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**

一、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

二、总体设计

1. 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。

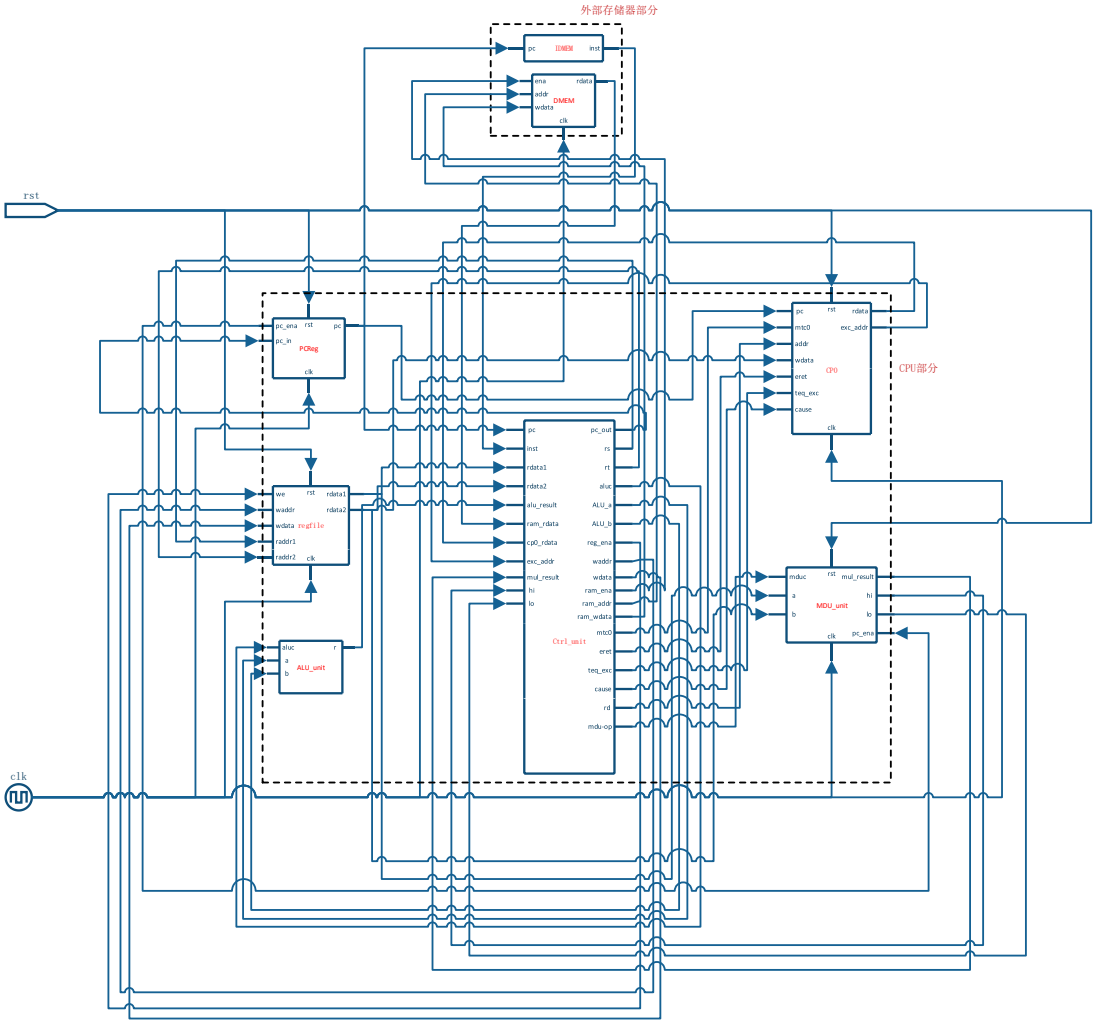
   pc 寄存器为时序逻辑部件，用于存储上一周期指令执行完毕下一条指令的地址，在时钟上升沿到来时赋值给 pc，并从外部存储器指令段获取指令。同时其会有一个使能端，仅有使能端允许时才会对 pc 进行赋值，这是受除法器所限制的功能。

   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

2. 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）：



三、模块建模

1. 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
```

2. 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
    );
```

3. 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
    );
```

4. 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
    );
```

5. 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
    );
```

6. 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
    );
```

7. 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
    );
```

8. 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
    );
```

9. 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
    );
```

10. 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
    );
```

11. 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
    );
```

12. 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
    );
```

13. 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
);
```

14. 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
);
```

15. 整倍数分频器

```
module Divider
    #(
        parameter mod = 32'd20
    )
    (
        input I_CLK,
        input RST,
        output reg O_CLK
);
```

四、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：

1.1 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
```

1.2 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
```

2. 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
```

3. 主程序 testbench：

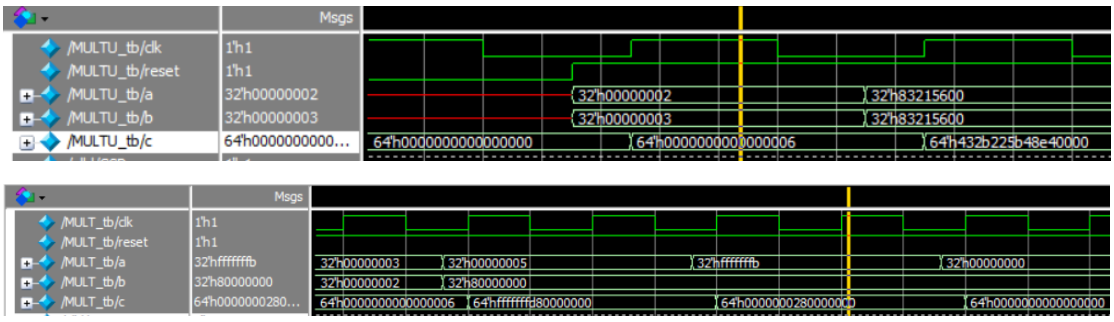
```
`include "defines.vh"

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",
```

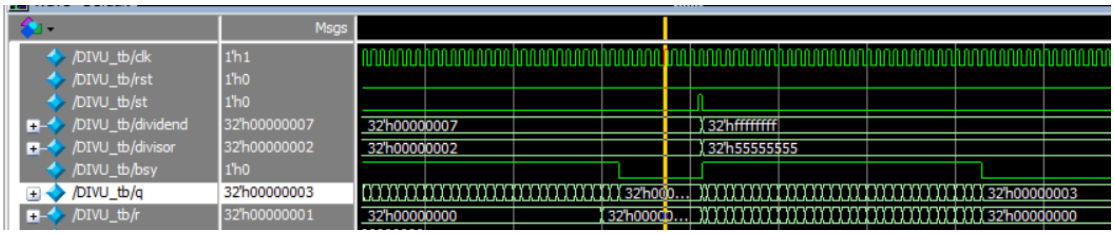
```
cpu_tb.uut.sccpu.cpu_ref.array_reg[20]);
                $fdisplay(file_output,          "regfile21:          %h",
cpu_tb.uut.sccpu.cpu_ref.array_reg[21]);
                $fdisplay(file_output,          "regfile22:          %h",
cpu_tb.uut.sccpu.cpu_ref.array_reg[22]);
                $fdisplay(file_output,          "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
```

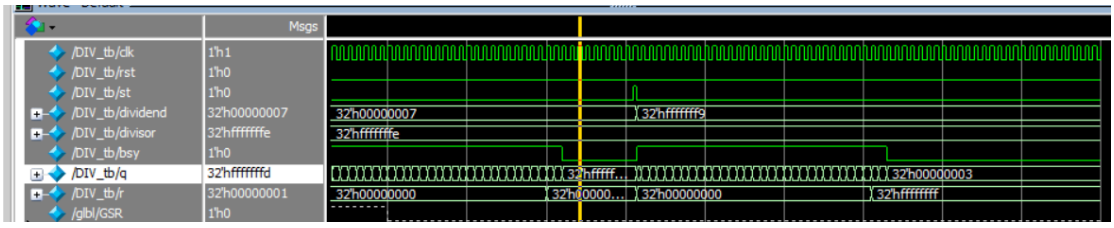
五、CPU 测试结果分析

1. MULT/MULTU 测试结果（modelsim 仿真波形图）



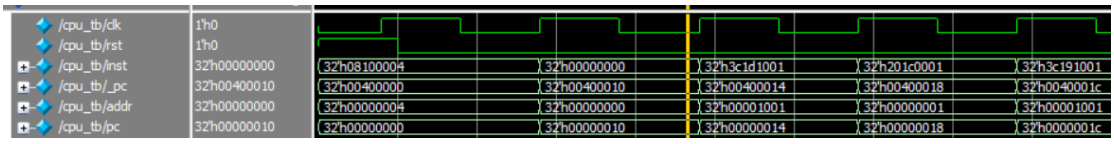
2. DIV/DIVU 测试结果（modelsim 仿真波形图）





3. 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

六、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

七、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
```

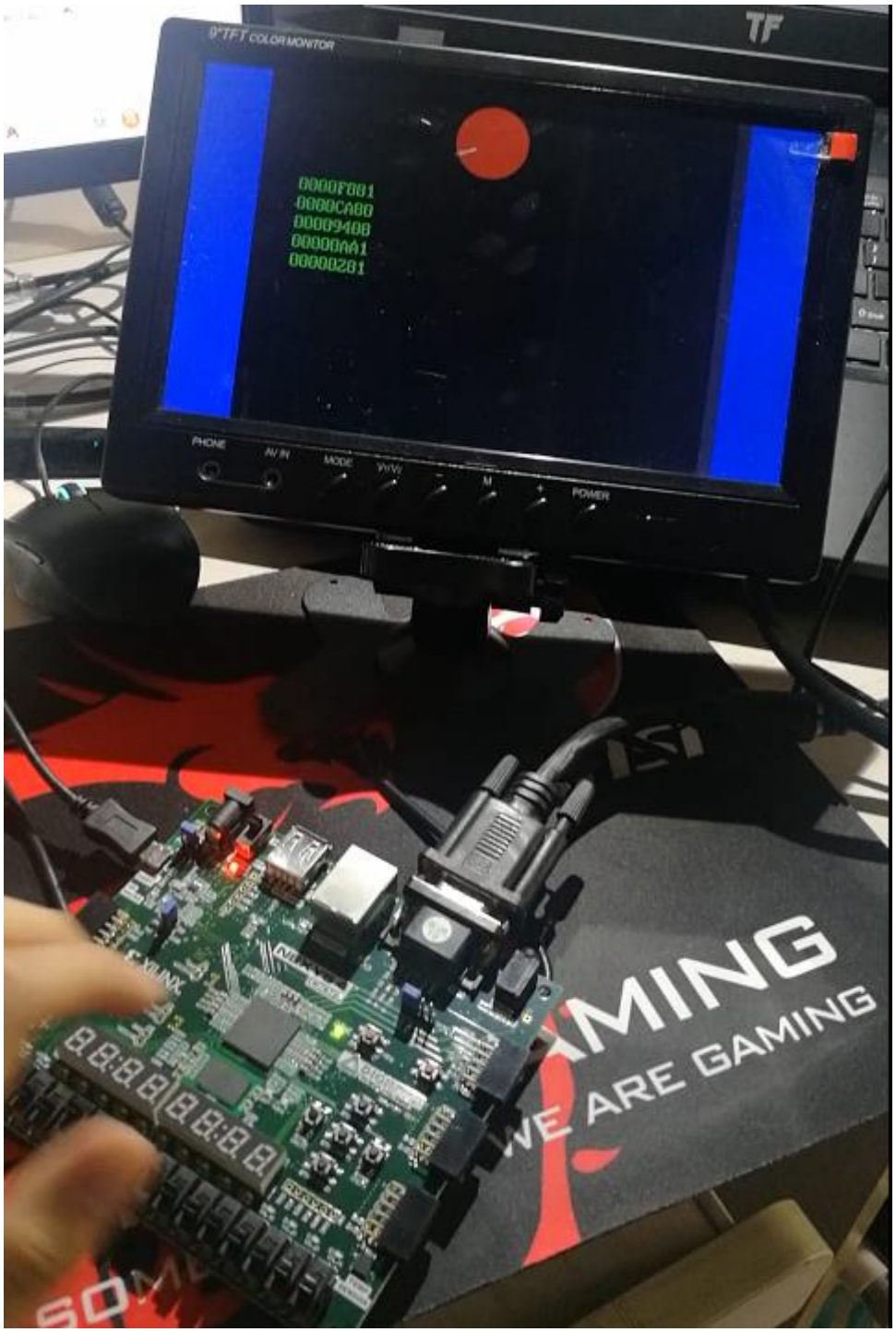
```
        addi $7, $9, 0
skip25:
        sub $8, $5, $6
        bgez $8, skip34
        addi $9, $5, 0
        addi $5, $6, 0
        addi $6, $9, 0
skip34:
        sub $8, $5, $7
        bgez $8, skip35
        addi $9, $5, 0
        addi $5, $7, 0
        addi $7, $9, 0
skip35:
        sub $8, $6, $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
```

八、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



九、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。