中山大学数据科学与计算机学院本科生实验报告

（2020 学年秋季学期）

课程名称：**计算机组成原理实验**　　任课教师：　　助教：

年级&班级	19 计科超算	专业(方向)	
学号	18324034	姓名	林天皓
电话		Email	linth5@mail2.sysu.edu.cn
开始日期	2020.11.20	完成日期	2020.11.27

一、 实验题目

实验 9 单时钟周期 CPU 的设计实验-1

二、实验目的

⑴ 理解 MIPS 常用的指令系统并掌握单周期 CPU 的工作原理与逻辑功能实现。

⑵ 过对单周期 CPU 的运行状况进行观察和分析，进一步加深理解。

三、实验内容与

1 实验原理

本次实验是设计单周期 CPU 中的一次整体模块实验，在之前的实验中，我们已经实现了 ALU 模块，立即数扩展模块，数码管显示模块，寄存器堆模块等，在这次的实验中，需要实现一个控制模块，完成从指令到各种控制信号的转换，通过输入指令，来生成各种的控制信号，以便完成各个组件之间的沟通。通过仿真验证，独立验证控制模块生成的各种控制信号。然后建立顶端文件测试被控制单元控制的各个模块能否正常工作，接下来实现单周期 cpu 的顶层文件，通过仿真测试 16 条指令的运行是否准确。

2. 实验步骤

由于ALU模块，立即数扩展模块，数码管显示模块，寄存器堆模块已经在之前的实验中

实现过，此处不再讲述。

第一部分：实现控制器模块

再根据mips指令关系与各种控制信号关系的真值表，如下图。

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

图1-指令与控制信号关系图（来源课本）

根据上表实现控制器模块，采用 case 语句，使得不同的控制信号经过控制模块的转换后可以得到各个控制信号，同时需要加入一个 default 的条件使得当输入的指令不合法的情况下的处理情况。通过查看老师给的代码，发现其中有两个 add 指令，却没有 addiu 指令，于是将其中第二个 addi 指令修改为 addiu 指令的控制，同时缺少了 sltiu 和 jal 指令，于是自行添加了。实现代码如下：

```
`timescale 1ns / 1ps
module ctr(
          input [5:0] opCode, output reg regDst, output reg aluSrc, output reg me
mToReg, output reg regWrite, output reg memRead, output reg memWrite, output reg b
ranch,
          output reg ExtOp, //符号扩展方式，1 为 sign-extend，0 为 zero-extend
          output reg[3:0] aluop, // 经过 ALU 控制译码决定 ALU 功能
          output reg jmp
      );
always@(opCode) begin
    // 操作码改变时改变控制信号
    case(opCode)
        6'b000010: begin // 'J 型' 指令操作码：000010，无需 ALU
```

```
            regDst = 0;  aluSrc = 0; memToReg = 0;
            regWrite = 0; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0111;
jmp = 1; ExtOp = 1;
        end
        6'b000011: begin // 'jal' 指令操作码: 000011，无需 ALU
            regDst = 0;  aluSrc = 0; memToReg = 0;
            regWrite = 0; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0111;
jmp = 1; ExtOp = 1;
        end
        6'b000000: begin// 'R 型' 指令操作码: 000000
            regDst = 1;  aluSrc = 0; memToReg = 0;
            regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0000;
jmp = 0; ExtOp = 1;
        end
        6'b100011: begin// 'lw' 指令操作码: 100011
            regDst = 0;  aluSrc = 1; memToReg = 1;
            regWrite = 1; memRead = 1; memWrite = 0; branch = 0; aluop = 4'b0011;
jmp = 0; ExtOp = 1;
        end
        6'b101011: begin// 'sw' 指令操作码: 101011
            regDst = 0;  aluSrc = 1; memToReg = 0;
            regWrite = 0; memRead = 0; memWrite = 1; branch = 0; aluop = 4'b0011;
jmp = 0; ExtOp = 1;
        end
        6'b000100: begin// 'beq' 指令操作码: 000100
            regDst = 0;  aluSrc = 0; memToReg = 0;
            regWrite = 0; memRead = 0; memWrite = 0; branch = 1; aluop = 4'b0101;
jmp = 0; ExtOp = 1;
        end
        6'b000101: begin// 'bne' 指令操作码: 000101
            regDst = 0;  aluSrc = 0; memToReg = 0;
            regWrite = 0; memRead = 0; memWrite = 0; branch = 1; aluop = 4'b0110;
jmp = 0; ExtOp = 1;
        end
        6'b001000: begin// 'addi' 指令操作码: 001000
            regDst = 0;  aluSrc = 1; memToReg = 0;
            regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0011;
jmp = 0;ExtOp = 0;
        end
        6'b001001: // 'addiu' 指令操作码: 001001
        begin
            regDst = 0; aluSrc = 1; memToReg = 0;
            regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0011;
jmp = 0; ExtOp = 0;
        end
        6'b001100: begin// 'andi' 指令操作码: 001100
```

```
            regDst = 0;  aluSrc = 1; memToReg = 0;
            regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0001;
jmp = 0; ExtOp = 0;
        end
        6'b001101: begin// 'ori' 指令操作码: 001101
            regDst = 0;  aluSrc = 1; memToReg = 0;
            regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0010;
jmp = 0; ExtOp = 0;
        end
        6'b001110: begin// 'xori' 指令操作码: 001110
            regDst = 0;  aluSrc = 1; memToReg = 0;
            regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b1000;
jmp = 0; ExtOp = 0;
        end
        6'b001010: begin// 'slti' 指令操作码: 001 010
            regDst = 0;  aluSrc = 1; memToReg = 0;
            regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0100;
jmp = 0; ExtOp = 1;
        end
        6'b001011: begin// 'sltiu' 指令操作码: 001011
            regDst = 0;  aluSrc = 1; memToReg = 0;
            regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0100;
jmp = 0; ExtOp = 0;
        end
        6'b001111: begin// 'lui' 指令操作码: 001111
            regDst = 0;  aluSrc = 1; memToReg = 0;
            regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0111;
jmp = 0; ExtOp = 0;
        end
        default: begin// 默认设置
            regDst = 0;  aluSrc = 0; memToReg = 0;
            regWrite = 0; memRead = 0; memWrite = 0; branch = 0; aluop = 3'b0xxx;
jmp = 0; ExtOp = 0;
        end
    endcase end
endmodule
```

第二部分：实现cpu顶层模块

接下来通过顶端模块，将之前实现过的所有的模块 组合到一起，同时将需要执行的指令通过硬编码的方式写入到指令内存中

顶层模块代码如下：

```
`timescale 1ns / 1ps
module top(
```

```
input clkin, input reset,
input [7:0]PC,
output [6:0] sm_duan,//段码
output [3:0] sm_wei,//哪个数码管
output [31:0]aluRes,
output [31:0]instruction
);
// 复用器信号线
//wire[31:0] expand2, mux4, mux5, address, jmpaddr;
//数据存储器
wire[31:0] memreaddata;
// 指令存储器
//wire [31:0] instruction;
reg[7:0] Addr;
// CPU 控制信号线
wire reg_dst,jmp,branch, memread, memwrite, memtoreg,alu_src,ExtOp;
wire[3:0] aluop;
wire regwrite;
// ALU 控制信号线
wire ZF,OF,CF; //alu 运算为零标志
wire[31:0] aluRes; //alu 运算结果
// ALU 控制信号线
wire[3:0] aluCtr;//根据 aluop 和指令后 6 位 选择 alu 运算类型
//wire[31:0] aluRes; //alu 运算结果
wire[31:0] input2;
wire [15:0]data;
// 寄存器信号线
wire[31:0] RsData, RtData;
wire[31:0] expand; wire[4:0] shamt;
wire [4:0]regWriteAddr;
wire[31:0]regWriteData;
assign shamt=instruction[10:6];
assign regWriteAddr = reg_dst ? instruction[15:11] : instruction[20:16];//写寄存器
的目标寄存器来自 rt 或 rd
assign data=aluRes[15:0];
assign regWriteData = memtoreg ? memreaddata : aluRes; //写入寄存器的数据来自 ALU 或
数据寄存器
assign input2 = alu_src ? expand : RtData; //ALU 的第二个操作数来自寄存器堆输出或指令
低 16 位的符号扩展
// 例化指令存储器
IM_unit IM ( .clk(clkin), .Addr(PC), .instruction(instruction) );

// 实例化控制器模块
ctr mainctr(
.opCode(instruction[31:26]),
.regDst(reg_dst),
```

```
.aluSrc(alu_src),
.memToReg(memtoreg),
.regWrite(regwrite),
.memRead(memread),
.memWrite(memwrite),
.branch(branch),
.ExtOp(ExtOp),
.aluop(aluop),
.jmp(jmp));
//  实例化 ALU 控制模块
aluctr aluctr1(
.ALUOp(aluop),
.funct(instruction[5:0]),
.ALUCtr(aluCtr));
// 。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。实例化寄存器模块
RegFile regfile(
.R_Addr_A(instruction[25:21]),
.R_Addr_B(instruction[20:16]),
.Clk(!clkin),
.Clr(reset),
.W_Addr(regWriteAddr),
.W_Data(regWriteData),
.Write_Reg(regwrite),
.R_Data_A(RsData),
.R_Data_B(RtData)
);

// 。。。。。。。。。。。。。。。。。。。。。。。。。。。。。实例化 ALU 模块
alu alu(
.shamt(shamt),
.input1(RsData), //写入 alu 的第一个操作数必是 Rs
.input2(input2),
.aluCtr(aluCtr),
.ZF(ZF),
.OF(OF),
.CF(CF),
.aluRes(aluRes));
//实例化符号扩展模块
signext signext(.inst(instruction[15:0]),.ExtOp(ExtOp), .data(expand));
//实例化数据存储器
 DM_unit dm(.clk(clkin), .Wr(memwrite),
            .reset(reset),
             .DMAdr(aluRes),
             .wd(RtData),
             .rd( memreaddata));
//.............................实例化数码管显示模块
```

```
display Smg(.clk(clkin),.sm_wei(sm_wei),.data(data),.sm_duan(sm_duan));

endmodule
```

仿真测试部分：

接下来通过仿真测试来检验模块实现是否正确，直接给 OP 赋值，来查看对饮控制变量的值的变化关系。

第一部分：控制模块的仿真测试

仿真测试控制模块代码如下：

```
module ctrsim;
// Inputs
reg [5:0] opCode;
// Outputs
wire regDst;
wire aluSrc;
wire memToReg;
wire regWrite;
wire memRead;
wire memWrite;
wire branch;
wire [3:0] aluop;
wire jmp;
// Instantiate the Unit Under Test (UUT)
ctr uut (
    .opCode(opCode),
    .regDst(regDst),
    .aluSrc(aluSrc),
    .memToReg(memToReg),
    .regWrite(regWrite),
    .memRead(memRead),
    .memWrite(memWrite),
    .branch(branch),
    .aluop(aluop),
    .jmp(jmp)
);
initial begin
    $timeformat(-9, 0, " ns");
    opCode = 6'b000000;
    $display ("%0t opCode=%6b regDst=%1b aluSrc=%1b memToReg=%1b regWrite=%1b memR
ead=%1b memWrite=%1b branch=%1b aluop=%4b jmp=%1b",
    $time, opCode,regDst,aluSrc,memToReg,regWrite,memRead,memWrite,branch,aluop,jm
p);
```

```
    #100;
    opCode = 6'b000010;
    $display ("%0t opCode=%6b regDst=%1b aluSrc=%1b memToReg=%1b regWrite=%1b memR
ead=%1b memWrite=%1b branch=%1b aluop=%4b jmp=%1b",
    $time, opCode,regDst,aluSrc,memToReg,regWrite,memRead,memWrite,branch,aluop,jm
p);
    #100;
    opCode = 6'b110000;//不存在的指令
    $display ("%0t opCode=%6b regDst=%1b aluSrc=%1b memToReg=%1b regWrite=%1b memR
ead=%1b memWrite=%1b branch=%1b aluop=%4b jmp=%1b",
    $time, opCode,regDst,aluSrc,memToReg,regWrite,memRead,memWrite,branch,aluop,jm
p);
    #100;
    opCode = 6'b100011;
    $display ("%0t opCode=%6b regDst=%1b aluSrc=%1b memToReg=%1b regWrite=%1b memR
ead=%1b memWrite=%1b branch=%1b aluop=%4b jmp=%1b",
    $time, opCode,regDst,aluSrc,memToReg,regWrite,memRead,memWrite,branch,aluop,jm
p);
    #100;
    opCode = 6'b000100;
    $display ("%0t opCode=%6b regDst=%1b aluSrc=%1b memToReg=%1b regWrite=%1b memR
ead=%1b memWrite=%1b branch=%1b aluop=%4b jmp=%1b",
    $time, opCode,regDst,aluSrc,memToReg,regWrite,memRead,memWrite,branch,aluop,jm
p);
    #100;
    opCode = 6'b001000;
    $display ("%0t opCode=%6b regDst=%1b aluSrc=%1b memToReg=%1b regWrite=%1b memR
ead=%1b memWrite=%1b branch=%1b aluop=%4b jmp=%1b",
    $time, opCode,regDst,aluSrc,memToReg,regWrite,memRead,memWrite,branch,aluop,jm
p);
end
endmodule
```

在检测的过程中，其中有一个不存在的指令，来测试遇到不在指令集内的指令的情况下仍然能正常运行。

第二部分：整体程序执行仿真模拟

首先要了解老师给的文件中指令所代表的汇编语句，这部分老师给的语句中后面的注释与语句实际情况有相当多的错误，以下的代码修改一处的代码，其他为写出原来的语句实际上的对应汇编语句。经过修改后的代码指令储存器代码的指令如下:

```
`timescale 1ns / 1ps
module IM_unit( input clk,
```

```
                input [3:0] Addr,       //指令存储器地址编码
                output reg [31:0]  instruction );// 寄存器的值
//寄存器地址都是 4 位二进制数，因为寄存器只有 16 个，4 位就能表示所有寄存器
reg [31:0] regs [0:15]; // 寄存器组
initial
begin
    regs[0]  = 32'h20110003;// addi $s1,$0,3 #alures=3
    regs[1]  = 32'h20120005;// addi $s2,$0,5 #alures=5
    //regs[2]原始的有错误原来第四位为 2，应该为 3,已经修改
    regs[2]  = 32'h20130003;// addi $s3,$0,3  #alures=3
    regs[3]  = 32'h2324820;// add $t1,$s1,$s2 #t1=3+5=8, alures=8
    regs[4]  = 32'h02515022;// sub $t2,$s2,$s1  #t2=5-3=2, alures=2
    regs[5]  = 32'h20111535;// addi $s1,$0,0x00001535 #aluRes=1535
    regs[6]  = 32'h2324824;// and $t1,$s1,$s2 #011&101=001, alures=1
    //regs[7]的指令注释有错误，应该为 addi $s2,$0,2446
    regs[7]  = 32'h20122446;//add $t2,$t2,$0（错误）
    //regs[8]的指令注释有错误，应该为 sw   mem(0),$17(s1),0
    regs[8]  = 32'hac110000;// lw（错误）
    //regs[9]的指令注释有错误，应该为 sw   mem(0),$18(s2),4
    regs[9]  = 32'hac120004;// sw（错误）
    //regs[10]的指令注释有错误，应该为 add  $4,$4,$3
    regs[10]  = 32'h00832020;//add $4,$2,$3
    regs[11]  = 32'h00831022;//sub $2,$4,$3
    //regs[12]的指令注释有部分错误，应该为 lui  $7,4099
    regs[12]  = 32'h3c471003;// lui（不完整）
    //regs[13]的指令注释有部分错误，应该为 slitu $1,$2,6186
    regs[13]  = 32'b00101100001000100001100000101010;// addi（错误）
    regs[14]  = 32'h00831025;//or $2,$4,$3
    regs[15]  = 32'h00831024;//and $2,$4,$3
end
always @( posedge clk ) // 时钟上升沿操作
    instruction=regs[Addr] ; //  取指令
endmodule
```

该段代码与 mips 汇编指令的正确对应关系如下：

```
1.addi $17(s1),$0,3
2.addi $18(s2),$0,5
3.addi $19(s3),$0,3
4.add  $9(t1),$17(s1),$18(s2)
5.sub  $10(t2),$18(s2),$17(s1)
6.addi $17(s1),$0,1535
7.and  $9(t1),$17(s1),$18(s2)
8.addi  $17(s1),$0,2446
9.sw   mem(0),$17(s1),0
10.sw   mem(4),$18(s2),0
```

```
11.add  $4,$4,$3
12.sub  $2,$4,$3
13.lui  $7,1003
14.slitu $1,$2,0x182A
15.or   $2,$4,$3
16.and  $2,$4,$3
```

顶层模块将各种模块的组合，在模拟仿真中，通过调整 PC 的值，在时钟周期的上升沿就有一个新的指令开始被执行。

顶层仿真文件的代码如下：

```
`timescale 1ns / 1ps
module topsim;
// Inputs
reg clkin;
reg reset;
reg [7:0]PC;
wire [6:0] sm_duan;//段码
wire [3:0] sm_wei;//哪个数码管
wire [31:0]aluRes;
wire [31:0]instruction;
// Instantiate the Unit Under Test (UUT)
top uut (
.clkin(clkin),
.reset(reset) ,
.PC(PC),
.aluRes(aluRes),
.instruction(instruction),
.sm_duan(sm_duan),
.sm_wei(sm_wei)
);
//wire reg_dst,jmp,branch, memread, memwrite, memtoreg,alu_src;
//ire[1:0] aluop;

initial begin
// Initialize Inputs
clkin = 0;
reset = 1;
PC=0;
// Wait 100 ns for global reset to finish
#10;
reset = 0;
end
parameter PERIOD = 20;
always begin
```

```
clkin = 1'b1;
#(PERIOD / 2) clkin = 1'b0;
#(PERIOD / 2) ;
PC=PC+1;
end
endmodule
```

所有的模块实现完毕，接下来通过仿真测测试展示结果。

四、实验结果

第一部分：控制模块的仿真实验结果

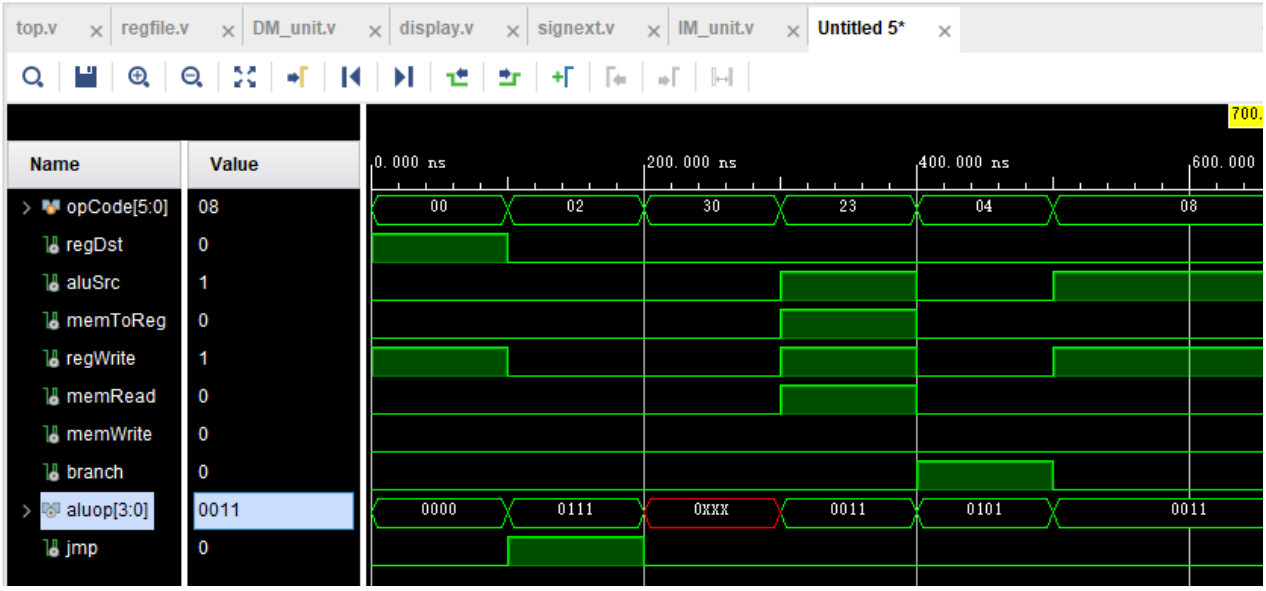

图 2-控制模块仿真波形图

```
# run 1000ns
0 ns opCode=000000 regDst=x aluSrc=x memToReg=x regWrite=x memRead=x memWrite=x branch=x aluop=xxxx jmp=x
100 ns opCode=000010 regDst=1 aluSrc=0 memToReg=0 regWrite=1 memRead=0 memWrite=0 branch=0 aluop=000 jmp=0
200 ns opCode=110000 regDst=0 aluSrc=0 memToReg=0 regWrite=0 memRead=0 memWrite=0 branch=0 aluop=111 jmp=1
300 ns opCode=100011 regDst=0 aluSrc=0 memToReg=0 regWrite=0 memRead=0 memWrite=0 branch=0 aluop=xxx jmp=0
400 ns opCode=000100 regDst=0 aluSrc=1 memToReg=1 regWrite=1 memRead=1 memWrite=0 branch=0 aluop=011 jmp=0
500 ns opCode=001000 regDst=0 aluSrc=0 memToReg=0 regWrite=0 memRead=0 memWrite=0 branch=1 aluop=101 jmp=0
INFO: [USF-XSim-96] XSim completed. Design snapshot 'ctrsim_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:03 ; elapsed = 00:00:06 . Memory (MB): peak = 851.449 ; gain = 0.000
```

图 3-控制模块仿真 vivado 输出

分析：上图为控制单元的仿真结果波形图，

第一段：opCode=000010 regDst=1 aluSrc=0 memToReg=0 regWrite=1 memRead=0 memWrite=0 branch=0 aluop=000 jmp=0，符合真值表预期结果，正确

第二段：opCode=110000 regDst=0 aluSrc=0 memToReg=0 regWrite=0 memRead=0 memWrite=0 branch=0 aluop=111 jmp=1，符合真值表预期结果，正确

第三段： opCode=100011 regDst=0 aluSrc=0 memToReg=0 regWrite=0 memRead=0 memWrite=0 branch=0 aluop=xxx jmp=0，符合真值表预期结果，正确

第四段：opCode=000100 regDst=0 aluSrc=1 memToReg=1 regWrite=1 memRead=1 memWrite=0 branch=0 aluop=011 jmp=0，符合真值表预期结果，正确

第五段：opCode=001000 regDst=0 aluSrc=0 memToReg=0 regWrite=0 memRead=0 memWrite=0 branch=1 aluop=101 jmp=0，符合真值表预期结果，正确

经过更多的测试和对比，控制器模块的仿真均符合能实现真值表中的信号状态，控制模块的实现正确。

第二部分：接下来进行 cpu 顶层模块的仿真测试。

仿真波形图如下：

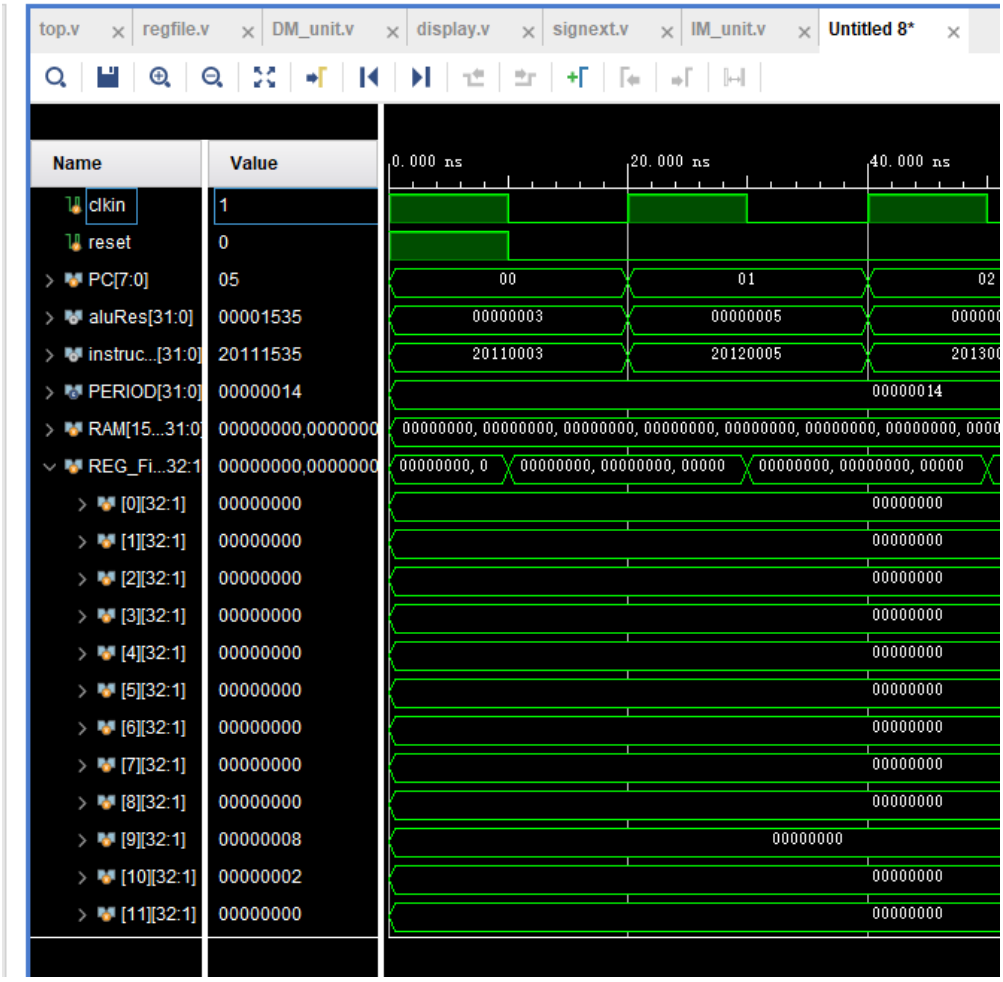

图 4-单周期 CPU 顶层仿真波形图

下面针对仿真文件中的逐条指令进行分析：

下图为执行前四条指令时，寄存器堆内数值的变化，用红色的数字表明这个寄存器的变化与第几条指令有关。

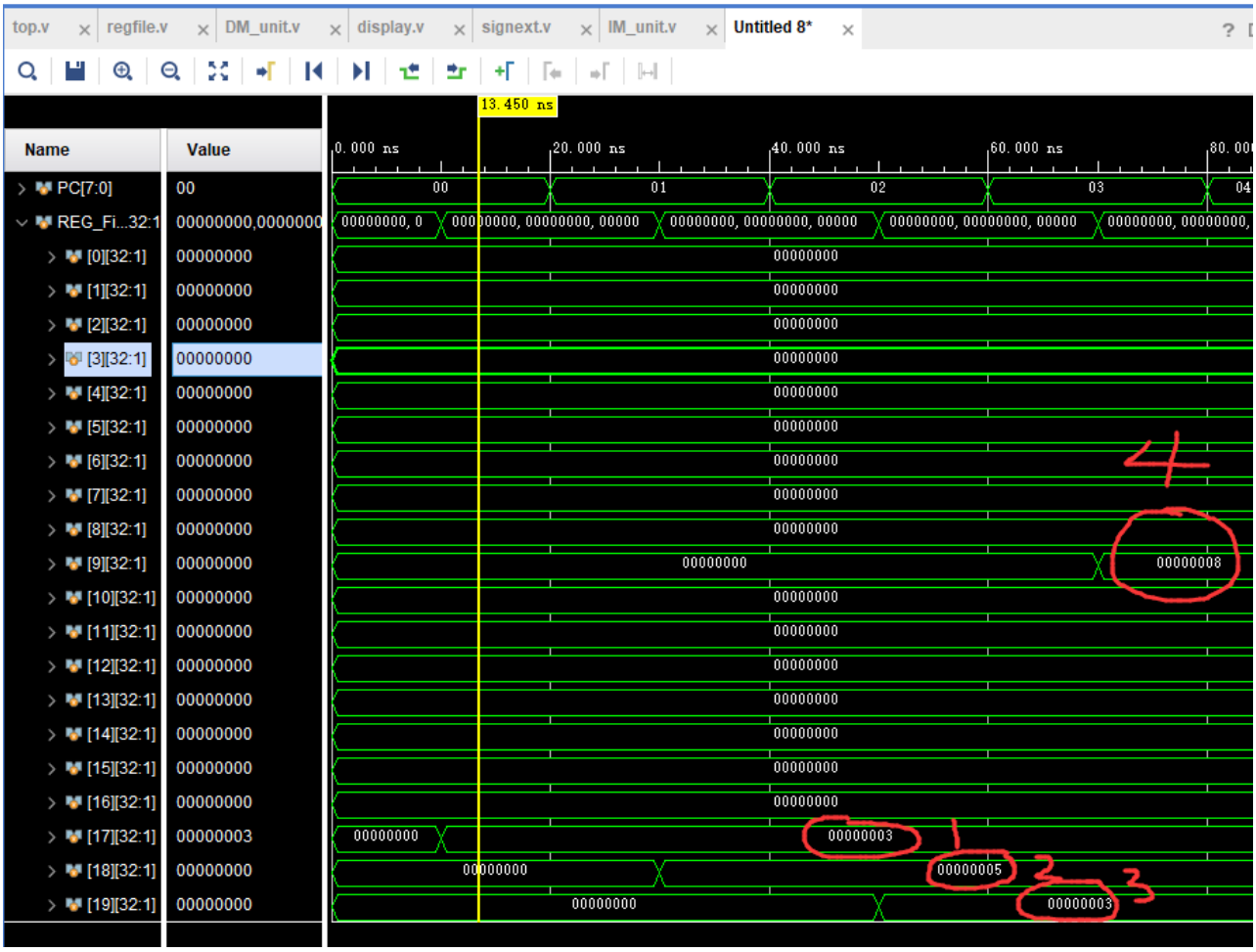

图 5-单周期 CPU 顶层仿真第 1，2，3，4 条指令寄存器变化

分析：

第一条指令为 addi $17(s1),$0,3

执行后寄存器 17 内的数值为 3，符合该汇编语句的预期结果，正确

第二条指令为 addi $18(s2),$0,5

执行后寄存器 18 内的数值为 5，符合该汇编语句的预期结果，正确

第三条指令为 addi $19,$0,3

执行后寄存器 19 内的数值为 3，符合该汇编语句的预期结果，正确

第四条指令为 add　$9(t1),$17(s1),$18(s2)

执行后寄存器 9 内的数值为 3+5=8，符合该汇编语句的预期结果，正确

第 5-8 条语句的波形图如下：

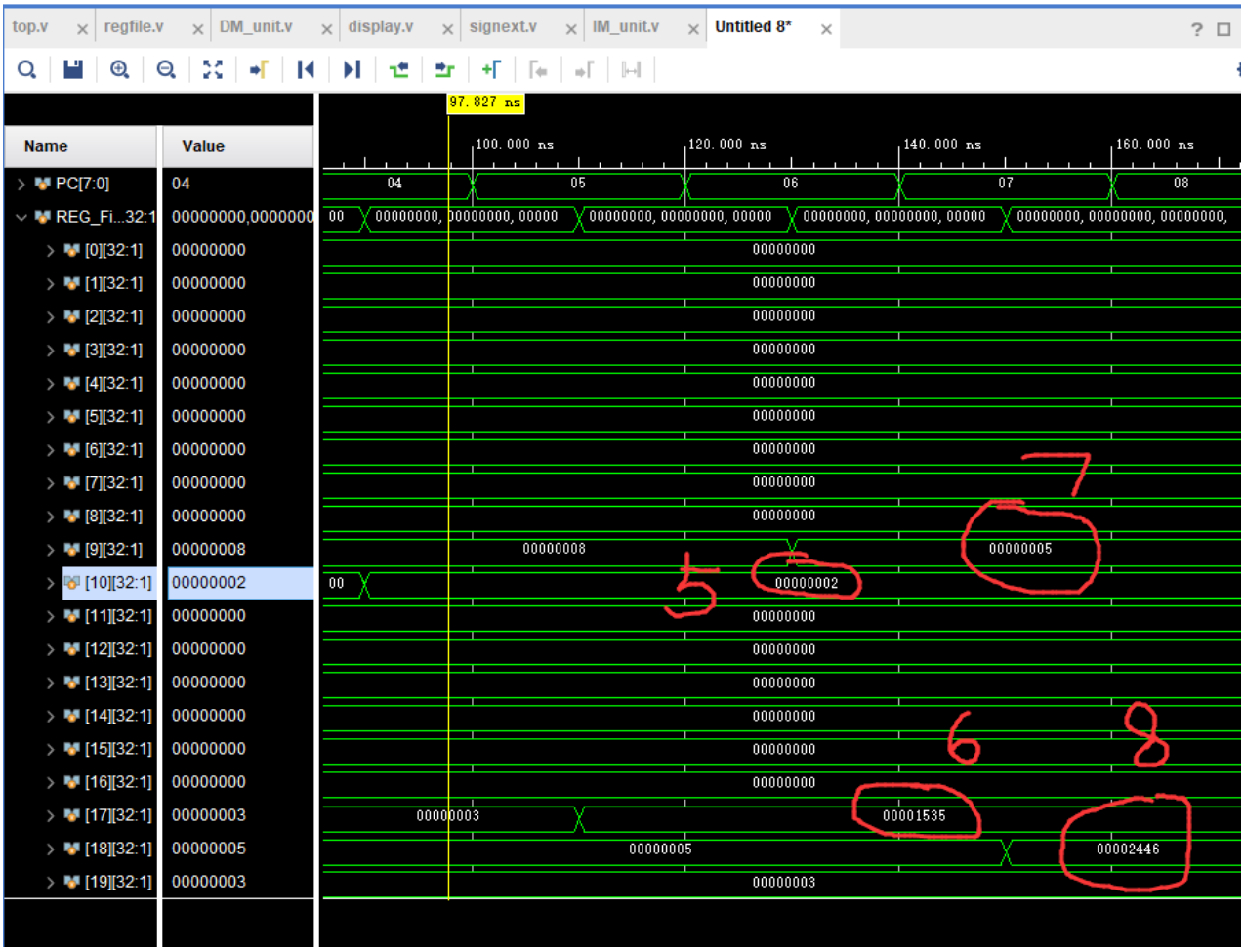

图 6-单周期 CPU 顶层仿真第 5，6，7，8 条指令寄存器变化

分析：

第五条指令为 sub　$10(t2),$18(s2),$17(s1)

执行后寄存器 10 内的数值为 5-3=2，符合该汇编语句的预期结果，正确

第六条指令为 addi $17(s1),$0,0x1535

执行后寄存器 17 内的数值为 0x1535，符合该汇编语句的预期结果，正确

第七条指令为 and　$9(t1),$17(s1),$18(s2)

执行后寄存器 9 内的数值为 0x1535&0x5=5，符合该汇编语句的预期结果，正确

第八条指令为 $17(s1),$0,0x2446

执行后寄存器 17 内的数值为 0x2446，符合该汇编语句的预期结果，正确

第 9-10 两条指令为 lw 指令此时展示的是数据内存空间的波形图

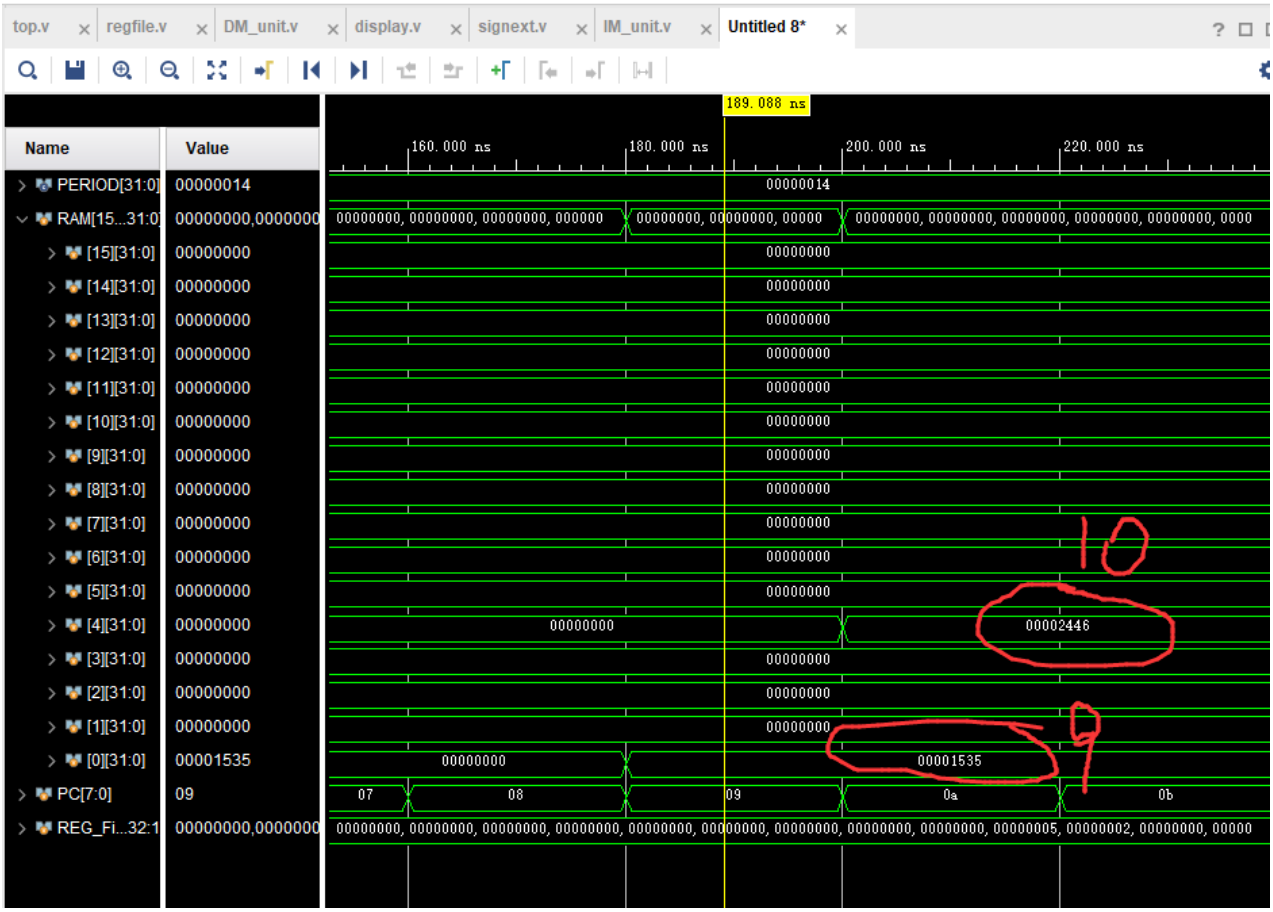

图 7-单周期 CPU 顶层仿真第 9，10 条指令内存变化

分析：

第九条指令为 mem(0),$17(s1),0

执行后内存地址为 0 的内存中的数值为 0x1535，符合该汇编语句的预期结果，正确

第十条指令为 mem(4),$18(s2),0

执行后内存地址为 4 的内存中的数值为 0x2446，符合该汇编语句的预期结果，正确

第十一条指令和第十二条指令分别为

add　$4,$4,$3

sub　$2,$4,$3

由于这下操作数都是 0，看不出来寄存器堆发生变化，因此不截图

第十三条指令的波形图变化如下：

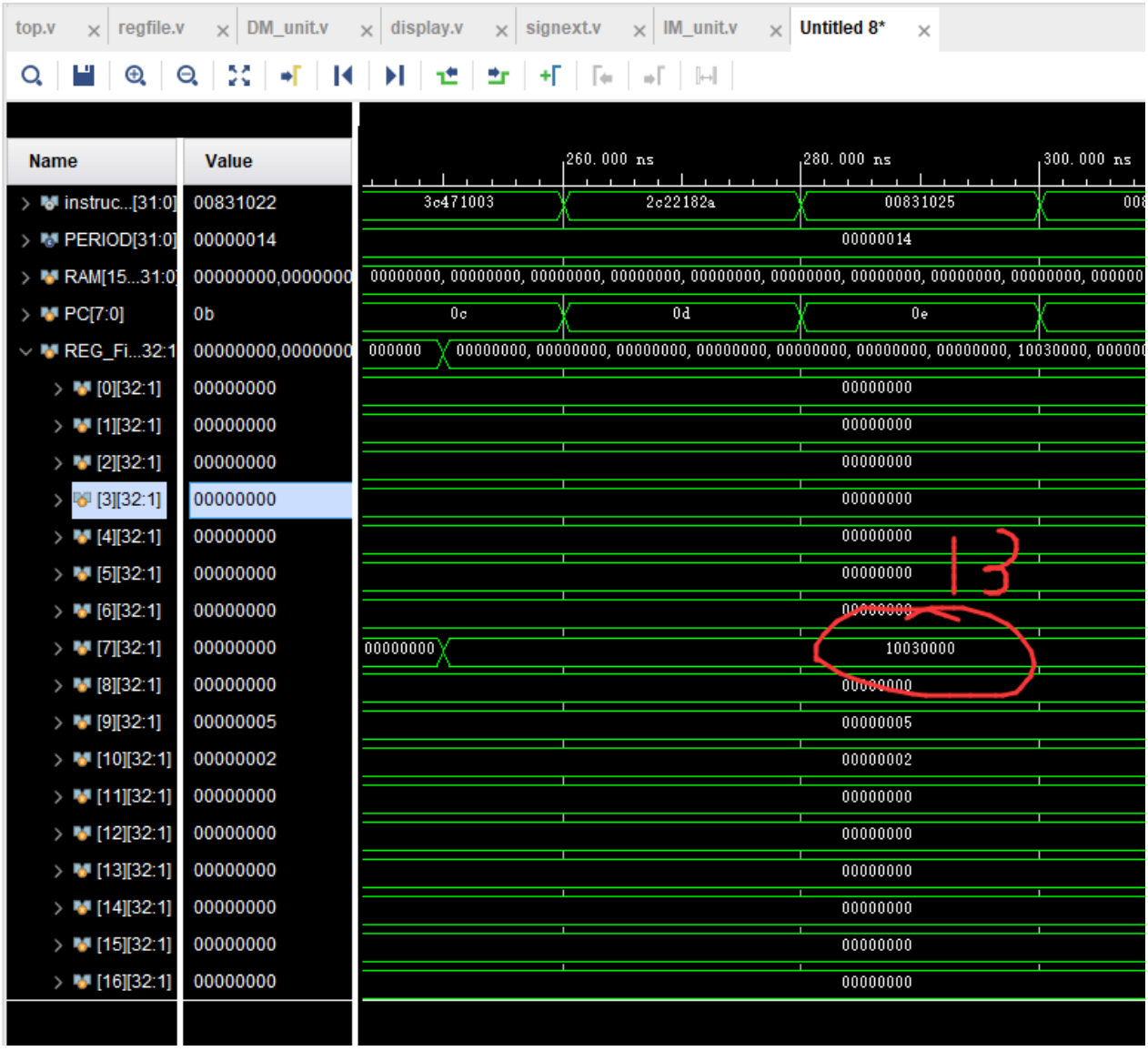

图 8-单周期 CPU 顶层仿真第 13 条指令寄存器变化

分析：指令为 lui　$7,1003

执行后寄存器 7 的中的数值为 0x10030000，符合该汇编语句的预期结果，正确

第十四条指令和第十五条与第十六条指令分别为

slitu $1,$2,0x182A

or　　$2,$4,$3

and　$2,$4,$3

由于这下操作经过运算后的结果为 0，看不出来寄存器堆发生变化，因此不截图了

总结：通过上述的截图，与逐条语句的分析，上述指令的执行结果均正确，单周期 CPU 的整体仿真均无误。

五、实验感想

本次终于初步实现了单周期 CPU 的仿真，并且一步一步的得到了预期的结果。在本次实验中，进一步领略了 mips 的指令与 CPU 中各种控制信号的关系。同时老师直接给的指令储存器模块文件中的指令的注释有很多不匹配的地方，也锻炼了我查找错误的能力。为了方便 debug 写了一个 python 程序将 16 进制的指令转换为相对可读的形式，python 代码如下：

```
while(1):
    a=str(input("hex instruction: "))
    a='0x'+a
    num=eval(a)
    b=str(bin(num))
    inst=b.split('b')[1]
    ap=32-len(inst)
    for i in range(0,ap):
        inst='0'+inst
    op=inst[0:6]
    rs=inst[6:11]
    rt=inst[11:16]
    rd=inst[16:21]
    shamt=inst[21:26]
    funct=inst[26:32]
    print(inst)
    print('op: ',op)
    print('rs: ',rs)
    print('rt: ',rt)
    print('rd: ',rd)
    print('shamt: ',shamt)
    print('funct: ',funct)
```

运行效果如下：

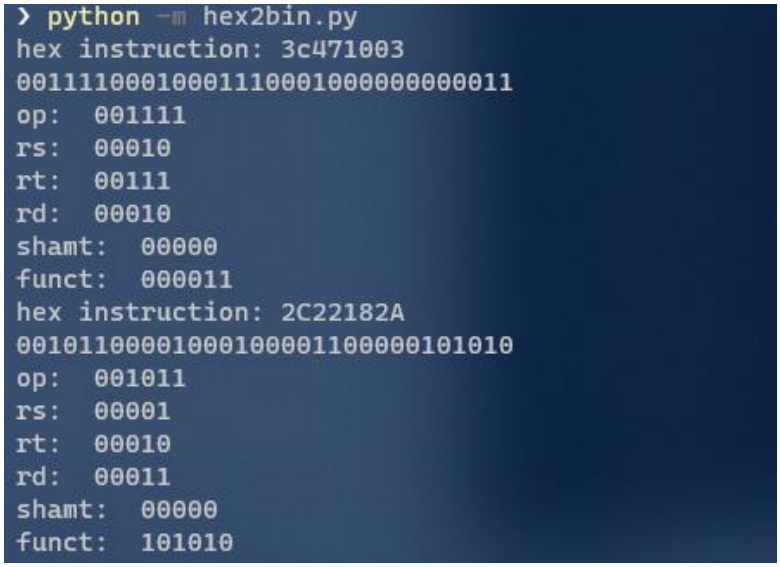

图 9-python 实现十六进制指令转换为指令各段的二进制

通过这种方式，将人很难读的机器指令转换为相对容易查表的格式，再进一步查表查找对应的指令。对于立即数的 I 型指令，使用计算机来计算二进制对应的十六进制数，慢慢将仿真的指令内存中的每一条指令更正，最终理解了仿真波形图中的每一条波形变化的原理，才有了上述对仿真结果的逐条分析。

附录（流程图，注释过的代码）：

本次CPU实验的结构如下：

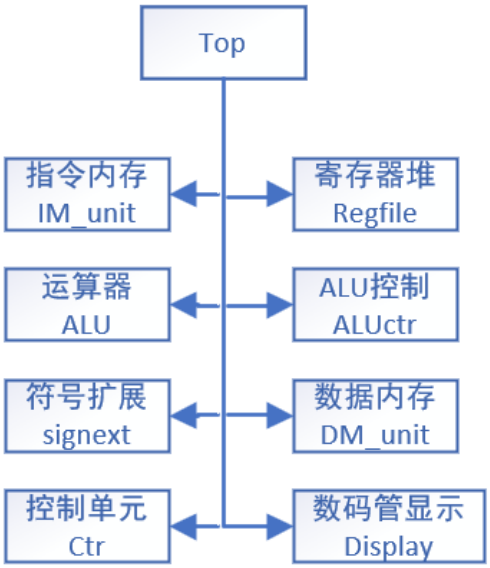

图 9-本次实验实现的单周期 CPU 结构图

本次实验代码较多，均存放在压缩包内，此处不再重复。