D类功放

2120160972 周文茜

1. 硬件原理图
2. PWM部分
3. 总线驱动电路

高电平时输出5V，低电平时输出0V

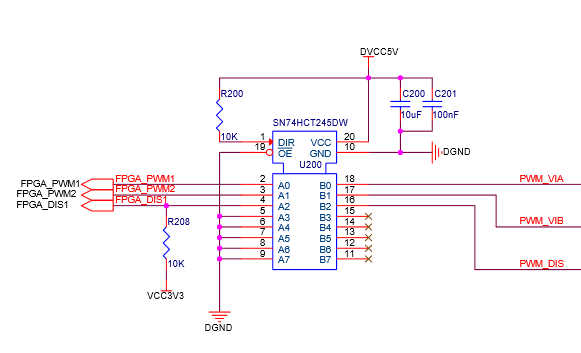
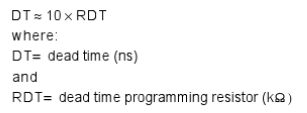


图1 总线驱动电路

1. 隔离门驱动器电路

高电平时输出12V，低电平时输出0V，

同时加入硬件死区。不过按照数据手册，死区20ns的话电阻R15应该改成2K



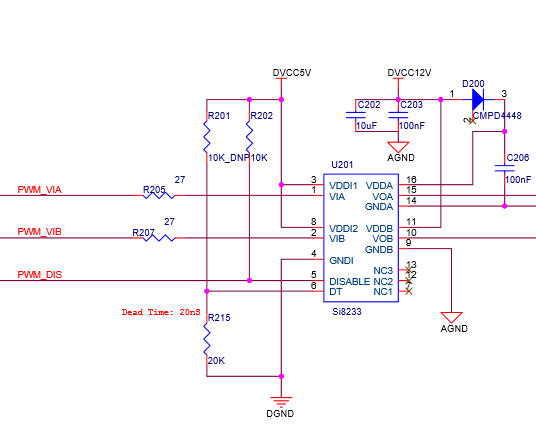


图2 隔离门驱动电路

1. ADC部分

参考电压为LM4041的输出电压1.225V，

GAIN引脚接高电平，输入电压差的范围在-1.225V~1.225V，

OF引脚接低电平，偏移二进制编码输出，因此输入电压和输出数字量的换算关系为：

考虑到Q0~Q9和I0~I9都接到FPGA上，OS引脚应该为高电平，也就是改成R226作为DNP而R227保留，这样ADC可以工作在并行模式而非混合模式

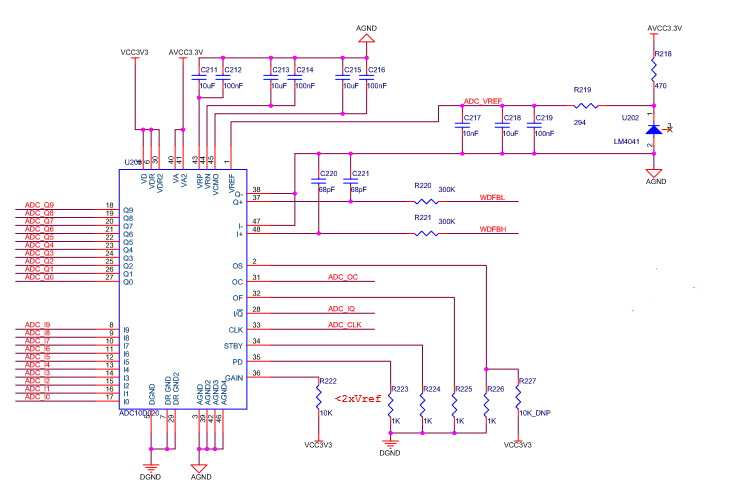


图3 模数转换电路

1. 负载部分

其中R203、R210、C207、C210起阻容吸收作用

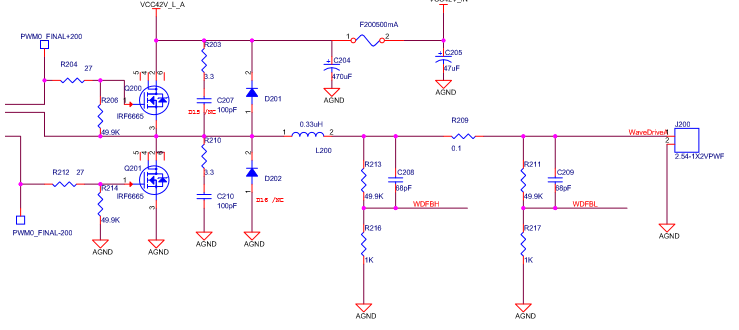


图4 负载电路

1. 算法原理
2. 系统框图



图5 系统工作框图

1. 时序设计

按照条件，PWM波频率是5M，ADC采样频率是20M，所用FPGA最大工作频率约470M。

一开始将锁相环倍频后频率设成400M，RTL级仿真工作正常，而门级仿真工作一段时间就会出错，于是将频率改为200M，门级仿真正常工作，遂将200M作为基础工作频率。

后来为了ADC换算准确，PWM计数器满值希望是2的幂，于是将工作频率换成320M，这样PWM频率是5M时，计数可以是64，结果320M好像还是太快，最终换成160M。

由于生成PWM波的算法采用的是不对称规则采样法，因此一个PWM周期内会计算两次占空比，相当于频率为10M，因此对于20M的ADC，每采集两次计算平均值作为当前电压。

查看ADC10D020的数据手册，并行模式下，采样点大约在时钟的下降沿，且距下降沿有tAD（2.4ns）的延时，因此延时一个工作周期再进行采样。相应的由PID控制算法计算控制量则延时两个工作周期。





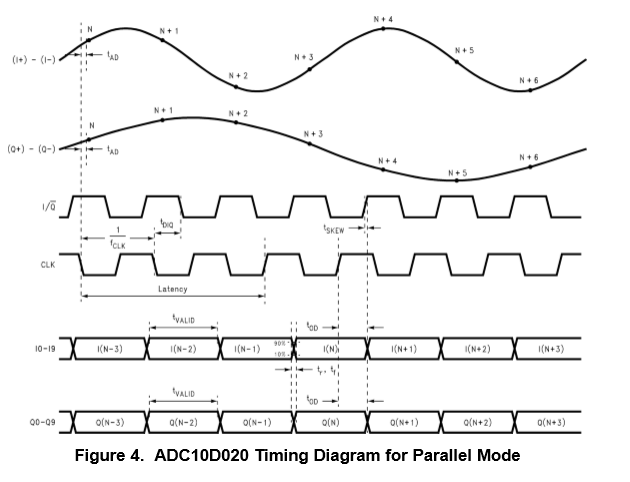


图6 ADC10D020时序图

1. PWM原理

PWM波使用不对称规则采样法生成，是自然采样法的一种近似。自然采样法的原理如图7所示。

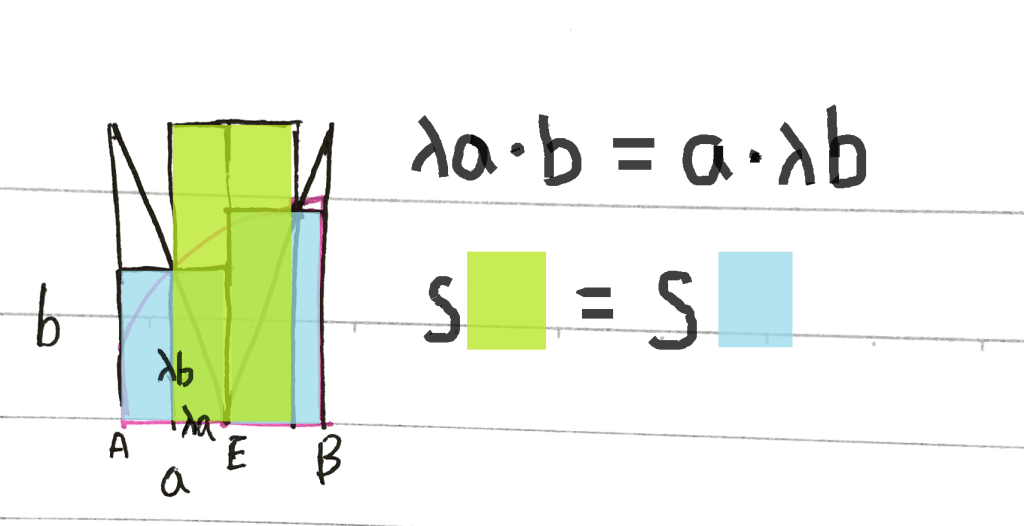


图7 自然采样法面积等效原理

将自然采样法近似为不对称规则采样法后则不用造三角波和目标值时刻作比较，只需要在三角波的波峰和波谷对应时刻对目标值进行采样即可算出相应占空比。示意图如图8所示。相较于图9的平均对称规则采样法，不对称规则采样法的优点为没有滞后，当前算出占空比即可实现，而用平均对称规则采样法算出来的电平改变点在采样点之前，所以必须在下一个周期才能实现。

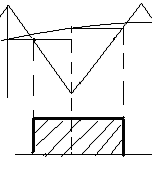
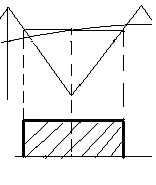
 

图8 不对称规则采样法示意图 图9 平均对称规则采样法示意图

1. PID原理

为了避免产生负的占空比，加入了输出限幅，

为了避免积分项影响过大，加入了积分限幅。

1. Verilog程序
2. verilog\_yunfang.v

如图10所示，分成signal\_module,adc\_module,pid\_module,pwm\_module和pll共5个模块。

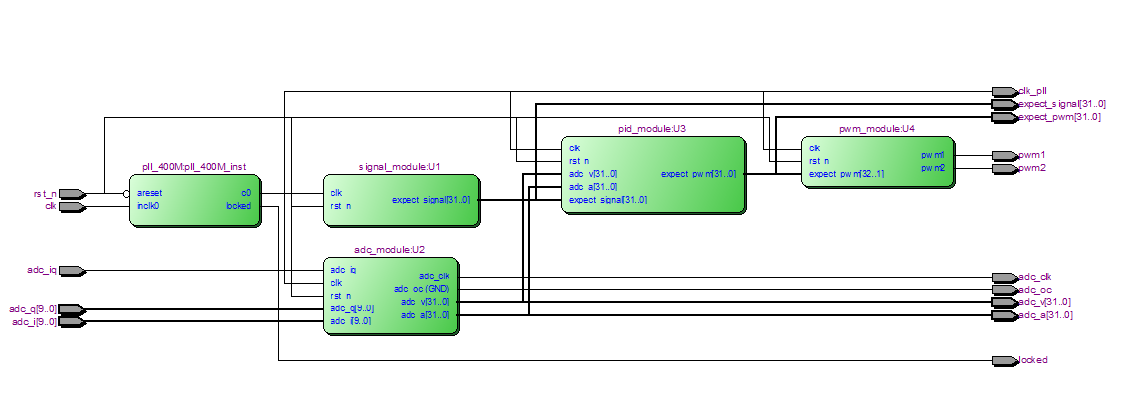


图10 系统RTL视图

其中signal\_module用来生成目标信号，adc\_module用来处理adc芯片输出的模数转换量，pid\_module用来生成pid控制量，pwm\_module用来生成pwm波，pll用来生成倍频后的时钟信号作为工作时钟。

module verilog\_yunfang(

clk,rst\_n,clk\_pll,locked,

pwm1,pwm2,expect\_pwm,expect\_signal,

adc\_q,adc\_i,adc\_oc,adc\_iq,adc\_clk,

adc\_v,adc\_a

);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

input clk;

input rst\_n;

output pwm1;

output pwm2;

output clk\_pll;

output locked;

output[31:0] expect\_pwm;

output[31:0] expect\_signal;

input[9:0] adc\_q;

input[9:0] adc\_i;

output adc\_oc;

input adc\_iq;

output adc\_clk;

output[31:0] adc\_v;

output[31:0] adc\_a;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

wire[31:0] adc\_v;

wire[31:0] adc\_a;

wire clk\_pll;

wire locked;

wire[31:0] expect\_pwm;

wire[31:0] expect\_signal;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

pll\_400M pll\_400M\_inst (

.areset ( !rst\_n ),

.inclk0 ( clk ),

.c0 (clk\_pll),

.locked (locked)

);

signal\_module U1

(

.clk(clk\_pll),

.rst\_n(rst\_n),

.expect\_signal(expect\_signal)

);

adc\_module U2

(

.clk(clk\_pll),

.rst\_n(rst\_n),

.adc\_q(adc\_q),

.adc\_i(adc\_i),

.adc\_oc(adc\_oc),

.adc\_iq(adc\_iq),

.adc\_clk(adc\_clk),

.adc\_v(adc\_v),

.adc\_a(adc\_a)

);

pid\_module U3

(

.clk(clk\_pll),

.rst\_n(rst\_n),

.adc\_v(adc\_v),

.adc\_a(adc\_a),

.expect\_signal(expect\_signal),

.expect\_pwm(expect\_pwm)

);

pwm\_module U4

(

.clk(clk\_pll),

.rst\_n(rst\_n),

.pwm1(pwm1),

.pwm2(pwm2),

.expect\_pwm(expect\_pwm)

);

endmodule

1. signal\_module.v

生成梯形波作为目标波形，存储在expect\_signal中

module signal\_module(

clk,rst\_n,expect\_signal

);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

input clk;

input rst\_n;

output expect\_signal;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

reg[9:0] count\_signal;

reg[3:0] count\_10;

reg signal\_clk;

reg[1:0] state;

integer expect\_signal;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

always @(posedge clk or negedge rst\_n) //为了防止信号变得太快看不出效果，信号周期拉长

begin

if(!rst\_n)

count\_10 <= 1'b0;

else

begin

signal\_clk <= (count\_10 < 5) ? 0 : 1; //信号时钟

if(count\_10 < 9)

count\_10 <= count\_10 + 1'b1;

else

count\_10 <= 1'b0;

end

end

parameter Up = 2'b00, //状态机分别代表梯形波上升、

Hold = 2'b01, //高电平

Down = 2'b10, //下降

Low = 2'b11; //低电平四个状态

always @(posedge signal\_clk or negedge rst\_n)

begin

if(!rst\_n)

begin

expect\_signal <= 0;

state <= Up;

count\_signal <= 0;

end

else

begin

if(count\_signal < 79)

count\_signal <= count\_signal + 1;

else

count\_signal <= 0;

case(state)

Up:

if(count\_signal < 19)

expect\_signal <= expect\_signal+1;

else

state <= Hold;

Hold:

if(count\_signal < 39)

expect\_signal <= 20;

else

state <= Down;

Down:

if(count\_signal < 59)

expect\_signal <= expect\_signal-1;

else

state <= Low;

Low:

if(count\_signal < 79)

expect\_signal <= 0;

else

state <= Up;

default:

state <= Up;

endcase

end

end

endmodule

1. adc\_module.v

module adc\_module(

clk,rst\_n,adc\_q,adc\_i,adc\_oc,adc\_iq,adc\_clk,adc\_v,adc\_a

);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

input clk;

input rst\_n;

input[9:0] adc\_q;

input[9:0] adc\_i;

output adc\_oc;

input adc\_iq;

output adc\_clk;

output[31:0] adc\_v;

output[31:0] adc\_a;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

reg[5:0] count\_adc;

reg[2:0] count\_data;

wire[9:0] adc\_q;

wire[9:0] adc\_i;

reg adc\_oc;

wire adc\_iq;

reg adc\_clk;

reg[31:0] adc\_vt;

reg[31:0] adc\_at;

reg[31:0] adc\_v;

reg[31:0] adc\_a;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

always @(negedge clk or negedge rst\_n)

begin

if(!rst\_n)

begin

adc\_v <= 0;

adc\_a <= 0;

adc\_vt <= 0;

adc\_at <= 0;

count\_adc <= 0;

count\_data <= 0;

adc\_clk <= 0;

end

else

begin

adc\_clk <= (count\_adc < 4) ? 0 : 1;

if(count\_adc < 7)

count\_adc <= count\_adc + 1;

else

count\_adc <= 0;

if(count\_adc == 0) //延迟一个工作周期作为转换时间延迟

begin

count\_data <= count\_data + 1;

adc\_vt <= adc\_vt + adc\_i; //累计采集的和

adc\_at <= adc\_at + adc\_q;

end

if(count\_data > 1) //采了两次

begin

adc\_v <= ((adc\_vt >> 6) - 16); //采两次取平均值，右移1位，满值512，右移9位，PWM计数满值16，左移4位，所以共计右移6位，再减去偏移编码时的偏移

adc\_a <= ((adc\_at >> 6) - 16);

adc\_vt <= 0;

adc\_at <= 0;

count\_data <= 0;

end

end

end

endmodule

1. pid\_module.v

pid参数还没有调整，全都设成了1，加入了积分限幅，积分项的值限制在了-5~5之间

module pid\_module(

clk,rst\_n,adc\_v,adc\_a,expect\_signal,expect\_pwm

);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

input clk;

input rst\_n;

input[31:0] adc\_v;

input[31:0] adc\_a;

output[31:0] expect\_pwm;

input[31:0] expect\_signal;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

wire[31:0] adc\_v;

wire[31:0] adc\_a;

integer expect\_pwm;

reg[5:0] count\_pid;

reg pid\_clk;

integer P;

integer I;

parameter kp = 1;

parameter ki = 1;

parameter kd = 1;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

always @(negedge clk or negedge rst\_n)

begin

if(!rst\_n)

begin

count\_pid <= 0;

pid\_clk <= 0;

end

else

begin

pid\_clk <= (count\_pid < 10) ? 0 : 1; //为了配合adc的延时加入的延时

if(count\_pid < 15) //一个pwm周期算两次pid，计算频率10M

count\_pid <= count\_pid + 1;

else

count\_pid <= 0;

end

end

always @(posedge pid\_clk or negedge rst\_n)

begin

if(!rst\_n)

begin

P <= 0;

I <= 0;

expect\_pwm <= 0;

end

else

begin

P <= expect\_signal - adc\_v;

if(I + expect\_signal - adc\_v < -5) //积分限幅，不小于-5

begin

I <= -5;

expect\_pwm <= (expect\_signal - adc\_v - P) \* kd + (expect\_signal - adc\_v) \* kp + (-5) \* ki;

end

else if(I + expect\_signal - adc\_v > 5) //积分限幅，不大于5

begin

I <= 5;

expect\_pwm <= (expect\_signal - adc\_v - P) \* kd + (expect\_signal - adc\_v) \* kp + 5 \* ki;

end

else

begin

I <= I + expect\_signal - adc\_v;

expect\_pwm <= (expect\_signal - adc\_v - P) \* kd + (expect\_signal - adc\_v) \* kp + (I + expect\_signal - adc\_v) \* ki;

end

end

end

endmodule

1. pwm\_module.v

使用不对称规则采样法，目标信号为expect\_pwm,pwm1和pwm2为pwm波形输出

module pwm\_module(

clk,rst\_n,pwm1,pwm2,expect\_pwm

);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

input clk;

input rst\_n;

output pwm1;

output pwm2;

input[32:1] expect\_pwm;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

reg[7:0] count\_pwm; //160M时钟，5M pwm，一个pwm波计数32次

integer duty1;

integer duty2;

reg pwm1;

wire pwm2;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

always @(posedge clk or negedge rst\_n)

begin

if(!rst\_n)

begin

duty1 <= 0;

duty2 <= 0;

pwm1 <= 0;

count\_pwm <= 0;

end

else

begin

if(count\_pwm == 31)

//稍微提前一个工作周期算出占空比这样万一下一个duty1是0就不会措手不及

begin

if(expect\_pwm[32] == 1) duty1 <= 16; //符号位判断，负数输出限幅

else if(expect\_pwm > 16) duty1 <= 0; //正数输出限幅

else duty1 <= 32 - expect\_pwm;

end

if(count\_pwm == 15)

//为了和算duty1时提前的那个占空比保持一致

begin

if(expect\_pwm[32] == 1) duty2 <= 16; //符号位判断，负数输出限幅

else if(expect\_pwm > 16) duty2 <= 32; //正数输出限幅

else duty2 <= 32 + expect\_pwm;

end

if(count\_pwm < 31)

count\_pwm <= count\_pwm + 1;

else

count\_pwm <= 0;

if(count\_pwm < duty1 || count\_pwm >= duty2)

pwm1 <= 0;

else pwm1 <= 1; //duty1和duty2之间为高电平，之外为低电平

end

end

assign pwm2 = ~pwm1; //有了硬件死区，直接反相

endmodule

1. verilog\_yunfang.vt

仿真时用的testbench，造了50M的时钟输入，造了复位信号，还造了ad芯片数字量输出引脚的数据。

`timescale 1 ns/ 1 ps

module verilog\_yunfang\_vlg\_tst();

// constants

// general purpose registers

reg eachvec;

// test vector input registers

reg clk;

reg rst\_n;

// wires

wire pwm1;

wire pwm2;

wire[31:0] expect\_pwm;

wire[31:0] expect\_signal;

reg[9:0] adc\_q;

reg[9:0] adc\_i;

wire adc\_oc;

wire adc\_iq;

wire adc\_clk;

wire[31:0] adc\_v;

wire[31:0] adc\_a;

wire clk\_pll;

wire locked;

// assign statements (if any)

verilog\_yunfang i1 (

// port map - connection between master ports and signals/registers

.clk(clk),

.rst\_n(rst\_n),

.clk\_pll(clk\_pll),

.locked(locked),

.pwm1(pwm1),

.pwm2(pwm2),

.expect\_pwm(expect\_pwm),

.expect\_signal(expect\_signal),

.adc\_q(adc\_q),

.adc\_i(adc\_i),

.adc\_oc(adc\_oc),

.adc\_iq(adc\_iq),

.adc\_clk(adc\_clk),

.adc\_v(adc\_v),

.adc\_a(adc\_a)

);

initial

begin //造复位

rst\_n=0;

#100 rst\_n=1;

#2000000 $stop;

end

initial

begin //造时钟

clk=0;

forever #10 clk=~clk;

end

always @(posedge clk or negedge rst\_n) //造ad输出

begin

if(!rst\_n)

begin

adc\_q <= 0;

adc\_i <= 512;

end

else

begin

adc\_q <= adc\_q + 1;

adc\_i <= adc\_i + 1;

if(adc\_i < 1000)

adc\_i <= adc\_i + 1;

else

adc\_i <= 512;

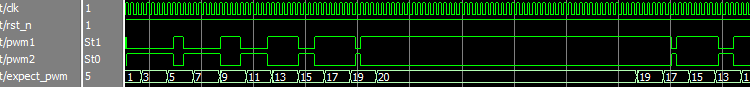
end

end

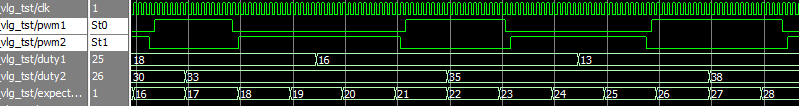
endmodule

1. 仿真结果

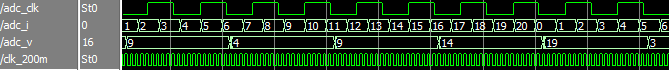
1、pwm仿真结果



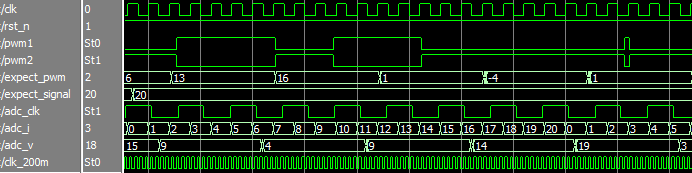
2、pwm好不容易加了死区仿真结果（然而经崔市委提醒才意识到硬件死区因此并不用加）



3、adc采样仿真结果（adc\_v为输出采样信号，adc\_i为造的目标信号）



4、加了PID的整体仿真结果（输出信号也是造的因此看不出来个啥）



1. 心得体会
2. 绝对不要在两个always里面修改或者比较同一个变量
3. always里面只能用非阻塞赋值<=，assign只能用阻塞赋值=
4. always里面只能为reg型赋值，为wire型赋值请找assign
5. 要想生成时钟的话在always里面比较计数值会好些，在assign中比较会有毛刺
6. 比较reg当中的某一位时竟必须要用[n:1]表示法
7. 目前为止除了判断符号位没有找到能够比较reg型负数的有效方法
8. 如果来不及算完下一个敏感信号就来了的话，门级仿真可能运行一段时间会错
9. pll仿真时输入时钟如果和配置时输入时钟不一样，门级仿真可能运行一段时间就会错
10. 其实加入了对偏移二进制码换算的adc部分代码后，门级仿真依然出错中，还没有调好，rtl级倒是很容易就能满足不出错…