多路选择器

一.实验内容

1.做一个 4 选 1 的多路选择器，并进行波形仿真。

2.将 4 选 1 多路选择器同 2 选 1 多路选择器对比，观察资源消耗的变化。

二.实验步骤

3.编写 Verilog HDL 代码

具体代码如下：

```
// module  top，选择器（mux）的代码，
module top(
   IN0          ,   // input 1
   IN1          ,   // input 2
   IN2          ,   // input 3
   IN3          ,   // input 4
   SEL          ,   // select
   OUT          );  // out data
input [15:0] IN0, IN1, IN2, IN3;//  选择器的输入数据信号
input [1:0] SEL;                    //  通道选通的控制信号
output[15:0] OUT;       //  选择器的输入数据信号
reg    [15:0] OUT;
//  生成组合逻辑的代码
always @ (IN0 or IN1 or IN2 or IN3 or SEL) begin
   if(SEL==0) // SEL 为 0  选择输入 0
      OUT = IN0;
   else   if(SEL==1)   // SEL 为 1  选择输入 1
      OUT = IN1;
   else   if(SEL==2)   // SEL 为 2  选择输入 2
      OUT = IN2;
   else   if(SEL==3)   // SEL 为 3  选择输入 3
      OUT = IN3;
end
endmodule
// endmodule top
```

查看多路选择器的 RTL 结构

8.两种多路选择器的资源消耗对比

2 选 1 多路选择器：

4 选 1 多路选择器：

可见 4 选 1 多路选择器相比 2 选 1 多路选择器消耗更大的硬件资源。

交叉开关

一.实验内容

1.编写一个 4X4 路交叉开关的 Verilog 代码，然后编译，进行波形仿真。

2.观察 RTL View，比较 2x2 路交叉开关与 4x4 路交叉开关之间消耗资源的区别。

二.实验步骤

1.编写 Verilog HDL 代码

```
// module    top, a 4x4 crossbar switch circuit
module top(
   IN0          ,     // input 1
   IN1          ,     // input 2
   IN2          ,     // input 3
   IN3          ,     // input 4
   SEL0         ,     // select the output0 source
   SEL1         ,     // select the output1 source
   SEL2         ,     // select the output2 source
   SEL3         ,     // select the output3 source
   OUT0          ,     // output data 0
   OUT1          ,     // output data 1
   OUT2          ,     // output data 2
   OUT3          );   // output data 3
input [15:0] IN0, IN1, IN2, IN3;
input [1:0] SEL0, SEL1, SEL2, SEL3;
output[15:0] OUT0, OUT1, OUT2, OUT3;
reg     [15:0] OUT0, OUT1, OUT2, OUT3;
// get the OUT0
always @ (IN0 or IN1 or IN2 or IN3 or SEL0) begin
   if(SEL0==0)
      OUT0 = IN0;
   else if(SEL0==1)
      OUT0 = IN1;
   else if(SEL0==2)
      OUT0 = IN2;
   else if(SEL0==3)
      OUT0 = IN3;
end
// get the OUT1
always @ (IN0 or IN1 or IN2 or IN3 or SEL1) begin
   if(SEL1==0)
      OUT1 = IN0;
   else if(SEL1==1)
      OUT1 = IN1;
   else if(SEL1==2)
      OUT1 = IN2;
   else if(SEL1==3)
```

```
        OUT1 = IN3;
end
// get the OUT2
always @ (IN0 or IN1 or IN2 or IN3 or SEL2) begin
    if(SEL2==0)
        OUT2 = IN0;
    else if(SEL2==1)
        OUT2 = IN1;
    else if(SEL2==2)
        OUT2 = IN2;
    else if(SEL2==3)
        OUT2 = IN3;
end
// get the OUT3
always @ (IN0 or IN1 or IN2 or IN3 or SEL3) begin
    if(SEL3==0)
        OUT3 = IN0;
    else if(SEL3==1)
        OUT3 = IN1;
    else if(SEL3==2)
        OUT3 = IN2;
    else if(SEL3==3)
        OUT3 = IN3;
end
endmodule
// endmodule top
```

4.查看 2×2 交叉开关的 RTL 结构

5.两种交叉开关的资源消耗对比

2×2 交叉开关的资源消耗：

4×4 交叉开关的资源消耗：

可见，4×4 交叉开关的逻辑单元消耗是 2×2 交叉开关的 4 倍。

优先编码器

一.实验内容

1.编写一个 8 输入的优先编码器，然后编译，查看 RTL View。

二.实验步骤

1.编写 Verilog HDL 代码

```
// module top, 8 input priority encoder with zero input check
module top(
   IN              ,     // input
   OUT             );    // output
input [7:0] IN;
output[3:0] OUT;
reg     [3:0] OUT;
// get the OUT
always @ (IN) begin
    if(IN[7])            //  第一优先级
       OUT = 4'b111;
    else if(IN[6])    //  第二优先级
       OUT = 4'b110;
    else if(IN[5])    //  第三优先级
       OUT = 4'b101;
    else if(IN[4])    //  第四优先级
       OUT = 4'b100;
    else if(IN[3])    //  第五优先级
       OUT = 4'b011;
    else if(IN[2])    //  第六优先级
       OUT = 4'b010;
    else if(IN[1])    //  第七优先级
       OUT = 4'b001;
    else if(IN[0])    //  第八优先级
       OUT = 4'b000;
    else                    //  什么都没有检测到
       OUT = 4'b1111; //  输出值可自定义，不和上面的输出值混淆即可
end
endmodule
```

3.查看 RTL View

多路译码器

一.实验内容

1.编写一个 4－16 的译码器，编译，仿真。

2.查看 RTL View，并和 3－8 译码器对比资源开销。

二.实验步骤

1.编写 Verilog HDL 代码

```
// module top, 4-16 decoder
module top(
   IN              ,     // input
   OUT             );   // output
input [3:0] IN;
output[15:0] OUT;
reg     [15:0] OUT;
// get the OUT
always @ (IN) begin
   case(IN)
      4'b0000: OUT = 16'b0000_0000_0000_0001;
      4'b0001: OUT = 16'b0000_0000_0000_0010;
      4'b0010: OUT = 16'b0000_0000_0000_0100;
      4'b0011: OUT = 16'b0000_0000_0000_1000;
      4'b0100: OUT = 16'b0000_0000_0001_0000;
      4'b0101: OUT = 16'b0000_0000_0010_0000;
      4'b0110: OUT = 16'b0000_0000_0100_0000;
      4'b0111: OUT = 16'b0000_0000_1000_0000;
      4'b1000: OUT = 16'b0000_0001_0000_0000;
      4'b1001: OUT = 16'b0000_0010_0000_0000;
      4'b1010: OUT = 16'b0000_0100_0000_0000;
      4'b1011: OUT = 16'b0000_1000_0000_0000;
      4'b1100: OUT = 16'b0001_0000_0000_0000;
      4'b1101: OUT = 16'b0010_0000_0000_0000;
      4'b1110: OUT = 16'b0100_0000_0000_0000;
      4'b1111: OUT = 16'b1000_0000_0000_0000;
      //   full case  不需要写 default，否则一定要有 default
   endcase
end
endmodule
```

3.查看 RTL View

   4-16 译码器：

   3-8 译码器：

4.对比资源开销

   3-8 译码器：

   4-16 译码器：

   4-16 译码器的资源消耗约为 3-8 译码器的 2 倍。

加法器

无符号加法器

一.实验内容

1.把加法器的输入信号和输出信号都改成 4 比特位宽，编译，波形仿真。观察输出结果，说出输出和输入的对应关系。

2.把加法器的输入信号改成 8 比特位宽，编译，波形仿真。观察加法器的输出延迟，和 4 比特输入位宽的情况对比。

二.实验步骤

1.编写 Verilog HDL 代码

```
module top(
   IN1     ,
   IN2     ,
   OUT     );
input[3:0] IN1, IN2;
output[3:0] OUT;
reg[3:0] OUT;
always@(IN1 or IN2) begin //  生成组合逻辑的 always  块
   OUT = IN1 + IN2;
end
endmodule
```

3.输入信号为 8 位宽的加法器的 Verilog HDL 代码

```
module top(
   IN1     ,
   IN2     ,
   OUT     );
input[7:0] IN1, IN2;
output[8:0] OUT;
reg[8:0] OUT;
always@(IN1 or IN2) begin //  生成组合逻辑的 always  块
   OUT = IN1 + IN2;
end
endmodule
```

4.8 位宽加法器仿真波形

从以上两张波形图可以看出 4 位宽加法器和 8 位宽加法器的输出延迟差别不大，基本都在 8ns 左右。

补码加法器

一.实验内容

1.把加法器的输出信号改成 4 比特位宽，编译，波形仿真。观察输出结果，观察输出结果在什么时候是正确的?

2. 把加法器的输入信号改成 8 比特位宽，编译，波形仿真。观察加法器的输出延迟，和 4 比特输入位宽的情况对比。

二.实验步骤

1.编写 4 比特位宽输出加法器的 Verilog HDL 代码

```
module top(
   IN1     ,
   IN2     ,
   OUT     );
input signed [3:0] IN1, IN2;
output signed [3:0] OUT;
reg signed [3:0] OUT;
always@(IN1 or IN2) begin //  生成组合逻辑的 always  块
   OUT = IN1 + IN2;
end
endmodule
```

2.仿真结果波形

从仿真结果可知，当两个输入信号的和大于 7 或小于-8 时计算结果会出错，这是由于次高位进位使得符号位变化导致的。

3.编写 8 比特位宽输出加法器的 Verilog HDL 代码

```
module top(
   IN1     ,
   IN2     ,
   OUT     );
input signed [7:0] IN1, IN2;
output signed [8:0] OUT;
reg signed [8:0] OUT;
always@(IN1 or IN2) begin //  生成组合逻辑的 always  块
   OUT = IN1 + IN2;
end
endmodule
```

4.仿真结果波形

由仿真波形可知，4 位补码加法器和 8 位补码加法器的延迟时间相差不多，基本都是 8ns。

带流水线的加法器

一.实验内容

1.不改变流水线的级数，把加法器的输入信号改成 8 比特位宽，编译，波形仿真，和不带流水线的情况对比一下，你有什么结论？

2.在 8 比特输入位宽的情况下，在输入上再添加一级流水线，观察编译和仿真的结果，你有什么结论？

二.实验步骤

1.编写 8 位输入带一级流水线的加法器的 Verilog HDL 代码

```
module top(
   IN1     ,
   IN2     ,
   CLK     ,
   OUT     );
input    [7:0] IN1, IN2;
input CLK;
output    [8:0] OUT;
reg [7:0] in1_d1R, in2_d1R;
reg    [8:0] adder_out, OUT;
always@(posedge CLK) begin //  生成 D 触发器的 always 块
   in1_d1R <= IN1;
   in2_d1R <= IN2;
   OUT         <= adder_out;
end
always@(in1_d1R or in2_d1R) begin //  生成组合逻辑的 always  块
   adder_out = in1_d1R + in2_d1R;
end
endmodule
```

2.仿真结果波形

带有流水线的加法器相较于没有流水线的加法器拥有更短的毛刺，但输出延时更长。

4.编写 8 位输入带两级级流水线的加法器的 Verilog HDL 代码

```
module top(
   IN1     ,
   IN2     ,
   CLK     ,
   OUT     );
input    [7:0] IN1, IN2;
input CLK;
output    [8:0] OUT;
reg [7:0] in1_d1R, in2_d1R, in1_d2R, in2_d2R;
reg    [8:0] adder_out, OUT;
always@(posedge CLK) begin //  生成 D 触发器的 always 块
   in1_d1R <= IN1;
   in1_d2R <= in1_d1R;
   in2_d1R <= IN2;
```

```
    in2_d2R <= in2_d1R;
    OUT         <= adder_out;
end
always@(in1_d2R or in2_d2R) begin //  生成组合逻辑的 always 块
    adder_out = in1_d2R + in2_d2R;
end
endmodule
```

5.仿真结果波形

增加一级流水线后使得毛刺的时间长度进一步减小，但是输出延迟变得更大。

乘法器

一.实验内容

1.改变乘法器的输入位宽为 8 比特，编译，波形仿真，观察信号毛刺的时间长度。

2.选一款没有硬件乘法器的 FPGA 芯片（例如 Cyclone EP1C6）对比 8 比特的乘法器和加法器两者编译之后的资源开销(Logic Cell 的数目）

3.编写一个输入和输出都有 D 触发器的流水线乘法器代码，编译后波形仿真，观察组合逻辑延迟和毛刺的时间，和不带流水线的情况下对比。

二.实验步骤

1.编写 8 位输入的乘法器的 Verilog HDL 代码

```
//////////////////   有符号的 2 补码乘法器   ////////////////////////
module top(
   IN1     ,
   IN2     ,
   OUT     );
  input signed[7:0] IN1, IN2;
  output signed [15:0] OUT;
  reg signed[15:0] OUT;
  always@(IN1 or IN2) begin //  生成组合逻辑的 always  块
    OUT = IN1 * IN2;
end
endmodule
```

3.没有硬件乘法器的 FPGA 芯片的 8 比特加法器和乘法器的资源开销对比

加法器资源开销：

乘法器硬件开销：

由上图可见，乘法器相较加法器更加消耗资源。

4.输入和输出都有 D 触发器的流水线乘法器的 Verilog HDL 代码

```
//////////////////   带有流水线的补码乘法器   ////////////////////////
module top(
   IN1     ,
   IN2     ,
   CLK     ,
   OUT     );
  input signed[7:0] IN1, IN2;
  input CLK;
  output signed [15:0] OUT;
  reg signed[15:0] OUT;
  reg signed[7:0] in1_d1R, in2_d1R;
  reg signed[15:0] mul_out;
always@(posedge CLK) begin //  生成 D 触发器的 always 块
   in1_d1R <= IN1;
   in2_d1R <= IN2;
   OUT        <= mul_out;
end
  always@(in1_d1R or in2_d1R) begin //  生成组合逻辑的 always  块
```

```
    mul_out = in1_d1R * in2_d1R;
end
endmodule
```

5.仿真结果波形

有仿真波形可得：带有流水线的乘法器的组合逻辑延迟和毛刺的时间约为 2ns，相比不带流水线的乘法器要减少了一半。

计数器

一.实验内容

1.设计一个最简单的计数器，只有一个 CLK 输入和一个 Overflow 输出，当计数到最大值的时钟周期 CLK 输出 1

2.设计复杂的计数器，和本例相似，带有多种信号，其中同步清零 CLR 的优先级最高，使能 EN 次之，LOAD 最低。

二.实验步骤

1.编写简单计数器的 Verilog HDL 代码

```
////////////////////  计数器代码   ////////////////////////
module top(
   CLK     , //  时钟，上升沿有效
   OV       );//  计数溢出信号，计数值为最大值时该信号为 1
input CLK ;
output OV;
reg OV;
reg [3:0] CNTVAL, cnt_next;
//  电路编译参数，最大计数值
parameter CNT_MAX_VAL = 9;
//  组合逻辑，生成 cnt_next
always @(CNTVAL) begin
   if(CNTVAL < CNT_MAX_VAL) begin //  未计数到最大值，   下一值加 1
      cnt_next = CNTVAL + 1'b1;
   end
   else begin //  计数到最大值，下一计数值为 0
      cnt_next = 0;
   end
end
//  时序逻辑 更新下一时钟周期的计数值
// CNTVAL  会被编译为 D 触发器
always @ (posedge CLK) begin
      CNTVAL <= cnt_next;
end
//  组合逻辑，生成 OV
always @ (CNTVAL) begin
   if(CNTVAL == CNT_MAX_VAL)
      OV = 1;
   else
      OV = 0;
end
endmodule
```

2.仿真结果波形

如波形图所示，当经过 9 个周期的时钟信号后 OV 端口输出一个高电平的溢出信号。

3.编写复杂计数器的 Verilog HDL 代码

```
////////////////////  计数器代码   ////////////////////////
```

```
module top(
   RST     , //  异步复位，  高有效
   CLK     , //  时钟，上升沿有效
   EN      , //  输入的计数使能，高有效
   CLR     , //  输入的清零信号，高有效
   LOAD    , //  输入的数据加载使能信号，高有效
   DATA    , //  输入的加载数据信号
   CNTVAL, //  输出的计数值信号
   OV      );// 计数溢出信号，计数值为最大值时该信号为 1
input RST     , CLK     , EN      , CLR     , LOAD    ;
input [3:0] DATA ;
output [3:0] CNTVAL;
output OV;
reg [3:0] CNTVAL, cnt_next;
reg OV;
//  电路编译参数，最大计数值
parameter CNT_MAX_VAL = 9;
//  组合逻辑，生成 cnt_next
// 1st clr ,2nd en , 3rd load
always @(EN or CLR or LOAD or DATA or CNTVAL) begin
   if(CLR) begin       //  清零有效
      cnt_next = 0;
   end
   else begin    //  清零无效
      if(EN) begin //  使能有效
         if(LOAD) begin //  加载有效
            cnt_next = DATA;
         end
         else begin         //  加载无效，正常计数
            //  使能有效，清零和加载都无效，根据当前计数值计算下一值
            if(CNTVAL < CNT_MAX_VAL) begin //  未计数到最大值，   下一值加 1
               cnt_next = CNTVAL + 1'b1;
            end
            else begin //  计数到最大值，下一计数值为 0
               cnt_next = 0;
            end
         end // else LOAD
      end // EN
      else begin    //  使能无效，计数值保持不动
         cnt_next = CNTVAL;
      end // else EN
   end //else CLR
end
//  时序逻辑 更新下一时钟周期的计数值
```

```
// CNTVAL 会被编译为 D 触发器
always @ (posedge CLK or posedge RST) begin
   if(RST)
      CNTVAL <= 0;
   else
      CNTVAL <= cnt_next;
end
//  组合逻辑，生成 OV
always @ (CNTVAL) begin
   if(CNTVAL == CNT_MAX_VAL)
      OV = 1;
   else
      OV = 0;
end
endmodule
```

4.仿真结果波形

由仿真结果可知，计数器按照同步清零 CLR 的优先级最高，使能 EN 次之，LOAD 最低的设置工作。

状态机

一.实验内容

1.设计一个用于识别 2 进制序列“1011”的状态机

基本要求：

电路每个时钟周期输入 1 比特数据，当捕获到 1011 的时钟周期，电路输出 1，否则输出 0

使用序列 101011010 作为输出的测试序列

扩展要求：

给你的电路添加输入使能端口，只有输入使能 EN 为 1 的时钟周期，才从输入的数据端口向内部获取 1 比特序列数据。

二.实验步骤

1.绘制状态跳转逻辑表

当前状态	IN	EN	次态
ST_00	0		ST_0
ST_00	1		ST_0
ST_01	0		ST_0
ST_01	1		ST_0
ST_10	0		ST_1
ST_10	1		ST_2
ST_11	0		ST_1
ST_11	1		ST_1
ST_20	0		ST_2
ST_20	1		ST_0
ST_21	0		ST_2
ST_21	1		ST_3
ST_30	0		ST_3
ST_30	1		ST_2
ST_31	0		ST_3
ST_31	1		ST_4
ST_4X	X		ST_0

2.绘制输出逻辑表

当前状态	输出
ST_00	
ST_10	
ST_20	
ST_30	
ST_41	

3.编写 Verilog HDL 代码

```
//////////////////   三段式状态机代码   ////////////////////////
module top(
   CLK          ,    // clock
   RST          ,    // reset
   IN           ,    // input
   EN           ,    // EN
   OUT          );   // output
```

```
input    CLK          ;
input    RST          ;
input    EN           ;
input    IN           ;
output OUT            ;
parameter ST_0 = 0;
parameter ST_1 = 1;
parameter ST_2 = 2;
parameter ST_3 = 3;
parameter ST_4 = 4;
reg [2:0]stateR          ;
reg [2:0]next_state     ;
reg OUT                        ;
// calc next state
always @ (IN or EN or stateR) begin
   case (stateR)
      ST_0 :begin if(IN==0&&EN==0) next_state = ST_0 ; else if(IN==0&&EN==1) next_state
= ST_0; else if(IN==1&&EN==0) next_state = ST_0; else if(IN==1&&EN==1) next_state =
ST_1; end
      ST_1 :begin if(IN==0&&EN==0) next_state = ST_1 ; else if(IN==0&&EN==1) next_state
= ST_2; else if(IN==1&&EN==0) next_state = ST_1; else if(IN==1&&EN==1) next_state =
ST_1; end
      ST_2 :begin if(IN==0&&EN==0) next_state = ST_2 ; else if(IN==0&&EN==1) next_state
= ST_0; else if(IN==1&&EN==0) next_state = ST_2; else if(IN==1&&EN==1) next_state =
ST_3; end
      ST_3 :begin if(IN==0&&EN==0) next_state = ST_3 ; else if(IN==0&&EN==1) next_state
= ST_2; else if(IN==1&&EN==0) next_state = ST_3; else if(IN==1&&EN==1) next_state =
ST_4; end
      ST_4 :begin next_state = ST_0; end
   endcase
end
// calc output
always @ (stateR) begin
   if(stateR == ST_4)
      OUT = 1'b1;
   else
      OUT = 1'b0;
end
// state DFF
always @ (posedge CLK or posedge RST)begin
   if(RST)
      stateR <= ST_0;
   else
      stateR <= next_state;
```

```
end
endmodule
```

移位寄存器

一.实验内容

设计一个带加载使能和移位使能的并入串出的移位寄存器，电路的 RTL 结构图如下图图所示。

二.实验步骤

1.编写移位寄存器代码

```
//////////////////////////////PISO WITH EN_LOAD AND EN_SHIFT//////////////////////////////
module top(
   CLK            , //时钟信号
   RST            , //复位信号输入
   EN_LOAD        , //加载输入数据使能
   EN_SHIFT       , //移位使能
   IN             , //并行输入数据
   OV             , //一组数据完全移出提示
   OUT            );//穿行输出
input RST, CLK, EN_LOAD, EN_SHIFT;
input [7:0] IN;
output OUT,OV;
reg shift_R,OV;
reg [7:0] shift_V;
reg [3:0] n;//移位次数计数
assign OUT = shift_R;//最右端移出数据
always @ (posedge CLK or posedge RST) begin
   if(RST) begin
      shift_R <= 0;
      shift_V <= 0;
      n <= 0;
   end
   else begin
      if(EN_SHIFT) begin
         if(EN_LOAD) begin
            shift_V <= IN;并行输入载入
         end
         else begin
            shift_R <= shift_V[0];
            shift_V[6:0] <= shift_V[7:1];
            shift_V[7] <= 0;//一次移位完成
            n <= n + 1;//移位次数+1
         end
      end
      else begin
         shift_R <= shift_R;//未使能移位，保持不动
      end
   end
```

```
end
always @ (n) begin
   if(n==8)   //当移完一组（8bit）数据后 OV 端口输出 1 提示
      OV = 1;
   else
      OV = 0;
end
endmodule
```

时钟分频器

一.实验内容

设计一个偶分频器（奇分频器）

1.偶分频模块设计

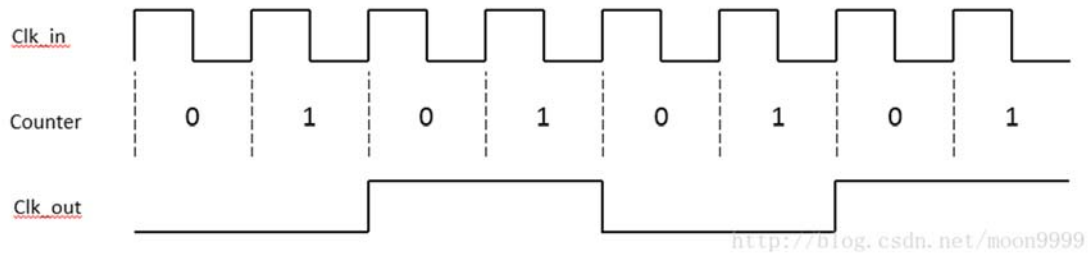
偶分频意思是时钟模块设计最为简单。首先得到分频系数 M 和计数器值 N。

M = 时钟输入频率 / 时钟输出频率

N = M / 2

如输入时钟为 50M，输出时钟为 25M，则 M=2，N=1。偶分频则意味着 M 为偶数。

以 M=4，N=2 为例



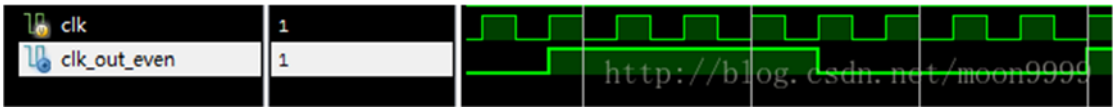
因此只需要将 counter 以 clk_in 为时钟驱动计数，当 counter = (N-1)时，clk_out 翻转即可。

verilog 代码如下，其中 WIDTH 为(N 的位宽-1):

```
module time_adv_even #(
parameter N = 2,
	WIDTH = 7
)
(
	input clk,
	input rst,
	output reg clk_out
	);

reg [WIDTH:0]counter;
always @(posedge clk or posedge rst) begin
	if (rst) begin
		// reset
		counter <= 0;
	end
	else if (counter == N-1) begin
		counter <= 0;
	end
```

```
        else begin
            counter <= counter + 1;
        end
end

always @(posedge clk or posedge rst) begin
        if (rst) begin
            // reset
            clk_out <= 0;
        end
        else if (counter == N-1) begin
            clk_out <= !clk_out;
        end
end

endmodule
```

testbench 测试 8 分频即 N=4，ISE 仿真结果如下：



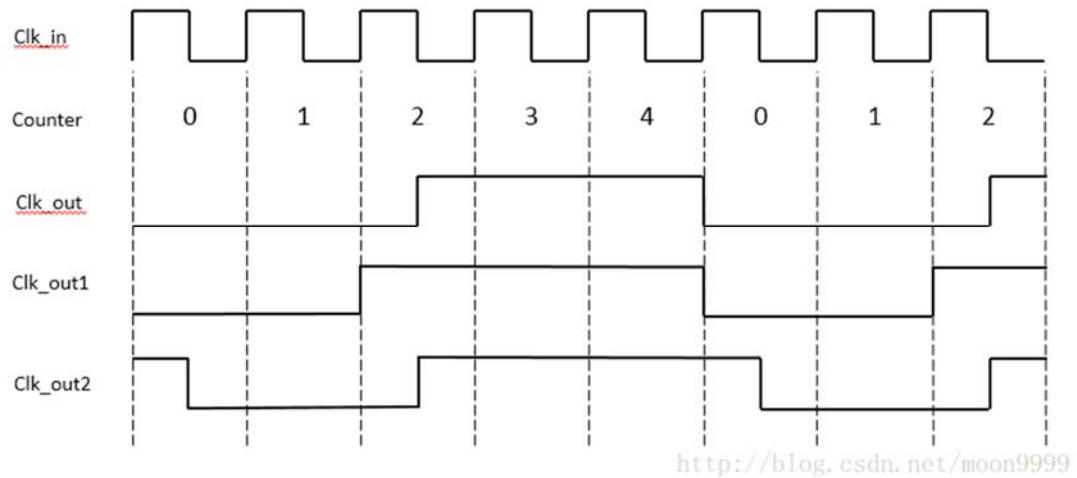
2.奇分频模块设计

奇分频需要通过两个时钟共同得到。首先得到分频系数 M 和计数器值 N。

M = 时钟输入频率 / 时钟输出频率

N = (M-1) / 2

如输入时钟为 50M，输出时钟为 10M，则 M=5，N=2。奇分频则意味着 M 为奇数。
以 M=5，N=2 为例，我们希望得到的输出时钟时序如下：



其中 clk_out 为最终输出时钟，clk_out1 和 clk_out2 为辅助时钟生成。

计数器 counter 由 0 技术至(M-1)。

clk_out1 在在 clk_in 的上升延跳变，条件是 counter==(N-1)或(M-1)。

clk_out2 在在 clk_in 的下降延跳变，条件是 counter==(N-1)或(M-1)。

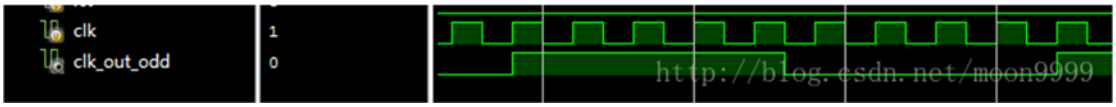
之后 clk_out = clk_out1 & clk_out2 即可得到 M 分频的时钟。

verilog 代码如下，其中 WIDTH 为(N 的位宽-1):

```
module time_adv_odd #(
parameter N = 2,
    WIDTH = 7
     )(
    input clk,
    input rst,
    output clk_out
    );

reg [WIDTH:0]counter;
always @(posedge clk or posedge rst) begin
    if (rst) begin
        // reset
        counter <= 0;
    end
    else if (counter == (N << 1)) begin
        counter <= 0;
    end
    else begin
        counter <= counter + 1;
    end
end

reg clk_out1;
always @(posedge clk or posedge rst) begin
    if (rst) begin
        // reset
        clk_out1 <= 0;
    end
    else if (counter == N-1) begin
        clk_out1 <= !clk_out1;
```

```
        end
        else if (counter == (N << 1)) begin
            clk_out1 <= !clk_out1;
        end
end

reg clk_out2;
always @(negedge clk or posedge rst) begin
        if (rst) begin
            // reset
            clk_out2 <= 0;
        end
        else if (counter == N-1) begin
            clk_out2 <= !clk_out2;
        end
        else if (counter == (N << 1)) begin
            clk_out2 <= !clk_out2;
        end
end

assign clk_out = clk_out1 & clk_out2;
endmodule
```

testbench 测试 9 分频即 N=4，ISE 仿真结果如下：



FIFO 器件实现：

同步 FIFO 的 Verilog 代码 之一

在 modlesim 中验证过。

```
/*****************************************************
A fifo controller verilog description.
*****************************************************/
module fifo(datain, rd, wr, rst, clk, dataout, full, empty);
input [7:0] datain;
input rd, wr, rst, clk;
output [7:0] dataout;
output full, empty;
wire [7:0] dataout;
reg full_in, empty_in;
reg [7:0] mem [15:0];
reg [3:0] rp, wp;
assign full = full_in;
assign empty = empty_in;
// memory read out
assign dataout = mem[rp];
// memory write in
always@(posedge clk) begin
    if(wr && ~full_in) mem[wp]<=datain;
```

```
end
// memory write pointer increment
always@(posedge clk or negedge rst) begin
    if(!rst) wp<=0;
    else begin
      if(wr && ~full_in) wp<= wp+1'b1;
    end
end
// memory read pointer increment
always@(posedge clk or negedge rst)begin
    if(!rst) rp <= 0;
    else begin
      if(rd && ~empty_in) rp <= rp + 1'b1;
    end
end
// Full signal generate
always@(posedge clk or negedge rst) begin
    if(!rst) full_in <= 1'b0;
    else begin
      if( (~rd && wr)&&((wp==rp-1)||(rp==4'h0&&wp==4'hf)))
          full_in <= 1'b1;
      else if(full_in && rd) full_in <= 1'b0;
```

```
    end
end
// Empty signal generate
always@(posedge clk or negedge rst) begin
    if(!rst) empty_in <= 1'b1;
    else begin
      if((rd&&~wr)&&(rp==wp-1 || (rp==4'hf&&wp==4'h0)))
        empty_in<=1'b1;
      else if(empty_in && wr) empty_in<=1'b0;
    end
end
endmodule
```

同步 FIFO 的 Verilog 代码 之二

这一种设计的 FIFO，是基于触发器的。宽度，深度的扩展更加方便，结构化跟强。以下代码在 modelsim 中验证过。

```
module fifo_cell (sys_clk, sys_rst_n, read_fifo, write_fifo,
fifo_input_data,
                  next_cell_data, next_cell_full, last_cell_full,
cell_data_out, cell_full);
                  parameter WIDTH =8;
                  parameter D = 2;
```

```
                    input sys_clk;
                    input sys_rst_n;
                    input read_fifo, write_fifo;
                    input [WIDTH-1:0] fifo_input_data;
                    input [WIDTH-1:0] next_cell_data;
                    input next_cell_full, last_cell_full;
                    output [WIDTH-1:0] cell_data_out;
                    output cell_full;
                    reg [WIDTH-1:0] cell_data_reg_array;
                    reg [WIDTH-1:0] cell_data_ld;
                    reg cell_data_ld_en;
                    reg cell_full;
                    reg cell_full_next;
                    assign cell_data_out=cell_data_reg_array;
                    always @(posedge sys_clk or negedge sys_rst_n)
                      if (!sys_rst_n)
                        cell_full <= #D 0;
                      else if (read_fifo || write_fifo)
                        cell_full <= #D cell_full_next;
                    always @(write_fifo or read_fifo or next_cell_full
or last_cell_full or cell_full)
                      casex ({read_fifo, write_fifo})
```

```
                2'b00: cell_full_next = cell_full;
                2'b01: cell_full_next = next_cell_full;
                2'b10: cell_full_next = last_cell_full;
                2'b11: cell_full_next = cell_full;
            endcase
        always @(posedge sys_clk or negedge sys_rst_n)
            if (!sys_rst_n)
              cell_data_reg_array [WIDTH-1:0] <= #D 0;
            else if (cell_data_ld_en)
              cell_data_reg_array [WIDTH-1:0] <= #D
cell_data_ld [WIDTH-1:0];
        always @(write_fifo or read_fifo or cell_full or
last_cell_full)
            casex
({write_fifo,read_fifo,cell_full,last_cell_full})
                4'bx1_xx: cell_data_ld_en = 1'b1;
                4'b10_01: cell_data_ld_en = 1'b1;
                default: cell_data_ld_en =1'b0;
            endcase
        always @(write_fifo or read_fifo or next_cell_full
or cell_full or last_cell_full or fifo_input_data or next_cell_data)
            casex ({write_fifo, read_fifo, next_cell_full,
```

```
cell_full, last_cell_full})
                        5'b10_x01: cell_data_ld[WIDTH-1:0] =
fifo_input_data[WIDTH-1:0];
                        5'b11_01x: cell_data_ld[WIDTH-1:0] =
fifo_input_data[WIDTH-1:0];
                        default: cell_data_ld[WIDTH-1:0] =
next_cell_data[WIDTH-1:0];
                    endcase
endmodule

module fifo_4cell(sys_clk, sys_rst_n, fifo_input_data, write_fifo,
fifo_out_data,
            read_fifo, full_cell0, full_cell1, full_cell2, full_cell3);
            parameter WIDTH = 8;
            parameter D = 2;
            input sys_clk;
            input sys_rst_n;
            input [WIDTH-1:0] fifo_input_data;
            output [WIDTH-1:0] fifo_out_data;
            input read_fifo, write_fifo;
            output full_cell0, full_cell1, full_cell2, full_cell3;
            wire [WIDTH-1:0] dara_out_cell0, data_out_cell1,
```

```
data_out_cell2,
                        data_out_cell3, data_out_cell4;
        wire full_cell4;
        fifo_cell #(WIDTH,D) cell0
        ( .sys_clk (sys_clk),
          .sys_rst_n (sys_rst_n),
          .fifo_input_data (fifo_input_data[WIDTH-1:0]),
          .write_fifo (write_fifo),
          .next_cell_data (data_out_cell1[WIDTH-1:0]),
          .next_cell_full (full_cell1),
          .last_cell_full (1'b1),
          .cell_data_out (fifo_out_data [WIDTH-1:0]),
          .read_fifo (read_fifo),
          .cell_full (full_cell0)
         );
        fifo_cell #(WIDTH,D) cell1
        ( .sys_clk (sys_clk),
          .sys_rst_n (sys_rst_n),
          .fifo_input_data (fifo_input_data[WIDTH-1:0]),
          .write_fifo (write_fifo),
          .next_cell_data (data_out_cell2[WIDTH-1:0]),
          .next_cell_full (full_cell2),
```

```
  .last_cell_full (full_cell0),
  .cell_data_out (data_out_cell1[WIDTH-1:0]),
  .read_fifo (read_fifo),
  .cell_full (full_cell1)
 );
fifo_cell #(WIDTH,D) cell2
( .sys_clk (sys_clk),
  .sys_rst_n (sys_rst_n),
  .fifo_input_data (fifo_input_data[WIDTH-1:0]),
  .write_fifo (write_fifo),
  .next_cell_data (data_out_cell3[WIDTH-1:0]),
  .next_cell_full (full_cell3),
  .last_cell_full (full_cell1),
  .cell_data_out (data_out_cell2[WIDTH-1:0]),
  .read_fifo (read_fifo),
  .cell_full (full_cell2)
 );
fifo_cell #(WIDTH,D) cell3
( .sys_clk (sys_clk),
  .sys_rst_n (sys_rst_n),
  .fifo_input_data (fifo_input_data[WIDTH-1:0]),
  .write_fifo (write_fifo),
```

```
                .next_cell_data (data_out_cell4[WIDTH-1:0]),
                .next_cell_full (full_cell4),
                .last_cell_full (full_cell2),
                .cell_data_out (data_out_cell3[WIDTH-1:0]),
                .read_fifo (read_fifo),
                .cell_full (full_cell3)
               );
               assign data_out_cell4[WIDTH-1:0] = {WIDTH{1'B0}};
               assign full_cell4 = 1'b0;
endmodule
```

……

异步 FIFO 的 Verilog 代码 之一

这个是基于 RAM 的异步 FIFO 代码，个人认为代码结构简单易懂，非常适合于考试中填写。记得 10 月份参加威盛的笔试的时候，就考过异步 FIFO 的实现。想当初要是早点复习，可能就可以通过威盛的笔试了。

与之前的用 RAM 实现的同步 FIFO 的程序相比，异步更为复杂。增加了读写控制信号的跨时钟域的同步。此外，判空与判满的也稍有不同。

```
module fifo1(rdata, wfull, rempty, wdata, winc, wclk, wrst_n,rinc,
rclk, rrst_n);
parameter DSIZE = 8; parameter ASIZE = 4;
```

```
output [DSIZE-1:0] rdata;
output wfull;
output rempty;
input [DSIZE-1:0] wdata;
input winc, wclk, wrst_n;
input rinc, rclk, rrst_n;
reg wfull,rempty;
reg [ASIZE:0] wptr, rptr, wq2_rptr, rq2_wptr, wq1_rptr,rq1_wptr;
reg [ASIZE:0] rbin, wbin;
reg [DSIZE-1:0] mem[0:(1<<ASIZE)-1];
wire [ASIZE-1:0] waddr, raddr;
wire [ASIZE:0] rgraynext, rbinnext,wgraynext,wbinnext;
wire rempty_val,wfull_val;
//-----------------双口 RAM 存储器--------------------
assign rdata=mem[raddr];
always@(posedge wclk)
if (winc && !wfull) mem[waddr] <= wdata;
//-------------同步 rptr  指针-------------------------
always @(posedge wclk or negedge wrst_n)
if (!wrst_n) {wq2_rptr,wq1_rptr} <= 0;
else {wq2_rptr,wq1_rptr} <= {wq1_rptr,rptr};
//-------------同步 wptr 指针--------------------------
```

```
always @(posedge rclk or negedge rrst_n)
if (!rrst_n) {rq2_wptr,rq1_wptr} <= 0;
else {rq2_wptr,rq1_wptr} <= {rq1_wptr,wptr};
//-------------rempty 产生与 raddr 产生-------------------
always @(posedge rclk or negedge rrst_n) // GRAYSTYLE2
pointer
begin
if (!rrst_n) {rbin, rptr} <= 0;
else {rbin, rptr} <= {rbinnext, rgraynext};
end
// Memory read-address pointer (okay to use binary to address
memory)
assign raddr = rbin[ASIZE-1:0];
assign rbinnext = rbin + (rinc & ~rempty);
assign rgraynext = (rbinnext>>1) ^ rbinnext;
// FIFO empty when the next rptr == synchronized wptr or on
reset
assign rempty_val = (rgraynext == rq2_wptr);
always @(posedge rclk or negedge rrst_n)
begin
if (!rrst_n) rempty <= 1'b1;
else rempty <= rempty_val;
```

```
end
//---------------wfull 产生与 waddr 产生---------------------------
--
always @(posedge wclk or negedge wrst_n) // GRAYSTYLE2
pointer
if (!wrst_n) {wbin, wptr} <= 0;
else {wbin, wptr} <= {wbinnext, wgraynext};
// Memory write-address pointer (okay to use binary to address
memory)
assign waddr = wbin[ASIZE-1:0];
assign wbinnext = wbin + (winc & ~wfull);
assign wgraynext = (wbinnext>>1) ^ wbinnext;
assign wfull_val = (wgraynext=={~wq2_rptr[ASIZE:ASIZE-1],
wq2_rptr[ASIZE-2:0]}); //:ASIZE-1]
always @(posedge wclk or negedge wrst_n)
if (!wrst_n) wfull <= 1'b0;
else wfull <= wfull_val;
endmodule
```

.....

异步 FIFO 的 Verilog 代码 之二

与前一段异步 FIFO 代码的主要区别在于，空/满状态标志的不同算法。

第一个算法：*Clifford E. Cummings 的文章中提到的 STYLE #1，构造一个指针宽度为 N+1，深度为 2^N 字节的 FIFO（为便方比较将格雷码指针转换为二进制指针）。当指针的二进制码中最高位不一致而其它 N 位都 相等时，FIFO 为满（在 Clifford E. Cummings 的文章中以格雷码表示是前两位均不相同，而后两位 LSB 相同为满，这与换成二进制表示的 MSB 不同其他相同为满是一样的）。当指针完全相等时，FIFO 为空。*

这种方法思路非常明了，为了比较不同时钟产生的指针，需要把不同时钟域的信号同步到本时钟域中来，而使用 Gray 码的目的就是使这个异步同步化的过 程发生亚稳态的机率最小，而为什么要构造一个 N+1 的指针，Clifford E. Cummings 也阐述的很明白，有兴趣的读者可以看下作者原文是怎么论述的，Clifford E. Cummings 的这篇文章有 Rev1.1 \ Rev1.2 两个版本，两者在比较 Gray 码指针时的方法略有不同，个 Rev1.2 版更为精简。

第二种算法：*Clifford E. Cummings 的文章中提到的 STYLE #2。它将 FIFO 地址分成了 4 部分，每部分分别用高两位的 MSB 00 、01、11、 10 决定 FIFO 是否为 going full 或 going empty (即将满或空)。如果写指针的高两位 MSB 小于读指针的高两位 MSB 则 FIFO 为“几乎满”，若写指针的高两位 MSB 大于读指针的高两位 MSB 则 FIFO 为“几乎空”。*

它是利用将地址空间分成 4 个象限（也就是四个等大小的区域），然后观察两个指针的相对位置，如果写指针落后读指针一个象限（25%

的距离，呵呵），则证明很可能要写满，反之则很可能要读空，这个时候分别设置两个标志位 dirset 和 dirrst，然后在地址完全相等的情况下，如果 dirset 有效就 是写满，如果 dirrst 有效就是读空。

这种方法对深度为 2^N 字节的 FIFO 只需 N 位的指针即可，处理的速度也较第一种方法快。

这段是说明的原话，算法一，还好理解。算法二，似乎没有说清楚，不太明白。有兴趣的可以查查论文，详细研究下。

总之，第二种写法是推荐的写法。因为异步的多时钟设计应按以下几个原则进行设计：

1，尽可能的将多时钟的逻辑电路（非同步器）分割为多个单时钟的模块，这样有利于静态时序分析工具来进行时序验证。

2，同步器的实现应使得所有输入来自同一个时钟域，而使用另一个时钟域的异步时钟信号采样数据。

3，面向时钟信号的命名方式可以帮助我们确定那些在不同异步时钟域间需要处理的信号。

4，当存在多个跨时钟域的控制信号时，我们必须特别注意这些信号，保证这些控制信号到达新的时钟域仍然能够保持正确的顺序。

```
module fifo2 (rdata, wfull, rempty, wdata,
winc, wclk, wrst_n, rinc, rclk, rrst_n);
parameter DSIZE = 8;
parameter ASIZE = 4;
output [DSIZE-1:0] rdata;
```

```
output wfull;
output rempty;
input [DSIZE-1:0] wdata;
input winc, wclk, wrst_n;
input rinc, rclk, rrst_n;
wire [ASIZE-1:0] wptr, rptr;
wire [ASIZE-1:0] waddr, raddr;
async_cmp #(ASIZE) async_cmp(.aempty_n(aempty_n),
.afull_n(afull_n),
.wptr(wptr), .rptr(rptr),
.wrst_n(wrst_n));
fifomem2 #(DSIZE, ASIZE) fifomem2(.rdata(rdata),
.wdata(wdata),
.waddr(wptr),
.raddr(rptr),
.wclken(winc),
.wclk(wclk));
rptr_empty2 #(ASIZE) rptr_empty2(.rempty(rempty),
.rptr(rptr),
.aempty_n(aempty_n),
.rinc(rinc),
.rclk(rclk),
```

```
.rrst_n(rrst_n));
wptr_full2 #(ASIZE) wptr_full2(.wfull(wfull),
.wptr(wptr),
.afull_n(afull_n),
.winc(winc),
.wclk(wclk),
.wrst_n(wrst_n));
endmodule
module fifomem2 (rdata, wdata, waddr, raddr, wclken, wclk);
parameter DATASIZE = 8; // Memory data word width
parameter ADDRSIZE = 4; // Number of memory address bits
parameter DEPTH = 1<<ADDRSIZE; // DEPTH = 2**ADDRSIZE
output [DATASIZE-1:0] rdata;
input [DATASIZE-1:0] wdata;
input [ADDRSIZE-1:0] waddr, raddr;
input wclken, wclk;
`ifdef VENDORRAM
// instantiation of a vendor's dual-port RAM
VENDOR_RAM MEM (.dout(rdata), .din(wdata),
.waddr(waddr), .raddr(raddr),
.wclken(wclken), .clk(wclk));
`else
```

```
reg [DATASIZE-1:0] MEM [0:DEPTH-1];
assign rdata = MEM[raddr];
always @(posedge wclk)
if (wclken) MEM[waddr] <= wdata;
`endif
endmodule
module async_cmp (aempty_n, afull_n, wptr, rptr, wrst_n);
parameter ADDRSIZE = 4;
parameter N = ADDRSIZE-1;
output aempty_n, afull_n;
input [N:0] wptr, rptr;
input wrst_n;
reg direction;
wire high = 1'b1;
wire dirset_n = ~( (wptr[N]^rptr[N-1]) & ~(wptr[N-1]^rptr[N]));
wire dirclr_n = ~((~(wptr[N]^rptr[N-1]) & (wptr[N-1]^rptr[N])) |
~wrst_n);
always @(posedge high or negedge dirset_n or negedge
dirclr_n)
if (!dirclr_n) direction <= 1'b0;
else if (!dirset_n) direction <= 1'b1;
else direction <= high;
```

```
//always @(negedge dirset_n or negedge dirclr_n)
//if (!dirclr_n) direction <= 1'b0;
//else direction <= 1'b1;
assign aempty_n = ~((wptr == rptr) && !direction);
assign afull_n = ~((wptr == rptr) && direction);
endmodule
module rptr_empty2 (rempty, rptr, aempty_n, rinc, rclk, rrst_n);
parameter ADDRSIZE = 4;
output rempty;
output [ADDRSIZE-1:0] rptr;
input aempty_n;
input rinc, rclk, rrst_n;
reg [ADDRSIZE-1:0] rptr, rbin;
reg rempty, rempty2;
wire [ADDRSIZE-1:0] rgnext, rbnext;
//---------------------------------------------------------
// GRAYSTYLE2 pointer
//---------------------------------------------------------
always @(posedge rclk or negedge rrst_n)
if (!rrst_n) begin
rbin <= 0;
rptr <= 0;
```

```
end
else begin
rbin <= rbnext;
rptr <= rgnext;
end
//------------------------------------------------------------
// increment the binary count if not empty
//------------------------------------------------------------
assign rbnext = !rempty ? rbin + rinc : rbin;
assign rgnext = (rbnext>>1) ^ rbnext; // binary-to-gray
conversion
always @(posedge rclk or negedge aempty_n)
if (!aempty_n) {rempty,rempty2} <= 2'b11;
else {rempty,rempty2} <= {rempty2,~aempty_n};
endmodule
module wptr_full2 (wfull, wptr, afull_n, winc, wclk, wrst_n);
parameter ADDRSIZE = 4;
output wfull;
output [ADDRSIZE-1:0] wptr;
input afull_n;
input winc, wclk, wrst_n;
reg [ADDRSIZE-1:0] wptr, wbin;
```

```
reg wfull, wfull2;
wire [ADDRSIZE-1:0] wgnext, wbnext;
//------------------------------------------------------------------
// GRAYSTYLE2 pointer
//------------------------------------------------------------------
always @(posedge wclk or negedge wrst_n)
if (!wrst_n) begin
wbin <= 0;
wptr <= 0;
end
else begin
wbin <= wbnext;
wptr <= wgnext;
end
//------------------------------------------------------------------
// increment the binary count if not full
//------------------------------------------------------------------
assign wbnext = !wfull ? wbin + winc : wbin;
assign wgnext = (wbnext>>1) ^ wbnext; // binary-to-gray
conversion
always @(posedge wclk or negedge wrst_n or negedge afull_n)
if (!wrst_n ) {wfull,wfull2} <= 2'b00;
```

```
else if (!afull_n) {wfull,wfull2} <= 2'b11;
else {wfull,wfull2} <= {wfull2,~afull_n};
endmodule
```