开发者说明：

每个成员的学号、姓名、所负责的工作、贡献百分比

12110932 秦李旸（1/3） ALU 模块，Controller 模块，测试用例，uart 接口，报告
12110429 郑余奥泽（1/3） memory 模块 decode 模块 测试用例 vga IO
12110424 张阳（1/3） Ifetc 模块 top 模块 vga 模块

CPU 架构设计说明：

CPU 特性：

ISA（含所有指令（指令名、对应编码、使用方式），参考的 ISA，基于参考 ISA 本次作业所做的更新或优化；寄存器（位宽和数目）等信息）；对于异常处理的支持情况。

ISA：

Type	Name	opC(Ins[31:26])	funC(Ins[5:0])	Format
R	sll	6’ b00_0000	6’ b00_0000	sll $t1,$t2,10
	srl	6’ b00_0000	6’ b00_0010	srl $t1,$t2,10
	sllv	6’ b00_0000	6’ b00_0100	sllv $t1,$t2,$t3
	solve	6’ b00_0000	6’ b00_0110	srlv $t1,$t2,$t3
	sra	6’ b00_0000	6’ b00_0011	sra $t1,$t2,10
	srav	6’ b00_0000	6’ b00_0111	srav $t1,$t2,$t3
	jr	6’ b00_0000	6’ b00_1000	jr $t1
	add	6’ b00_0000	6’ b10_0000	add $t1,$t2,$t3
	addu	6’ b00_0000	6’ b10_0001	addu $t1,$t2,$t3
	sub	6’ b00_0000	6’ b10_0010	sub $t1,$t2,$t3
	subu	6’ b00_0000	6’ b10_0011	subu $t1,$t2,$t3
	and	6’ b00_0000	6’ b10_0100	and $t1,$t2,$t3
	or	6’ b00_0000	6’ b10_0101	or $t1,$t2,$t3
	xor	6’ b00_0000	6’ b10_0110	xor $t1,$t2,$t3
	nor	6’ b00_0000	6’ b10_0111	nor $t1,$t2,$t3
	slt	6’ b00_0000	6’ b10_1010	slt $t1,$t2,$t3
	sltu	6’ b00_0000	6’ b10_1011	sltu $t1,$t2,$t3
I	beq	6’ b00_0100		beq $t1,$t2,label
	bne	6’ b00_0101		bne $t1,$t2,label
	lw	6’ b10_0011		lw $t1,-100($t2)
	sw	6’ b10_1011		sw $t1,-100($t2)
	addi	6’ b00_1000		addi $t1,$t2,-100
	addiu	6’ b00_1001		addiu $t1,$t2,-100
	slti	6’ b00_1010		slti $t1,$t2,-100
	sltiu	6’ b00_1011		sltiu $t1,$t2,-100
	andi	6’ b00_1100		andi $t1,$t2,100
	ori	6’ b00_1101		ori $t1, $t2,100
	xori	6’ b00_11110		xori $t1,$t2,100
	lui	6’ b00_11111		lui $t1,100
J	J	6’ b00_0010		j target

	jal	6’ b00_0011		jal target

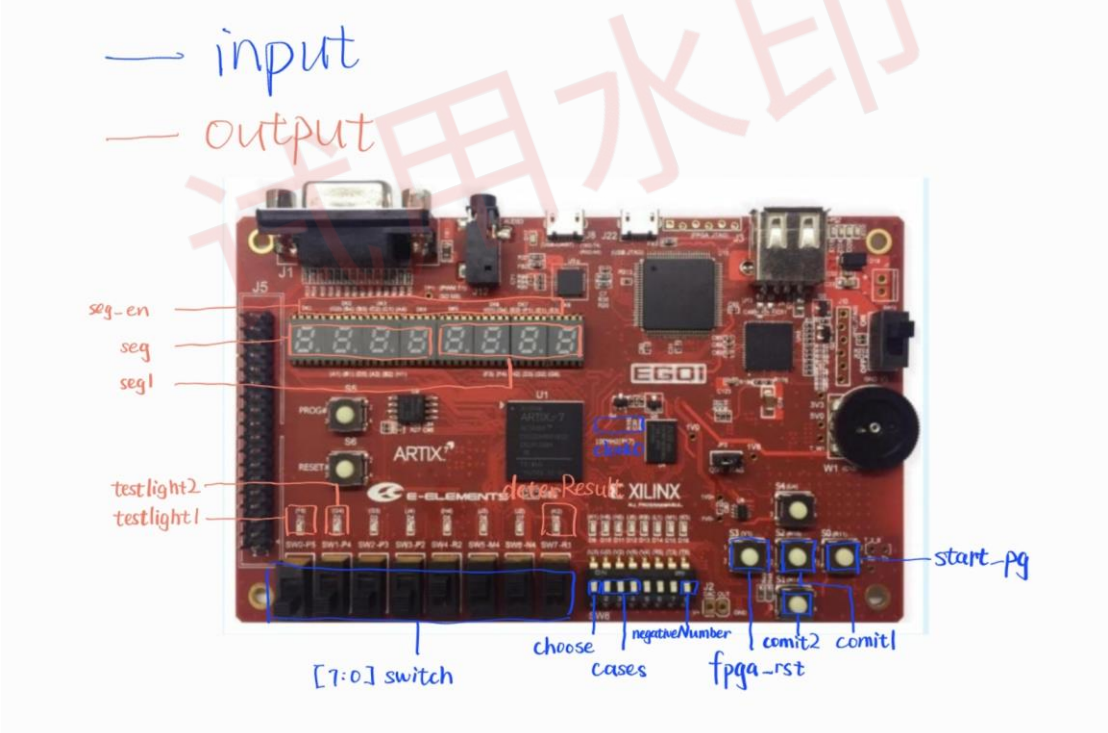
ISA 参考 Minisys 进行设计，无更新；使用 32 个 32bit 位宽的寄存器；不支持异常处理。

寻址空间设计：
属于冯·诺依曼结构。
寻址单位、指令空间、数据空间：32 位，4 字节。

对外设 IO 的支持：
采用单独的访问外设的指令（以及相应的指令）还是 MMIO（以及相关外设对应的地址），采用轮询还是中断的方式访问 IO。
通过指令 lw $t0 0x00003FFF 或 lw $t0 0x00003FFB 指令，使得 ioread 模块将外设上的数据读入，并经用 controller 模块的控制，使得 MemOrIO 模块将数据送至 decode32 模块后，将数值写入到目标寄存器中。(采用中断程序的模式)

CPU 的 CPI：
属于单周期 CPU，不支持 pipeline

CPU 接口：

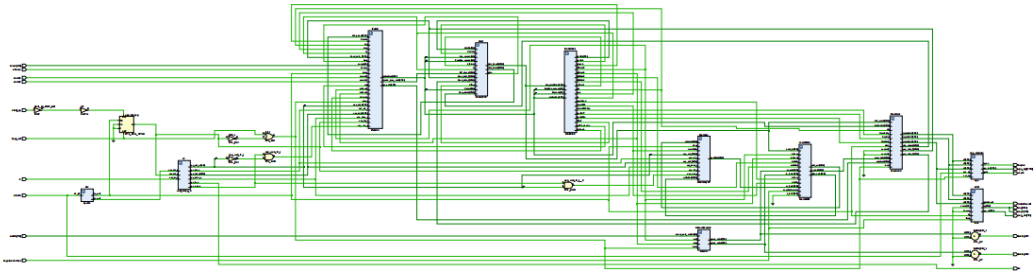


uart 接口：
输入 start_pg 为 uart 传输模式信号绑定 R11 按钮，rx 为输入数据绑定 T4 引脚；输出 tx 为输出数据绑定 N4 引脚。

其他常用 IO 接口使用说明
在软件上编写 lw $t0 0x00003FFF 或 lw $t0 0x00003FFB 指令让程序进入读取外设输入的数据状态,并将数据写入到相应的寄存器中。
vga:

```
set_property IOSTANDARD LVCMOS33 [get_ports hsync]
set_property IOSTANDARD LVCMOS33 [get_ports vsync]
```

CPU 内部结构 CPU 内部各子模块的接口连接关系图



CPU 内部子模块的设计说明（模块功能、子模块端口规格及功能说明）：
Top 模块：top.v
模块功能：顶层封装
端口规格及功能说明：
input clock0 //板子上的 100MHz 时钟
input fpga_rst //reset 信号
input [7:0] switch //开关上的数据
input [2:0] cases //三个拨码开关以选择测试用例
input choose //一个拨码开关以选择基本/进阶测试场景
input comit1 //提交 io 中的第一个数据到寄存器中
input comit2 //提交 io 中的第二个数据到寄存器中
input start_pg //uart 模式型号
input rx //uart 输入数据
input negativeNumber //表示接下来从 IO 中读取的数字为负数
output [7:0] seg_en //数码管使能端信号
output [7:0] seg //左四个数码管显示
output [7:0] seg1 //右四个数码管显示
output testLight1 //用于过程中的结果测试
output testLight2 //用于过程中的结果测试
output deterResult //用于过程中的结果测试
output hsync, vsync //vga 连接中所用的信号，详细见相应模块的注释
output[11:0] vga_rgb //vga 的 rgb 信号
output tx //uart 输出数据

IFetch 模块：Ifetc32.v
模块功能：载入指令
端口规格及功能说明：
input [31:0] Addr_result // ALU 中计算出的地址
input [31:0] Read_data_1 // jr 指令更新 PC 时用的地址
input Branch // 当 Branch 为 1 时，表示当前指令是 beq
input nBranch // 当 nBranch 为 1 时，表示当前指令为 bnq
input Jmp // 当 Jmp 为 1 时，表示当前指令为 jump

input Jal // 当 Jal 为 1 时，表示当前指令为 jal
input Jr // 当 Jr 为 1 时，表示当前指令为 jr
input Zero // 当 Zero 为 1 时，表示 ALUresult 为 0
input clock // 时钟
input reset // 复位（同步复位信号，高电平有效，当 reset=1 时，PC 赋值为 0）
input choose //配合 cases 信号用于跳转到所需进行的测试样例
input [2:0]cases //测试用例的选择
input ioRead1 //io 读取进行中的信号，会让指令跳转暂停，等待用户输入数据并提交
input ioRead2 //io 读取进行中的信号，会让指令跳转暂停，等待用户输入数据并提交
input commit1 //提交当前写入的第一个数据，并让指令跳转恢复正常
input commit2 //提交当前写入的第二个数据，并让指令跳转恢复正常
input exit ;//接收到该指令时，说明已经运行到了程序末尾，终止指令读取
input upg_rst_i // UPG 复位 (高电平有效)
input upg_clk_i // UPG 时钟 (10MHz)
input upg_wen_i // UPG 可写入
input[13:0] upg_adr_i // UPG 写入地址
input[31:0] upg_dat_i // UPG 写入数据
input upg_done_i // 完成则为 1
output [31:0] pc // 当前 PC 的值
output [31:0] Instruction // 从这个模块获取到的指令，输出到其他模块
output [31:0] branch_base_addr // 用于 beq,bne 指令，(pc+4)输出到 ALU
output reg [31:0] link_addr // 用于 jal 指令，(pc+4)输出到解码器

ALU 模块: executs32.v
模块功能：对二进制整数执行算术运算或位运算
端口规格及功能说明：
input[31:0] Read_data_1 // 从译码单元的 Read_data_1 中来
input[31:0] Read_data_2 // 从译码单元的 Read_data_2 中来
input[31:0] Sign_extend // 从译码单元来的扩展后的立即数
input[5:0] Function_opcode // 取指单元来的 r-类型指令功能码,r-form instructions[5:0]
input[5:0] Exe_opcode // 取指单元来的操作码
input[1:0] ALUOp // 来自控制单元的运算指令控制编码
input[4:0] Shamt // 来自取指单元的 instruction[10:6]，指定移位次数
input Sftmd // 来自控制单元的，表明是移位指令
input ALUSrc // 来自控制单元，表明第二个操作数是立即数（beq，bne 除外）
input I_format // 来自控制单元，表明是除 beq, bne, LW, SW 之外的 I-类型指令
input Jr // 来自控制单元，表明是 JR 指令
input[31:0] PC_plus_4 // 来自取指单元的 PC+4
output wire Zero // 为 1 表明计算值为 0
output reg [31:0] ALU_Result // 计算的数据结果
output wire [31:0] Addr_Result // 计算的地址结果

Controller 模块: control32.v
模块功能：控制其他模块运行

端口规格及功能说明：
input [5:0] Opcode //用于判断指令
input [5:0] Fuction_opcode //reg 型，用于绑定输入端口
input [31:0] Alu_resultHigh //来自于计算单元中 Alu_Result[31:0];
input [31:0] instruction //32 位指令码，用于判断是否 exit
output [1:0] ALUOp //wire 型，用于绑定输出端口
output Jr //jr 指令时为 1
output RegDST //目标寄存器为 rd 时为 1，rt 时为 0
output ALUSrc // 第二数据为立即数时为 1(除了 "beq","bne")
output MemorIOtoReg // 从内存读取写入寄存器时为 1
output RegWrite // 写入寄存器时为 1
output MemRead // 从内存读取时为 1
output MemWrite // 写入内存时为 1
output Branch // 指令为"beq" 时为 1
output nBranch // 指令为"bne"时为 1
output Jmp // 指令为"j"时为 1
output Jal // 指令为"jal"时为 1
output I_format // 指令为 I format 时为 1
output Sftmd // 移位信号
output IOWrite1 // 写入 IO
output IOWrite2 // 写入 IO
output IORead1 // 读取 IO
output IORead2 // 读取 IO
output exitProgramme // 退出程序

Memory 模块：dmemory32.v
模块功能：实现内存功能
端口规格及功能说明：
input clock //时钟
input memWrite //来自 controller. 写入内存时为 1
input [31:0] address //需要读取或写入的地址
input [31:0] writeData //需要写入的数据
input upg_rst_i // UPG 复位 (高电平有效)
input upg_clk_i // UPG 时钟 (10MHz)
input upg_wen_i // UPG 可写入
input [13:0] upg_adr_i // UPG 写入地址
input [31:0] upg_dat_i // UPG 写入数据
input upg_done_i // 完成时为 1
output[31:0] readData //从内从读出的数据

Decoder 模块：decode32.v
模块功能：对指令进行解码、实现读取数据到寄存器中
端口规格及功能说明：
input[31:0] Instruction // 取指单元来的指令

input[31:0] mem_data // 从 DATA RAM or I/O port 取出的数据
input[31:0] ALU_result // 从执行单元来的运算的结果
input Jal // 来自控制单元，说明是 JAL 指令
input RegWrite // 来自控制单元
input MemtoReg // 来自控制单元
input RegDst //用于描述目标寄存器是 rd 还是 rt
input clock //时钟
input reset // 复位
input[31:0] opcplus4 // 来自取指单元，JAL 中用
input commit1 //确认第一个输入数
input commit2 //确认第二个输入数
output reg[31:0] Sign_extend // 扩展后的 32 位立即数
output [31:0] read_data_1 // 输出的第一操作数
output [31:0] read_data_2 // 输出的第二操作数

LED 模块：leds.v
模块功能：实现 LED 输出
端口规格及功能说明：
input clk //时钟
input[2:0] cases //测试用例选择
input[0:0] choose //测试场景选择
input [31:0] a0 //$a0 寄存器内的数值
input [31:0] a1 //$a1 寄存器内的数值
input [31:0] a2 //$a2 寄存器内的数值
input flash //让流水灯闪烁起来,
output reg [7:0] seg_en //数码管使能信号
output reg [7:0] seg //数码管数值
output deterResult //某些测试用例的奇偶、大小判断等

VGA 模块：Vga.v
模块功能：实现 VGA 输出
端口规格及功能说明：
input wire vga_clk//相应的时钟周期
input wire sys_rst//复位信号
input wire [11:0] pix_data //存某点的像素信息
output reg [9:0] pix_x //x 坐标
output reg [9:0] pix_y //y 坐标
output wire[0:0] hsync //行同步信号
output wire[0:0] vsync //列同步信号
output reg [11:0] vga_rgb //rgb 色彩数值

IOread 模块：ioread.v
模块功能：实现 IO 输入
端口规格及功能说明：

Input reset //复位信号（高电平有效)
Input ior1 //从控制器来的 I/O 读取第一个数信号
Input ior2 //从控制器来的 I/O 读取第二个数信号
Input [7:0]ioread_data_switch // 从外设来的读数据，此处来自拨码开关
output reg [7:0] ioread_data1 //外设读取所得的第一个数据经处理后的结果
output reg [7:0] ioread_data2 //外设读取所得的第二个数据经处理后的结果

MemOrIO 模块：MenOrIO.v
模块功能：用于判断和甄别 decode32 接下来将读取的数据是来自 io 外设还是内存当中
端口规格及功能说明：
input mRead // 来自控制器，读取内存
input mWrite // 来自控制器，写入内存
input ioRead // 来自控制器，读取 IO
input ioWrite // 来自控制器，第一个数写入 IO
input ioWrite2 // 来自控制器，第二个数写入 IO
input[31:0] addr_in //来自 ALU 的 alu_result 接口
input[31:0] m_rdata // 从内存读取
input[7:0] io_rdata // 从 IO 读取第一个数
input[7:0] io_rdata2 // 从 IO 读取第二个数
input[31:0] r_rdata // 从解码器读取数据 data read from Decoder(register file)
input negativeNumber//将外设输入的数据由正数转为负数
output reg[31:0] r_wdata // 前往解码器的数据 data to Decoder(register file)
output[31:0] addr_out // 内存地址
output reg[31:0] write_data // 去内存或 IO 的数据（m_wdata, io_wdata）

测试说明：

模块	测试方法（仿真/上板）	测试类型（单元/集成）	测试用例描述	测试结果
IFetch	上板	集成	基本测试场景样例测试	通过
Controller	上板	集成	基本测试场景样例测试	通过
ALU	上板	集成	基本测试场景样例测试	通过
memOrIO	上板	集成	基本测试场景样例测试	通过
Memory	上板	集成	基本测试场景样例测试	通过
loread	上板	集成	基本测试场景样例测试	通过
Decoder	上板	集成	基本测试场景样例测试	通过
LED	上板	集成	基本测试场景样例测试	通过
VGA	上板	集成	基本测试场景样例测试	通过
IFetch	仿真	单元	OJ测试	通过
Controller	仿真	单元	OJ测试	通过
ALU	仿真	单元	OJ测试	通过
Decoder	仿真	单元	OJ测试	通过
Memory	仿真	单元	OJ测试	通过

测试结论：所有模块均通过测试。

问题及总结：
问题：
1、由于开发初期对于 cpu 中的各个模块间的关系并没有理解透彻，导致在小组成员拼接 cpu 的基础模块时，没能一次正确的将顶层模块的线路连接正确。这导致 cpu 无法正常工作。

相关思考：在开发接口关系复杂、模块多且相互关系紧密的模块时，应当对各个模块间的关系理解透彻，并规范命名各个 input、output、wire 与 reg，避免因误解导致线路连接错误。
2、没能形成软硬件协同的思想。开发中，我们遇到了很多在硬件层面难以实现的问题，如乘法等问题，没能及时想到可以通过更改 MIPS 代码的程序简单的满足需 求。这导致小组内花费了超额的时间投入在硬件的开发中。
相关思考：在做硬件开发的过程中，应当更加深入的去思考需求，并尝试从软硬件两方面思考物体，而不是拘泥于其中一种形式解决问题。
3、设计硬件功能如 IO 读入外设数据时，由于并未明确的理解需求，导致早期设计中，我们的 IO 为了支持可以读入两个数字的需求，采用了两排的按钮分别作为数据读入所需的外设，进而导致我们的按钮不够用。最后，只能撤销大量已成的代码，耗费了大量无意义的时间和精力。
相关思考：没能深入理解需求和自己的后续开发困难前，不应当鲁莽的展开工作。

总结：
1、有效的开发应当建立深入的理解需求和各个模块的功能的基础之上。
2、在分配按钮开关等外设设备的使用时，不应仅仅为当下考虑，而应深入思考之后可能的需求后，做出合理的规划。
3、做硬件时，要多想软件的需求和功能；做软件时，要多想硬件的功能和局限性。

Bonus 部分

vga 的实现
通过 vga 模块中的行列同步信号，扫描整个屏幕，并依据 vga_draw 模块中的颜色数值对屏幕进行绘制。相应的分辨率为 640x480.
vga_control 模块用于控制屏幕信号的连接和刷新、vga_draw 模块负责依照情况提供所需的像素点染色数值、Get25Clk 提供 vga 连接所需的 25MHz 的时钟信号
连接关系：

```
module vga(//clk 是25MHZ
input wire sys_clk,//10oMhz时钟
input wire sys_rst,//reset

input [31:0] a0,
input [31:0] a1,
input [31:0] a2,

output wire hsync,
output wire vsync,
output wire [11:0]vga_rgb
);
wire [9:0] pix_x;
wire [9:0] pix_y;
wire [11:0] pix_data;
wire clk_25m;

vga_control vc(
clk_25m , sys_rst , pix_data , pix_x , pix_y , hsync, vsync , vga_rgb
);

vga_draw vd(
clk_25m , sys_rst , a0, a1, a2 , pix_x , pix_y , pix_data
 );

Get25Clk Gc(
sys_clk ,sys_rst ,clk_25m
);

endmodule
```

核心代码：

```
module vga_control(
input wire vga_clk,
input wire sys_rst,
input wire [11:0] pix_data,//存某点的像素信息
output reg [9:0] pix_x,
output reg [9:0] pix_y,
output wire[0:0] hsync,
output wire[0:0] vsync,
output reg [11:0] vga_rgb
);

parameter H_SYNC =10'd96; //行同步周期
parameter H_BA=10'd48;
parameter H_VA = 10'd640; //合法显示部分
parameter H_TO=10'd800;

parameter V_SYNC =10'd2; //场同步周期
parameter V_BA=10'd33;
parameter V_VA = 10'd480; //合法显示部分
parameter V_TO=10'd525;

reg[9:0] cnt_h=10'b00000_00001;
reg[9:0] cnt_v=10'b00000_00001;
```

```
always@(posedge vga_clk,posedge sys_rst)begin
if(sys_rst==1'b1)begin
  cnt_v<=10'b00000_00001;
  end
  else if( (cnt_h==H_TO)&& (cnt_v < V_TO ) ) begin
    cnt_v<=cnt_v+10'b00000_00001;
  end
  else if( (cnt_h ==H_TO) &&(cnt_v == V_TO ) ) begin
    cnt_v<=10'b00000_00001;
  end
  else begin
  cnt_v <=cnt_v;
  end
end

always@(posedge vga_clk,posedge sys_rst)begin
   if(sys_rst==1'b1)begin
cnt_h<=0;
end
else if(cnt_h ==(H_TO) ) begin
  cnt_h<=10'b00000_00001;
end
else begin
cnt_h<=cnt_h+10'b00000_00001;
end
end


  always@(posedge  vga_clk)begin
      if( (cnt_h >=H_SYNC +H_BA-1'b1)&& (cnt_h <H_SYNC +H_BA + H_VA-1'b1) &&
        (cnt_v >= V_SYNC +V_BA ) && (cnt_v <V_SYNC +V_BA + V_VA)  )begin
          pix_x <=cnt_h -(H_SYNC +H_BA ) ;
          pix_y <=cnt_v-(V_SYNC +V_BA );
        end
        else begin
          pix_x<=10'b00000_00000;
          pix_y<=10'b00000_00000;
        end
   end

     always@(posedge  vga_clk)begin
          if( (cnt_h >=H_SYNC +H_BA)&& (cnt_h <H_SYNC +H_BA + H_VA) &&
            (cnt_v >= V_SYNC +V_BA ) && (cnt_v <V_SYNC +V_BA + V_VA)  )begin
             vga_rgb<=pix_data;
            end
            else begin
              vga_rgb<=12'b0000_0000_0000;
            end
        end

   assign hsync =(cnt_h <=H_SYNC  ? 1'b0:1'b1);
   assign vsync =(cnt_v <= V_SYNC  ? 1'b0 :1'b1);
```

uart 的实现

使用方法

将编写好的 asm 文件利用 Mars4_4 及本小组自行编写的修改脚本生成 coe 文件，再使用 GenUBit_Minisys3.0 和 UartAssist 传入 FPGA 板。

设计思路

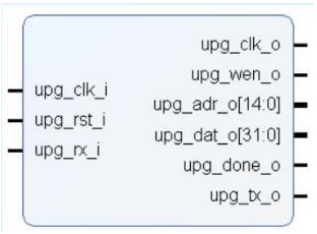
核心思想为 cpu 添加一个新模式 Uart Communication mode 用于接受 uart 信号。

添加了一个新的方法 Uart_bmgp_0 作为 uart 接口。

与周边模块的关系

uart 模块输出的数据会进入 IFetch 模块与 Memory 模块。如果处于 Uart Communication mode，这两个模块就会选用 uart 作为数据源进行操作。

核心代码及说明



该图为新创建的作为 uart 接口的 IP core。

```
wire kickOff = upg_rst_i | (~upg_rst_i & upg_done_i);
RAM ram (
.clka (kickOff ? clk : upg_clk_i),
.wea (kickOff ? memWrite : upg_wen_i),
.addra (kickOff ? address[15:2] : upg_adr_i),
.dina (kickOff ? writeData : upg_dat_i),
.douta (readData)
);
```

```
wire kickOff = upg_rst_i | (~upg_rst_i & upg_done_i );
prgrom instmem (
.clka (kickOff ? clock : upg_clk_i ),
.wea (kickOff ? 1'b0 : upg_wen_i ),
.addra (kickOff ? PC[15:2] : upg_adr_i ),
.dina (kickOff ? 32'h00000000 : upg_dat_i ),
.douta (Instruction)    );
```

这两张图片是 uart 接口收到数据后在其他模块中的后续处理。

更好的用户体验

1、七段数码管交替显示$a0,$a1,$a2 三个寄存器的值，代码中只需将输出添加至这三个寄存器中便可清晰知道结果。

核心代码及说明：

```
always @ * begin
    case (turn)
        2'b00: a <= a0;
        2'b01: a <= a1;
        2'b10: a <= a2;
    endcase
    case (cnt_ms/ms)
        3'b000: out <= a[31:28];
        3'b001: out <= a[27:24];
        3'b010: out <= a[23:20];
        3'b011: out <= a[19:16];
        3'b100: out <= a[15:12];
        3'b101: out <= a[11:8];
        3'b110: out <= a[7:4];
        3'b111: out <= a[3:0];
    endcase

end
```

```
always @ (posedge clk, posedge rst) begin
    if(rst) begin
        cnt<=zero;
        turn<=2'b00;
        cnt_ms<=18'd0;
    end
    else begin
        if(cnt == five_s) begin
            cnt <= zero;
            if (turn == 2'b10) turn <= 2'b00;
            else turn <= turn + 1'b1;
        end
        else cnt  <= cnt +1'b1;
        if (cnt_ms == ms_8) cnt_ms <= 18'd0;
        else cnt_ms <= cnt_ms + 1'b1;
    end

end
```

（每五秒进行 turn 转换，依据 turn 来确定输出哪个寄存器的值）

2、引入一个“negativeNumber”开关，由于手动转补码较困难，通过这个开关，在 MemOrIO 加以判断，主动转补码，让用户输负数更方便。

相应的核心代码：

```
//执行lW指令，将io或者Mem中的数据准备写入寄存器中
always @(*) begin
    if(mRead == 1'b1) begin
      r_wdata <= m_rdata;
    end
    else if(ioRead == 1'b1&&negativeNumber==0) begin
     r_wdata <= {{24{1'b0}}, io_rdata}; //无符号拓展
    end
    else if(ioRead ==1'b1 &&negativeNumber==1)begin
      r_wdata<={{24{1'b1}},temp_io_rdata};
    end
    else if(ioRead2==1'b1&&negativeNumber==0)begin
     r_wdata<= {{24{1'b0}}, io_rdata2};
    end
     else if(ioRead2==1'b1&&negativeNumber==1)begin
          r_wdata<= {{24{1'b1}}, temp_io_rdata2};
        end

end
```

（在 MemOrIO 模块下，会讲发送给 decode32 模块中的数据取补码）

测试结果

模块	测试方法（仿真/上板）	测试类型（单元/集成）	测试用例描述	测试结果
uart	上板	集成	通过编写新的asm文件，观察cpu是否读入更新后文件	通过
vga	上板	集成	基本测试场景样例，观察vga显示与正确输出是否一致	通过

测试结论：所有模块均通过测试。

问题与总结：

Vga：

问题：

1、由于在一开始开发时没能深入理解 vga 的扫描原理，导致对行列同步周期的变化理解不够深入，导致没能使得 vga 连接成功。之后，通过翻阅更多相关资料，明确了其详细的原理后才得以解决

2、一开始想显示更多数据，创建的 reg 过多，超出开发版承受范围，后来对显示内容做了删减

总结：在编写不熟悉的特殊功能的设备的代码时，应当对其相应的使用需求和实现原理有较为深入的提前知识储备后在进行编写代码的工作。

uart：

问题：

编写 asm 文件时用了部分非基础指令导致使用 Mars4_4 生成 coe 文件失败。后来通过修改 asm 文件及编写的 java 脚本文件解决了问题。

总结：

在进行工作前应先确认所有要求，避免因遗漏要求造成无效作业。