:result = A + B;
         2'b01:result = A - B;
         2'b10:result = B | A;
         2'b11:result = A & B;
         default:result = 32'b0;
      endcase
      zero = (result == 0) ? 1 :0;
end
```

2、非阻塞赋值（<=）,用于时序逻辑建模。

```
always@( posedge clk or negedge reset ) begin // posedge（上升沿）、negedge（下降沿）
   if( reset == 1'b0 )  curPC <= 0;        // PC初值
   else if (PCWre)  curPC <= nextPC;       // 下条指令的地址
end
```

3、阻塞赋值（=）与非阻塞赋值（<=），在同一个always中，不要混合使用。以下用法错误！

```
module ALU(input [1:0] ALUop,input [31:0] A, input [31:0] B,input zero, output reg [31:0] result);
      always@(ALUop or A or B)begin
         case(ALUop)
               2'b00:result <= A + B;
               2'b01:result <= A - B;
               2'b10:result = B | A;
               2'b11:result = A & B;
               default:result = 32'b0;
           endcase
           zero = (result == 0) ? 1 :0;
      end
endmodule
```

五、同一个变量不能在不同always块中赋值

1、同一模块中，在一个以上的always块中对同一个变量赋值容易产生竞争冒险，且不能综合（Synthesize）。

2、例子，以下例子中，a 在两个always块中被赋值，**错！**

```
reg a,b,c;
always@(*)begin
    a = b;
    c = b;
end
always@(*)begin
    a = c;
end
```

六、ROM的实现方法（代码）

```
module ROM ( rd, addr, dataOut);  // 存储器模块
    input  rd;     // 读使能信号
    input [ 31:0] addr;   // 存储器地址
   output  reg [31:0] dataOut;  // 输出的数据

   reg [7:0]  rom [99:0]; // 存储器定义必须用reg类型，存储器存储单元8位长度，共100个存储单元
   Initial begin    // 加载数据到存储器rom。注意：必须使用绝对路径，如：E:/Xlinx/VivadoProject/ROM/（自己定）
       $readmemb ("E:/Xlinx/VivadoProject/ROM/rom_data.txt", rom);  // 数据文件rom_data（.coe或.txt）。未指定，就从0地址开始存放。
   end
   always @( rd or addr ) begin
     if (rd==0)          // 为0，读存储器。大端数据存储模式
         dataOut[31:24] = rom[addr];
         dataOut[23:16] = rom[addr+1];
         dataOut[15:8] = rom[addr+2];
         dataOut[7:0] = rom[addr+3];
     end
endmodule
```

rom_data.txt文件内容表示方法，如下页所示：

rom_data.txt（.coe）数据文件内容为：（供参考）

00001000 00000001 00000000 00001000
01001000 00000010 00000000 00000010
00000000 01000001 00011000 00000000
00000100 01100001 00100000 00000000
01000100 10000010 00101000 00000000
01000000 10100001 00110000 00000000
11101000 00000000 00000000 00001110
10011001 01100010 00111000 00000000
10011000 11100001 01000000 00000000
10011101 00001001 00000000 00000101
10001101 00001011 00001000 10000101

以上为二进制数据，8位为一组，以空格隔开。

七、RAM的实现方法（代码，用时钟信号触发写）

```
module RAM(
    input clk,
    input [31:0] address,
    input [31:0] writeData, //  [31:24], [23:16], [15:8], [7:0]
    input nRD,                  // 为0，正常读；为1,输出高组态
    input nWR,                  // 为0，写；为1，无操作
   output [31:0] Dataout
    );
    reg [7:0] ram [0:60];    // 存储器定义必须用reg类型

   // 读
   assign Dataout[7:0]   = (nRD==0)?ram[address + 3]:8'bz; // z 为高阻态
   assign Dataout[15:8]  = (nRD==0)?ram[address + 2]:8'bz;
   assign Dataout[23:16] = (nRD==0)?ram[address + 1]:8'bz;
   assign Dataout[31:24] = (nRD==0)?ram[address ]:8'bz;
```

```
    // 写
      always@( negedge clk ) begin    // 用时钟下降沿触发写存储器，个例
          if( nWR==0 ) begin
              ram[address]  <= writeData[31:24];
              ram[address+1] <= writeData[23:16];
              ram[address+2] <= writeData[15:8];
              ram[address+3] <= writeData[7:0];
          end
      end
endmodule
```

温馨提示：address不能乘以4，即address*4，是错误的！

八、RAM的实现方法（代码，用电平信号触发写）

```
module RAM(
    input [31:0] address,
    input [31:0] writeData, //  [31:24], [23:16], [15:8], [7:0]
    input nRD,                  // 为0，正常读；为1,输出高组态
    input nWR,                  // 为0，写；为1，无操作
   output [31:0] Dataout
    );
    reg [7:0] ram [0:60];    // 存储器定义必须用reg类型

   // 读
   assign Dataout[7:0]   = (nRD==0)?ram[address + 3]:8'bz; // z 为高阻态
   assign Dataout[15:8]  = (nRD==0)?ram[address + 2]:8'bz;
   assign Dataout[23:16] = (nRD==0)?ram[address + 1]:8'bz;
   assign Dataout[31:24] = (nRD==0)?ram[address ]:8'bz;
```

```
    // 写
      always@( nWR or address or writeData ) begin    // 用电平信号触发写存储器，个例
          if( nWR==0 ) begin
              ram[address]  = writeData[31:24];
              ram[address+1] = writeData[23:16];
              ram[address+2] = writeData[15:8];
              ram[address+3] = writeData[7:0];
          end
      end
endmodule
```

温馨提示：address不能乘以4，即address*4，是错误的！

九、寄存器组的实现方法（代码）

```
module RegFile(CLK,RST,RegWre,ReadReg1,ReadReg2,WriteReg,WriteData,
                    ReadData1,ReadData2);
    input CLK;
    input RST;
    input RegWre;
    input [4:0] ReadReg1,ReadReg2,WriteReg;
    input [31:0] WriteData;
    output [31:0] ReadData1,ReadData2;

    reg [31:0] regFile[1:31]; //  寄存器定义必须用reg类型
    integer i;
```

```
    assign  ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1];   // 读寄存器数据
    assign  ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];

    always @ (negedge CLK or negedge RST) begin   // 必须用时钟边沿触发
        if (RST==0) begin
            for(i=1;i<32;i=i+1)
            regFile[i] <= 0;
       end
       else if(RegWre == 1 && WriteReg != 0)     // WriteReg != 0，$0寄存器不能修改
           regFile[WriteReg] <= WriteData;         // 写寄存器
    end

endmodule
```

十、ALU的实现方法（代码）

```
module ALU(
   input [2:0] ALUopcode,
   input [31:0] rega,
   input [31:0] regb,
   output zero，
   output reg [31:0] result
   );

   assign zero = (result==0)?1:0;
   always @( ALUopcode or rega or regb ) begin
      case (ALUopcode)
        3'b000 : result = rega + regb;
        3'b001 : result = rega - regb;
        3'b010 : result = rega & regb;
        3'b011 : result = rega | regb;
```

```
        3'b100 : result = (rega < regb)?1:0;   // 无符号比较
        3'b101 : begin                                  // 带符号比较
                  if (rega<regb &&( rega[31] ==  regb[31]) )
                       result = 1;
                  else if ( rega[31] == 1 && regb[31]==0)  result = 1;
                  else result = 0;
                 end
        default : begin
                  result = 32'h00000000;
                  $display (" no match");
                  end
       endcase
    end
endmodule
```

十一、锁存器的实现方法（代码）

锁存器建模时，用非阻塞赋值（<=）。

```
module lacth ( d,z,sel);
    Input d,sel;
    output z;

   // 如果在给定的条件下变量没有赋值，这个变量将保持原值，会生成一个锁存器
    always @(sel or d) begin
        if(sel == 1'b1)
            z <= d;            // 非阻塞赋值。
    end
endmodule
```

十二、延迟的实现方法（代码）

#n，使用这种方法延迟，只能在仿真中有效，在硬件中无法实现。硬件中的延迟，往往使用寄存器缓冲实现。

1、寄存器缓冲，延迟法，用于硬件中延迟

```
module DR(input clk,input [31:0] in,output reg [31:0] out);
    always@(posedge clk)begin   // 当然，也可以下降沿 negedge
        out <= in;
    end
endmodule
```

2、#n 延迟法，只能在仿真时使用

```
    reg a,b,c;
    always@(*)begin
        #15  a = b;              // 延迟15ns后，执行语句 a = b
             c = #15  b;         // 延迟15ns后，将 b赋给 c
    end
```

十三、关于系统任务$readmemb和$readmemh的使用

verilog HDL程序中的两个系统任务，$readmemb和$readmemh，从文件中读取数据到存储器。其格式如下：

数据文件中数据为二进制数据：

（1）$readmemb("<数据文件名>",<存储器名>);

（2）$readmemb("<数据文件名>",<存储器名>,<起始地址>);

（3）$readmemb("<数据文件名>",<存储器名>,<起始地址>,<终止地址>);

数据文件中数据为十六进制数据：

（1）$readmemh("<数据文件名>",<存储器名>);

（2）$readmemh("<数据文件名>",<存储器名>,<起始地址>);

（3）$readmemh("<数据文件名>",<存储器名>,<起始地址>,<终止地址>);

例子：数据文件rom_data.txt（.coe），假设问存放的路径：E:/Xlinx/VivadoProject/ROM/，则（注意反斜线“/”）

$readmemb("E:/Xlinx/VivadoProject/ROM/rom_data.txt",rom)，从0号地址单元开始存放，指定绝对路径（自己定）

$readmemb("E:/Xlinx/VivadoProject/ROM/rom_data.txt",rom,12)，从12号地址单元开始存放