**同济大学**

**计算机科学与技术系**

**计算机组成原理课程实验报告**

****

**学 号 1652228**

**姓 名 王哲源**

**专 业 计算机科学与技术**

**授课老师 陈永生**

**日 期 2018.6.3**

1. 实验目标

使用Verilog HDL语言实现54条MIPS指令的单周期CPU设计与仿真

并实现CPU下板，同时在该CPU上实现简单的MIPS程序

1. 总体设计
2. 设计思路：

首先，外部存储器需要与cpu原件分开，指令的读取在周期开始的上升沿执行，而数据的写入在同周期的下降沿执行

cpu部件内包含ALU模块，乘除法器，pc寄存器，寄存器堆，CP0，控制器六大部分。

寄存器堆的写入同外部储存器一样，在时钟下降沿执行，而读取则为在获取地址时随时进行。

pc寄存器为时序逻辑部件，用于存储上一周期指令执行完毕下一条指令的地址，在时钟上升沿到来时赋值给pc，并从外部存储器指令段获取指令。同时其会有一个使能端，仅有使能端允许时才会对pc进行赋值，这是受除法器所限制的功能。

指令从外部存储器获得后会送入控制器，控制器内为组合逻辑，不受时钟边沿控制，它会将得到的inst进行拆分，无论当前指令为何类型，它都会无条件的将指令所指向的内存地址送入寄存器堆中获取数据，对于获得的数据也会无条件送入如ALU、乘除法器中进行准备运算。在取数的同时控制器也会对于指令进行相应的分析，并且赋予指令所需要执行的原件相应的使能信号，在获得使能信号后相应原件进行运算/取数再回传至控制器，再由控制器对获得的数据进行统一写入/存储。上述各操作除乘法器外均不受时序逻辑控制，在pc未改变前只要时钟周期允许，各端口将形成一条数据通路实现数据的获取、运算及转移。

CP0与ALU、乘除法器设计思路已在控制器中介绍，唯一特殊的是乘除法器中的除法器，由于除法的特殊性导致除法器无法在一个时钟周期内运算得到结果，因此必须为除法器提供特殊的pc暂时中断功能，即前述的pc寄存器使能端口，用以保证程序能够在运算结束前不执行下一条指令。

CPU实现后，使用上学期编写的VGA模块访问内存中指定模块，使用MIPS指令将内部寄存器的数据搬至内存对应段，实现输出；输入则为由板上按键输入至内存指定部分，并通过指令移至寄存器内参与运算

1. CPU的数据通路图（注：MDU模块中包含32位的带符号与无符号乘除法运算，但由于数据通路中不便描述，因此不在此画出）：



1. 模块建模
2. defines模块，用于对cpu所需要使用的各种宏进行定义

//-------------------- 31\_base--------------------

//----------Common Part----------

`define Enable 1'b1

`define Disable 1'b0

`define True 1

`define False 0

`define RstEnable 1'b1

`define RstDisable 1'b0

`define ReadEnable 1'b1

`define ReadDisable 1'b0

`define WriteEnable 1'b1

`define WriteDisable 1'b0

`define ZeroWord 32'h00000000

//----------Instruction Part----------

`define InstAddrBus 31:0

`define InstBus 31:0

`define OpBus 5:0

`define DefaultInstAddr 32'h00400000

//----------Register Part----------

`define RegAddrBus 4:0

`define RegBus 31:0

`define DRegBus 63:0

`define HigherBus 63:32

`define LowerBus 31:0

`define HalfRegBus 15:0

`define RegSize 32

`define RegAddrWidth 5

`define FailRegAddr 5'b00000

//----------ALU Part----------

`define ALUCtrlBus 3:0

`define AnsBus 32:0

`define ALU\_addu 4'b0000

`define ALU\_add 4'b0010

`define ALU\_subu 4'b0001

`define ALU\_sub 4'b0011

`define ALU\_and 4'b0100

`define ALU\_or 4'b0101

`define ALU\_xor 4'b0110

`define ALU\_nor 4'b0111

`define ALU\_lui 4'b100x

`define ALU\_slt 4'b1011

`define ALU\_sltu 4'b1010

`define ALU\_sra 4'b1100

`define ALU\_sll 4'b111x

`define ALU\_srl 4'b1101

//----------Memory Part----------

`define MemSave 1'b1

`define MemLoad 1'b0

`define MemAddrBus 31:0

`define MemBus 31:0

`define DefaultDataAddr 32'h10010000

//----------Operation Part----------

`define No\_op 6'b111111

`define No\_func 6'b111111

`define R\_type\_op 6'b000000

`define add\_func 6'b100000

`define addu\_func 6'b100001

`define sub\_func 6'b100010

`define subu\_func 6'b100011

`define and\_func 6'b100100

`define or\_func 6'b100101

`define xor\_func 6'b100110

`define nor\_func 6'b100111

`define slt\_func 6'b101010

`define sltu\_func 6'b101011

`define sll\_func 6'b000000

`define srl\_func 6'b000010

`define sra\_func 6'b000011

`define sllv\_func 6'b000100

`define srlv\_func 6'b000110

`define srav\_func 6'b000111

`define jr\_func 6'b001000

`define addi\_op 6'b001000

`define addiu\_op 6'b001001

`define andi\_op 6'b001100

`define ori\_op 6'b001101

`define xori\_op 6'b001110

`define lw\_op 6'b100011

`define sw\_op 6'b101011

`define beq\_op 6'b000100

`define bne\_op 6'b000101

`define slti\_op 6'b001010

`define sltiu\_op 6'b001011

`define lui\_op 6'b001111

`define j\_op 6'b000010

`define jal\_op 6'b000011

//-------------------- 23\_extend --------------------

//----------Extend Operation Part----------

`define lb\_op 6'b100000

`define lbu\_op 6'b100100

`define lh\_op 6'b100001

`define lhu\_op 6'b100101

`define sb\_op 6'b101000

`define sh\_op 6'b101001

//op == ex\_op1

`define ex\_op1 6'b000000

`define jalr\_func 6'b001001

`define syscall\_func 6'b001100

`define teq\_func 6'b110100

`define break\_func 6'b001101

`define multu\_func 6'b011001

`define div\_func 6'b011010

`define divu\_func 6'b011011

`define mthi\_func 6'b010001

`define mtlo\_func 6'b010011

`define mfhi\_func 6'b010000

`define mflo\_func 6'b010010

`define ex\_op2 6'b011100

`define clz\_func 6'b100000

`define mul\_func 6'b000010

`define CP0\_op 6'b010000

`define eret\_func 6'b011000

`define mfc0\_rs 5'b00000

`define mtc0\_rs 5'b00100

`define bgez\_op 6'b000001

//----------CP0 Part----------

`define CP0\_size 32

`define CP0AddrBus 4:0

`define status\_i 12

`define cause\_i 13

`define epc\_i 14

`define IE 0

`define DefaultErrAddr 32'h00400004

`define SYSCALL\_ERR 4'b1000

`define BREAK\_ERR 4'b1001

`define TEQ\_ERR 4'b1101

//----------MDU Part----------

`define MDUCtrlBus 2:0

`define MDU\_default 3'h0

`define MDU\_multu 3'h2

`define MDU\_div 3'h3

`define MDU\_divu 3'h4

`define MDU\_mthi 3'h5

`define MDU\_mtlo 3'h6

1. sscomp\_dataflow模块，用于将cpu部件与数据寄存器和指令寄存器连接起来，并为cpu检测与使用提供相应的接口

`include "defines.vh"

module sccomp\_dataflow(

input clk\_in,

input reset,

output [`InstBus] inst,

output [`InstAddrBus] pc,

output [`MemAddrBus] addr

);

1. DMEM模块，用于存储数据。同时新增两个接口，用于与外部进行数据的交换

`include "defines.vh"

module DMEM(

input clk,

input wena,

input [`MemAddrBus] addr,

input [`MemBus] idata,

output [`MemBus] odata,

input [191:0] wdata,

output [159:0] rdata

);

1. cpu模块，作为cpu内部各组件的顶层文件，对cpu各组件之间实现连线，并提供与外部存储器连接的接口。同时新增两个接口，用于与外部进行数据的交换

`include "defines.vh"

module cpu(

input clk\_in,

input reset,

input [`InstBus] Instruction,

output [`InstAddrBus] pc,

output ram\_ena,

output [`MemAddrBus] ram\_addr,

input [`MemBus] ram\_rdata,

output [`MemBus] ram\_wdata,

input [191:0] wdata,

output [159:0] rdata

);

1. PCReg模块，在时钟上升沿到来时将前一条指令执行完毕后所跳转的下一条指令地址赋给PC

`include "defines.vh"

module PCReg(

input clk,

input rst,

input ena,

input [`InstAddrBus] pc\_in,

output reg [`InstAddrBus] pc

);

1. regfile模块，cpu内部的寄存器堆，支持在时钟下降沿到来时写入并随时读取

`include "defines.vh"

module regfile(

input clk,

input rst,

//----write\_port-----

input we,

input [`RegAddrBus] waddr,

input [`RegBus] wdata,

//-----read\_port-----

input [`RegAddrBus] raddr1,

output [`RegBus] rdata1,

input [`RegAddrBus] raddr2,

output [`RegBus] rdata2

);

1. ALU模块，根据传入的aluc值输出相应计算的结果

`include "defines.vh"

module ALU(

input [`ALUCtrlBus] aluc,

input [`RegBus] a,

input [`RegBus] b,

output [`RegBus] r,

output zero,

output carry,

output negative,

output overflow

);

1. MDU模块，作为乘法器与除法器的顶层文件控制并保存乘法与除法的结果，同时能够在运算时暂时中断对下一条指令的获取

`include "defines.vh"

module MDU(

input clk,

input rst,

input [`MDUCtrlBus] mduc,

input [`RegBus] a,

input [`RegBus] b,

output [`DRegBus] mul\_result,

output reg [`RegBus] hi,

output reg [`RegBus] lo,

output reg pc\_ena

);

1. MULT与MULTU模块，分别进行有符号与无符号的乘法运算

9.1 MULT模块

`include "defines.vh"

module MULT(

input reset,

input [`RegBus] a,

input [`RegBus] b,

output [`DRegBus] z

);

9.2 MULTU模块

`include "defines.vh"

module MULTU(

input reset,

input [`RegBus] a,

input [`RegBus] b,

output [`DRegBus] z

);

1. DIV与DIVU模块，分别支持带符号与不带符号的带余数除法

10.1 DIV模块

`include "defines.vh"

module DIV(

input [31:0]dividend,

input [31:0]divisor,

input start,

input clock,

input reset,

output [31:0]q,

output [31:0]r,

output reg busy

);

10.2 DIVU模块

`include "defines.vh"

module DIVU(

input [31:0]dividend,

input [31:0]divisor,

input start,

input clock,

input reset,

output [31:0]q,

output [31:0]r,

output reg busy

);

1. CP0模块，为cpu提供协处理功能，以支持异常中断指令

`include "defines.vh"

module CP0(

input clk,

input rst,

input mtc0,

input [`InstAddrBus] pc,

input [`CP0AddrBus] addr,

input [`RegBus] wdata, // data from GP register

input eret,

input teq\_exc,

input [3:0] cause,

output [`RegBus] rdata, // data for GP register

output [`InstAddrBus] exc\_addr

);

1. CtrlUnit模块，cpu组件的核心模块，同时兼具指令译码及数据选择功能的总控制模块，可以对指令进行译码从其他模块传输来的数据中选取需要的数据，也可以对于运算结果或读取结果根据指令译码结果进行选择，并送入制定的寄存器或存储器中

`include "defines.vh"

module CtrlUnit(

input [`InstBus] inst,

input [`InstAddrBus] pc,

output reg [`InstAddrBus] pc\_out,

output [`RegAddrBus] rs,

output [`RegAddrBus] rt,

input [`RegBus] rdata1,

input [`RegBus] rdata2,

output reg [`ALUCtrlBus] aluc,

output reg [`RegBus] ALU\_a,

output reg [`RegBus] ALU\_b,

input [`MemBus] ram\_rdata,

input [`RegBus] alu\_result,

output reg reg\_ena,

output [`RegAddrBus] waddr,

output reg [`RegBus] wdata,

output ram\_ena,

output [`MemAddrBus] ram\_addr,

output reg [`MemBus] ram\_wdata,

//----------extend\_op port----------

input [`RegBus] cp0\_rdata,

input [`InstAddrBus] exc\_addr,

output mtc0,

output eret,

output teq\_exc,

output reg [3:0] cause,

output [`CP0AddrBus] rd, // CP0 addr

input [`RegBus] mul\_result,

input [`RegBus] hi,

input [`RegBus] lo,

output reg [`MDUCtrlBus] mdu\_op

);

1. top\_file顶层模块，用于将CPU与VGA设备连接起来

module top\_file(

//-----common port-----

input clk,

input rst,

//-----input port-----

input start\_press,

input input\_press,

input [15:0] input\_num,

//-----VGA port-----

output [3:0] R,

output [3:0] G,

output [3:0] B,

output HS,

output VS

);

1. VGA模块，用于在VGA上打印结果

module VGA\_driver(

input CLK,

input RST,

input [2:0] Q,

input [159:0] reg1to5,

output reg[3:0] R,

output reg[3:0] G,

output reg[3:0] B,

output HS,

output VS

);

1. 整倍数分频器

module Divider

#(

parameter mod = 32'd20

)

(

input I\_CLK,

input RST,

output reg O\_CLK

);

1. CPU测试/调试过程

（※由于VGA在上学期已经通过测试，因此这里不再列出）

各组件对应testbench如下所示：

1. MULT/MULTU组件testbench：
   1. MULT组件testbench

module MULT\_tb;

reg clk = 0;

reg reset = 0;

reg signed [31:0] a,b;

wire signed [63:0] c;

MULT uut(.clk(clk), .reset(reset),

.a(a), .b(b), .z(c));

always #5 clk = ~clk;

initial

begin

#13 reset <= 'b1; a <= 'd3; b <= 'd2;

#20 a <= 'd5; b <= 'h80000000;

#20 a <= -5;

#20 a <= 'd0;

#30 reset <= 'b0;

end

endmodule

* 1. MULTU组件testbench

module MULTU\_tb;

reg clk = 0;

reg reset = 0;

reg [31:0] a,b;

wire [63:0] c;

MULTU uut(.clk(clk), .reset(reset),

.a(a), .b(b), .z(c));

always #5 clk = ~clk;

initial

begin

#13 reset <= 'b1; a <= 'd2; b <= 'd3;

#10 a <= 'd2200000000; b <= 'd2200000000;

#30 reset <= 'b0;

end

endmodule

1. DIV/DIVU组件testbench：

2.1 DIV组件testbench

module DIV\_tb;

reg clk = 1, rst, st;

reg [31:0] dividend, divisor;

wire bsy;

wire [31:0] q, r;

DIV uut(.clock(clk), .reset(rst), .start(st),

.dividend(dividend), .divisor(divisor),

.busy(bsy),

.q(q), .r(r));

always #5 clk = ~clk;

initial

begin

rst <= 0; st <= 1;

#3

rst <= 0;

dividend <= 7;

divisor <= -2;

#6 st <= 0;

#400 st <= 1;

#5

st <= 0;

dividend <= -7;

divisor <= -2;

end

endmodule

2.2 DIVU组件testbench

module DIVU\_tb;

reg clk = 1, rst, st;

reg [31:0] dividend, divisor;

wire bsy;

wire [31:0] q, r;

DIVU uut(.clock(clk), .reset(rst), .start(st),

.dividend(dividend), .divisor(divisor),

.busy(bsy),

.q(q), .r(r));

always #5 clk = ~clk;

initial

begin

rst <= 0; st <= 1;

#3

rst <= 0;

dividend <= 32'd7;

divisor <= 32'd2;

#6 st <= 0;

#400 st <= 1;

#5

st <= 0;

dividend <= 32'hffffffff;

divisor <= 32'h55555555;

end

endmodule

1. 主程序testbench：

`include "defines.vh"

module cpu\_tb;

reg clk = 0;

reg rst;

wire [`InstBus] inst;

wire [`InstAddrBus] \_pc;

wire [`MemAddrBus] addr;

wire [`InstAddrBus] pc = \_pc - `DefaultInstAddr;

integer file\_output;

integer counter;

initial

begin

file\_output = $fopen("regs.txt");

rst = 1; counter = 0;

#50 rst = 0;

end

always

begin

#40 clk = ~clk;

#10

if(clk == 1'b0)

begin

if(counter == 2049)

begin

$fclose(file\_output);

end

else

begin

counter = counter + 1;

$fdisplay(file\_output, "pc: %h", pc);

$fdisplay(file\_output, "instr: %h", cpu\_tb.uut.inst);

$fdisplay(file\_output, "regfile0: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[0]);

$fdisplay(file\_output, "regfile1: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[1]);

$fdisplay(file\_output, "regfile2: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[2]);

$fdisplay(file\_output, "regfile3: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[3]);

$fdisplay(file\_output, "regfile4: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[4]);

$fdisplay(file\_output, "regfile5: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[5]);

$fdisplay(file\_output, "regfile6: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[6]);

$fdisplay(file\_output, "regfile7: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[7]);

$fdisplay(file\_output, "regfile8: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[8]);

$fdisplay(file\_output, "regfile9: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[9]);

$fdisplay(file\_output, "regfile10: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[10]);

$fdisplay(file\_output, "regfile11: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[11]);

$fdisplay(file\_output, "regfile12: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[12]);

$fdisplay(file\_output, "regfile13: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[13]);

$fdisplay(file\_output, "regfile14: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[14]);

$fdisplay(file\_output, "regfile15: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[15]);

$fdisplay(file\_output, "regfile16: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[16]);

$fdisplay(file\_output, "regfile17: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[17]);

$fdisplay(file\_output, "regfile18: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[18]);

$fdisplay(file\_output, "regfile19: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[19]);

$fdisplay(file\_output, "regfile20: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[20]);

$fdisplay(file\_output, "regfile21: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[21]);

$fdisplay(file\_output, "regfile22: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[22]);

$fdisplay(file\_output, "regfile23: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[23]);

$fdisplay(file\_output, "regfile24: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[24]);

$fdisplay(file\_output, "regfile25: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[25]);

$fdisplay(file\_output, "regfile26: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[26]);

$fdisplay(file\_output, "regfile27: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[27]);

$fdisplay(file\_output, "regfile28: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[28]);

$fdisplay(file\_output, "regfile29: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[29]);

$fdisplay(file\_output, "regfile30: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[30]);

$fdisplay(file\_output, "regfile31: %h", cpu\_tb.uut.sccpu.cpu\_ref.array\_reg[31]);

end

end

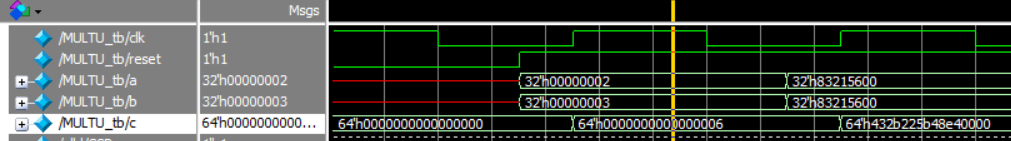
end

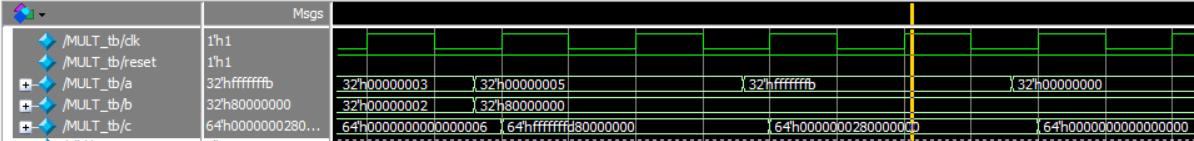
sccomp\_dataflow uut(.clk\_in(clk), .reset(rst),

.inst(inst), .pc(\_pc), .addr(addr));

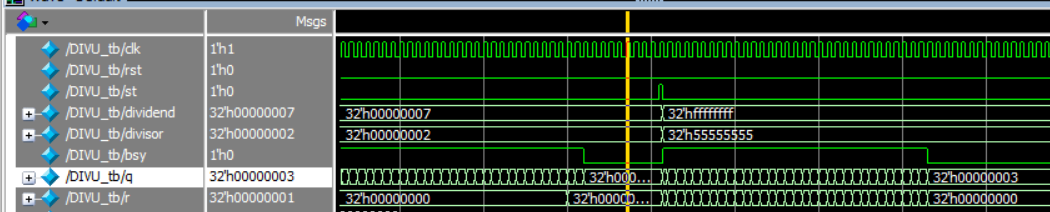
endmodule

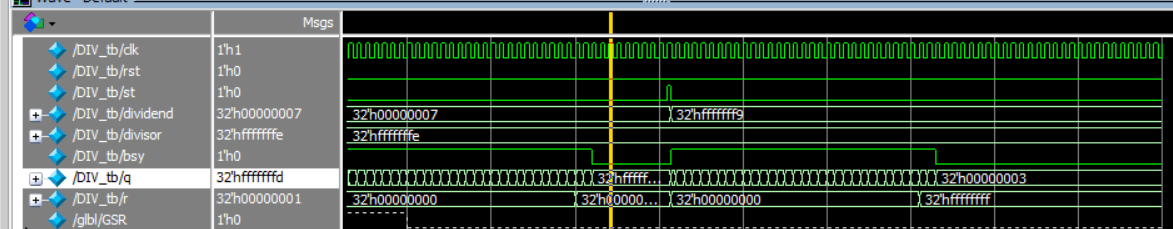
1. CPU测试结果分析
2. MULT/MULTU测试结果（modelsim仿真波形图）





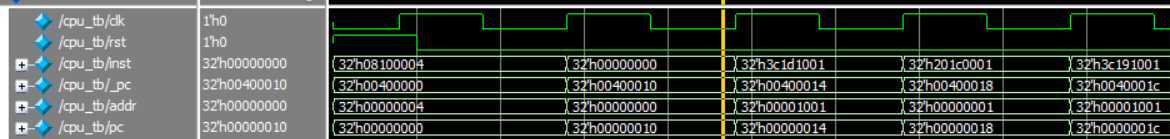
1. DIV/DIVU测试结果（modelsim仿真波形图）





1. 主程序测试结果（由于指令条数过多，此处仅能展示部分结果）

modelsim仿真波形图



输出文件比对：通过coe文件利用testbench生成reg.txt文件，并与mars执行对应指令得到的结果利用VSCode进行文件比对

1. 结论

该CPU各部件编写与连接正确，能够顺利在周期为80ns的情况下通过前仿真测试，并且能够正确执行54条MIPS指令并得到期望的结果

1. 相关MIPS指令编写

该MIPS指令完成对于5个数的排序

start:

j loop\_start

loop:

lw $3, 100

lw $4, 104

lw $5, 108

lw $6, 112

lw $7, 116

lw $31, 120

beq $31, 0x00000000, reloop

j sort\_start

reloop:

j loop

loop\_start:

addi $2, $0, 3

mtc0 $2, $12

syscall

sort\_start:

sub $8, $3, $4

bgez $8, skip12

addi $9, $3, 0

addi $3, $4, 0

addi $4, $9, 0

skip12:

sub $8, $3, $5

bgez $8, skip13

addi $9, $3, 0

addi $3, $5, 0

addi $5, $9, 0

skip13:

sub $8, $3, $6

bgez $8, skip14

addi $9, $3, 0

addi $3, $6, 0

addi $6, $9, 0

skip14:

sub $8, $3, $7

bgez $8, skip15

addi $9, $3, 0

addi $3, $7, 0

addi $7, $9, 0

skip15:

sub $8, $4, $5

bgez $8, skip23

addi $9, $4, 0

addi $4, $5, 0

addi $5, $9, 0

skip23:

sub $8, $4, $6

bgez $8, skip24

addi $9, $4, 0

addi $4, $6, 0

addi $6, $9, 0

skip24:

sub $8, $4, $7

bgez $8, skip25

addi $9, $4, 0

addi $4, $7, 0

addi $7, $9, 0

skip25:

sub $8, $5, $6

bgez $8, skip34

addi $9, $5, 0

addi $5, $6, 0

addi $6, $9, 0

skip34:

sub $8, $5, $7

bgez $8, skip35

addi $9, $5, 0

addi $5, $7, 0

addi $7, $9, 0

skip35:

sub $8, $6, $7

bgez $8, skip45

addi $9, $6, 0

addi $6, $7, 0

addi $7, $9, 0

skip45:

addi $31, $0, 0x00000000

sw $3, 100

sw $4, 104

sw $5, 108

sw $6, 112

sw $7, 116

sw $31, 120

eret

1. 实际下板情况

待输入状态：上部显示蓝色，通过底部开关决定输入二进制，通过按键确定输入

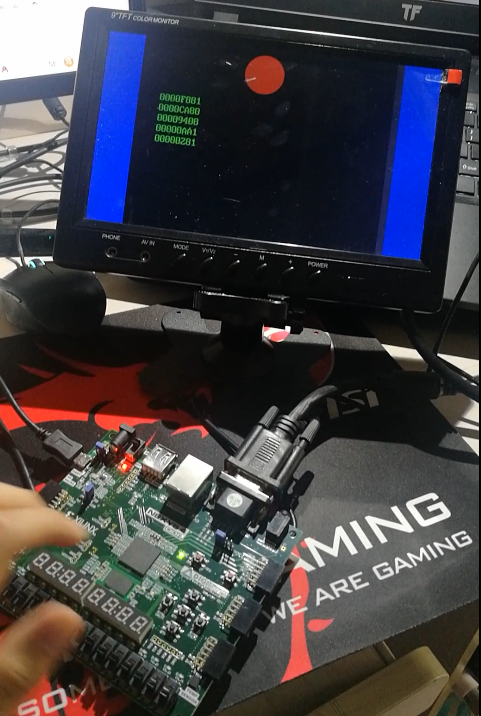


可开始状态，输入5个之后此时显示黄色，需要按下开始键执行排序

（注：此时仍可以对最后一个数进行改变）



结束状态，此时排序结束，显示红色，按reset重置



1. 心得体会及建议

**心得体会：**

大型程序的分块编写有助于编写效率和正确性的提高。

本次实验利用了上学期所编写的regfile组件与ALU组件，以及前期编写的MULT/MULTU/DIV/DIVU/CP0组件，由于这些组件在之前已经通过仿真验证其正确性，因此本次cpu的编写时只需要另外编写其他部分，再将这些组件与之进行连接组合起来即可。在查错时也可以跳过这些组件，缩小了查错的工作量，提高了编写效率。同时，由于之前已经先行对31条基础MIPS指令进行过编写与测试，并预留了剩余23条扩展指令所需的接口，因此本次编写相较于上一次的效率有显著的提高。

另一方面，宏定义与注释对于大型程序的阅读有一定的帮助。由于距离31条指令的编写已经有一段时间，因此对于一些代码段的具体功能已有些遗忘，而宏定义与注释及程序分块能够有效帮助对于程序段功能的回想，对于新功能及代码的添加有了一定的帮助

当然，本次的cpu程序编写不免有一些失误，比如对于常量进行宏定义后在条件编写时忘记与端口进行比较而是直接将其作为条件防止，导致了不小的查错工作量。而由于使用了第三方编辑器的缘故，其自带的代码补全也使得在宏定义文件的编写时出现了差错的问题，最终导致花费大量时间在该问题的查错上实在不应该。

因此本次编写一方面是对于cpu运作方式理解的更好掌握，更是对于程序编写经验的宝贵积累

**建议：**

由于本次实验提供的测试coe文件为十六进制文件，其对应的mars指令和操作数只能通过手动对照表格的方式进行转换，浪费了大量的时间，希望今后能在提供coe文件的同时提供对应的mars指令对照文件。另外由于coe文件中涵盖了所有指令的测试，使得无法对于单独指令进行有效测试，而在modelsim和输出文件中定位会消耗大量的时间，希望今后能够提供一些较小的方便手动模拟的测试文件用于测试。