中山大學
SUN YAT-SEN UNIVERSITY

《计算机组成原理实验》

实验报告

（实验二）

学 院 名 称	：	数据科学与计算机学院
专业（班级）	：	22 计教学 3 班
学 生 姓 名	：	王泓沣
学　　号	：	22336226
时　　间	：	2023　年　11　月　26　日

成 绩 ：

实验二 ： 单周期CPU设计与实现

一.实验目的

1.掌握单周期CPU数据通路图的构成、原理及其设计方法；

2.掌握单周期CPU的实现方法，代码实现方法；

3.认识和掌握指令与CPU的关系；

4.掌握测试单周期CPU的方法。

二.实验内容

设计一个单周期 CPU，该 CPU 至少能实现以下指令功能操作。指令与格式如下:

==> 算术运算指令

（1）add rd , rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100000

功能：GPR[rd] ← GPR[rs] + GPR[rt]。

（2）sub rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100010

功能：GPR[rd] ← GPR[rs] - GPR[rt]。

（3）addiu rt , rs ,**immediate**

001001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] + sign_extend(**immediate**)； **immediate** 做符号扩展再参加“与”运算。

==> 逻辑运算指令

（4）andi rt , rs ,**immediate**

001100	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] and zero_extend(**immediate**)；**immediate** 做 0 扩展再参加“与”运算。

（5）and rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100100

功能：GPR[rd] ← GPR[rs] and GPR[rt]。

（6）ori rt , rs ,**immediate**

001101	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] or zero_extend(**immediate**)。

（7）or rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100101

功能：GPR[rd] ← GPR[rs] or GPR[rt]。

==>移位指令

（8）sll　rd, rt,sa

000000	00000	rt(5 位)	rd(5 位)	sa(5 位)	000000

功能：GPR[rd] ← GPR[rt] << sa。

==>比较指令

（9）　slti　rt, rs,**immediate**　　带符号数

001010	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if GPR[rs] < sign_extend(**immediate**)　GPR[rt] =1 else　GPR[rt] = 0。

==> 存储器读/写指令

（10）sw　rt , **offset** (rs) 写存储器

101011	rs(5 位)	rt(5 位)	**offset**(16 位)

功能：memory[GPR[base] + sign_extend(**offset**)] ← GPR[rt]。

（11) lw　rt , **offset** (rs) 读存储器

100011	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：GPR[rt] ← memory[GPR[base] + sign_extend(**offset**)]。

==> 分支指令

（12）beq　rs,rt, **offset**

000100	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：if(GPR[rs] = GPR[rt]) pc←pc + 4 + sign_extend(**offset**)<<2

else pc ←pc + 4

特别说明：**offset 是从 PC+4 地址开始和转移到的指令之间指令条数。**offset 符号扩展之后左移 2 位再相加。为什么要左移 2 位？由于跳转到的指令地址肯定是 4 的倍数（每条指令占 4 个字节），最低两位是“00”，因此将 offset 放进指令码中的时候，是右移了 2 位的，也就是以上说的“指令之间指令条数”。

（13）bne　rs,rt, **offset**

000101	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：if(GPR[rs] != GPR[rt]) pc←pc + 4 + sign_extend(**offset**) <<2

else pc ←pc + 4

（14）bltz　rs, **offset**

000001	rs(5 位)	00000	**offset** (16 位)

功能：if(GPR[rs] < 0) pc←pc + 4 + sign_extend (**offset**) <<2

else pc ←pc + 4。

==>跳转指令

（15）j　addr

000010	**addr**(26 位)

功能：PC ← {PC[31:28] , **addr** , 2’b0}，无条件跳转。

说明：由于 MIPS32 的指令代码长度占 4 个字节，所以指令地址二进制数最低 2 位均为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位可用于存放地址，事实上，可存放 28 位地址，剩下最高 4 位由 pc+4 最高 4 位拼接上。

==> 停机指令

（16）halt

111111	00000000000000000000000000(26 位)

功能：停机；不改变 PC 的值，PC 保持不变。

三.实验原理

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期（如果晶振的输出没有经过分频就直接作为 CPU 的工作时钟，则时钟周期就等于振荡周期。若振荡周期经二分频后形成时钟脉冲信号作为 CPU 的工作时钟，这样，时钟周期就是振荡周期的两倍。）

CPU 在处理指令时，一般需要经过以下几个步骤：

⑴ 取指令(**IF**)：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，同时，PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 PC，当然得到的“地址”需要做些变换才送入 PC。

⑵ 指令译码(**ID**)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

⑶ 指令执行(**EXE**)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

⑷ 存储器访问(**MEM**)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

⑸ 结果写回(**WB**)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

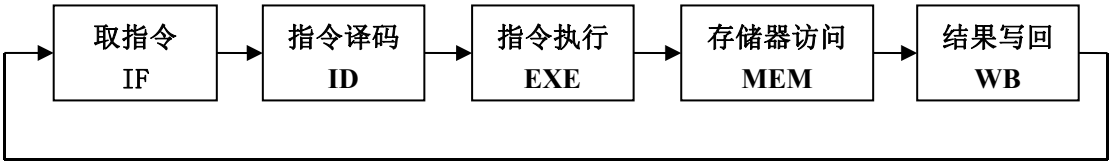


图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：

R 类型：

31–26	25–21	20–16	15–11	10–6	5–0
op	rs	rt	rd	sa	funct
6 位	5 位	5 位	5 位	5 位	6 位

I 类型：

31–26	25–21	20–16	15–0
op	rs	rt	immediate
6 位	5 位	5 位	16 位

J 类型：

31–26	25–0
op	address
6 位	26 位

其中，

op：为操作码；

rs：只读。为第 1 个源操作数寄存器，寄存器地址（编号）是 00000~11111，00~1F；

rt：可读可写。为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd：只写。为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

funct：为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能与操作码配合使用；

immediate：为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载（Laod）/数据保存（Store）指令的数据地址字节偏移量和分支指令中相对程序计数器（PC）的有符号偏移量；

address：为地址。

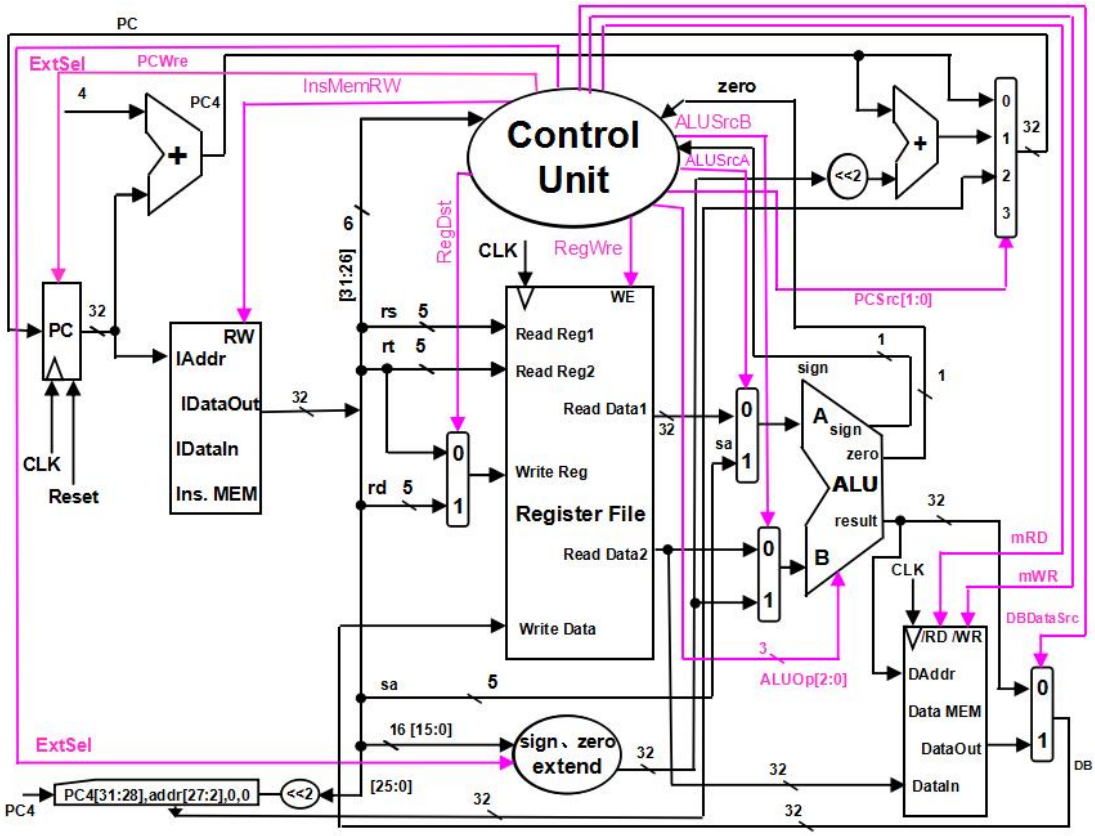


图 2　单周期 CPU 数据通路和控制线路图

图 2 是一个简单的基本上能够在单周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，先给出寄存器地址，读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。图中控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

表 1 控制信号的作用

相关部件及引脚说明：

控制信号名	状态“0”	状态“1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addiu、or、and、andi、ori、slti、beq、bne、bltz、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{1'b0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、beq、bne、bltz	来自 sign 或 zero 扩展的立即数，相关指令：addi、andi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、bltz、sw、halt	寄存器组写使能，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	输出高阻态	**读数据存储器，相关指令：lw**
mWR	无操作	**写数据存储器，相关指令：sw**
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addiu、andi、ori、slti、lw	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll
ExtSel	(zero-extend)**immediate(0 扩展)**，相关指令：addiu、andi、ori	(sign-extend)**immediate (符号扩展)**，相关指令：slti、sw、lw、beq、bne、bltz
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addiu、sub、or、ori、and、andi、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0)； 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bne(zero=0)、bltz(sign=1)； 10：pc<－{(pc+4)[31:28],addr[27:2],2'b00}，相关指令：j； 11：未用	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

Instruction Memory：指令存储器，

Iaddr，指令存储器地址输入端口

IDataIn，指令存储器数据输入端口（指令代码输入端口）

IDataOut，指令存储器数据输出端口（指令代码输出端口）

RW，指令存储器读写控制信号，为 0 写，为 1 读

Data Memory：数据存储器，

Daddr，数据存储器地址输入端口

DataIn，数据存储器数据输入端口

DataOut，数据存储器数据输出端口

/RD，数据存储器读控制信号，为 0 读

/WR，数据存储器写控制信号，为 0 写

Register File：寄存器组

Read Reg1，rs 寄存器地址输入端口

Read Reg2，rt 寄存器地址输入端口

Write Reg，将数据写入的寄存器端口，其地址来源 rt 或 rd 字段

Write Data，写入寄存器的数据输入端口

Read Data1，rs 寄存器数据输出端口

Read Data2，rt 寄存器数据输出端口

WE，写使能信号，为 1 时，在时钟边沿触发写入

ALU： 算术逻辑单元

result，ALU 运算结果

zero，运算结果标志，结果为 0，则 zero=1；否则 zero=0

sign，运算结果标志，结果最高位为 0，则 sign=0，正数；否则，sign=1，负数

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A － B	减
010	Y = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y= (A<B) ?1: 0	比较 A 与 B 不带符号
110	Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 && regb[31] == 0))) ? 1:0	比较 A 与 B 带符号
111	Y = A ⊕ B	异或

需要说明的是以上数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定 ALU 的运算功能(从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出以上表 1，这样，从表 1 可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表（留给学生完成），再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC 的改变是在时钟上升

沿进行的，这样稳定性较好。另外，值得注意的问题，设计时，用模块化的思想方法设计，

四.实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五.实验过程与结果

A. CPU 设计

在设计单周期CPU时，遵照模块化的思想方法自顶向下设计。先确定组成模块和线路，统一设计变量名后，分别实现底层模块，然后将对应的输入输出线路在顶层CPU中连接。

```
module CPU(
    input CLK,
    input Reset,
    output [31:0] nextPC,currPC,
    output [4:0] rs,rt,
    output [31:0] ReadData1,ReadData2,
    output [31:0] ALUOut,DataBus
);
    wire [5:0] opcode;
    wire [5:0] funct;
    wire [4:0] rd;
    wire [15:0] immediate;
    wire [31:0] bincode;
    wire [31:0] extended;
    wire PCWre,RegWre,RegDst,DBDataSrc,ALUSrcA,ALUSrcB,mRD,mWR,ExtSel;
    wire [1:0] PCSrc;
    wire [2:0] ALUOp;
    wire ALU_zero,ALU_sign;
    wire [5:0] WriteReg;
    wire [31:0] WriteData;
    wire [31:0] ALU_inA,ALU_inB;
    wire [31:0] DataOut;
    wire [31:0] next_in0,next_in1,next_in2;
    assign opcode = bincode[31:26];
    assign funct = bincode[5:0];
    assign rs = bincode[25:21];
    assign rt = bincode[20:16];
    assign rd = bincode[15:11];
    assign immediate = bincode[15:0];
    assign next_in0 = currPC+4;
```

```
assign next_in1 = next_in0+(extended<<2);
assign next_in2 = {next_in0[31:28],bincode[25:0]};
assign DataBus = WriteData;

ControlUnit ControlUnit(
        .OpCode(opcode),
        .funct(funct),
        .zero(ALU_zero),
        .sign(ALU_sign),
        .PCWre(PCWre),
        .ALUSrcA(ALUSrcA),
        .ALUSrcB(ALUSrcB),
        .DBDataSrc(DBDataSrc),
        .RegWre(RegWre),
        .InsMemRW(InsMemRW),
        .mRD(mRD),
        .mWR(mWR),
        .RegDst(RegDst),
        .ExtSel(ExtSel),
        .PCSrc(PCSrc),
        .ALUOp(ALUOp)
);
PC PC(
        .CLK(CLK),
        .Reset(Reset),
        .PCWre(PCWre),
        .PCIn(nextPC),
        .PCOut(currPC)
);
InstruMemory InstruMemory(
        .RW(InsMemRW),
        .IAddr(currPC),
        .IDataOut(bincode)
);
RegFile RegFile(
        .CLK(CLK),
        .Reset(Reset),
        .RegWre(RegWre),
        .ReadReg1(rs),
        .ReadReg2(rt),
        .WriteReg(WriteReg),
        .WriteData(WriteData),
        .ReadData1(ReadData1),
        .ReadData2(ReadData2)
);
```

```
  MUX_2_32bits Mux_ALU_inA(
            .Enable(ALUSrcA),
            .in0(ReadData1),
            .in1({27'd0,immediate[10:6]}),
            .out(ALU_inA)
      );

MUX_2_32bits Mux_ALU_inB(
            .Enable(ALUSrcB),
            .in0(ReadData2),
            .in1(extended),
            .out(ALU_inB)
      );
endmodule
```

1.PC

PC 是时序逻辑，存放当前的指令地址。引脚说明：
CLK，输入时钟信号；
Reset，复位，低电平有效，初始化 PC 为 0；
PCWre，控制信号，PC 是否可以更改，0 为不更改，1 为更改；
PCIn，当前指令地址；
PCOut，下一指令地址；

```
module PC(
      input CLK,
      input Reset,
      input PCWre,
      input [31:0] PCIn,
      output reg [31:0] PCOut
      );
      initial PCOut = 0;
      always@(posedge CLK)begin
            if(Reset==0)begin
                  PCOut=0;
            end
            else begin
                  if(PCWre==1)PCOut = PCIn;
            end
      end
endmodule
```

2. Instruction Memory

IM 为组合逻辑，设计为内含 100 个字节的 ROM。ROM 中存放要执行的测试程序段的机器码，使用 initial 语句中的$readmemb 伪指令从 single_cycleCPU.txt 中读出。引脚说明：

RW，读/写指令（此实验中指令为初始化时一次性写入，未用到）；

IAddr，指令地址；

IDataOut，输出的指令；

```
module InstruMemory(
    input RW,
    input [31:0] IAddr,
    output [31:0] IDataOut
    );
    reg [7:0] ram[0:99];
    initial begin
    $readmemb("E:/CPU_instructions/single_cycleCPU.txt",ram);
    end

    assign IDataOut[31:24]=ram[IAddr];
    assign IDataOut[23:16]=ram[IAddr+1];
    assign IDataOut[15:8]=ram[IAddr+2];
    assign IDataOut[7:0]=ram[IAddr+3];
endmodule
```

3. ControlUnit

CPU 控制单元通过输入的 zero 标志位 sign 当前指令中对应的操作码 op 和功能码 funct 来确定当前 CPU 需要采取的运算形式。

```
module ControlUnit(
    input [5:0] OpCode,
    input [5:0] funct,
    input sign,
    input zero,
    output reg PCWre,
    output reg ALUSrcA,
    output reg ALUSrcB,
    output reg DBDataSrc,
    output reg RegWre,
    output reg InsMemRW,
    output reg mRD,
    output reg mWR,
    output reg RegDst,
    output reg ExtSel,
    output reg [1:0] PCSrc,
    output reg [2:0] ALUOp
    );
```

```
always@(OpCode or funct or sign or zero)begin
    case(OpCode)
        6'b000000:begin //R-type
            case(funct)
                6'b100000:begin //add
                    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b10001;
                    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b1001x;
                    PCSrc[1:0]=2'b00;
                    ALUOp[2:0]=3'b000;
                end
                6'b100010:begin //sub
                    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b10001;
                    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b1001x;
                    PCSrc[1:0]=2'b00;
                    ALUOp[2:0]=3'b001;
                end
                6'b100100:begin //and
                    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b10001;
                    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b1001x;
                    PCSrc[1:0]=2'b00;
                    ALUOp[2:0]=3'b100;
                end
                6'b100101:begin //or
                    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b10001;
                    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b1001x;
                    PCSrc[1:0]=2'b00;
                    ALUOp[2:0]=3'b011;
                end
                6'b000000:begin//sll
                    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b11001;
                    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b10010;
                    PCSrc[1:0]=2'b00;
                    ALUOp[2:0]=3'b010;
                end
            endcase
        end
```

```
6'b001001:begin //addiu
        {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b10101;
        {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b10001;
        PCSrc[1:0]=2'b00;
        ALUOp[2:0]=3'b000;
end
6'b001100:begin //andi
    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b10101;
    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b10000;
    PCSrc[1:0]=2'b00;
    ALUOp[2:0]=3'b100;
end
6'b001101:begin //ori
    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b10101;
    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b10000;
    PCSrc[1:0]=2'b00;
    ALUOp[2:0]=3'b011;
end
6'b001010:begin //slti
    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b10101;
    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b10001;
    PCSrc[1:0]=2'b00;
    ALUOp[2:0]=3'b110;
end
6'b101011:begin //sw
    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b101x0;
    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b101x1;
    PCSrc[1:0]=2'b00;
    ALUOp[2:0]=3'b000;
end
6'b100011:begin //lw
    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b10111;
    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b11001;
    PCSrc[1:0]=2'b00;
    ALUOp[2:0]=3'b000;
end
6'b000100:begin //beq
    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b100x0;
    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b100x1;
    PCSrc[1:0]=(zero==0)?2'b00:2'b01;
    ALUOp[2:0]=3'b001;
end
```

```
                6'b000101:begin //bne
                    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b100x0;
                    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b100x1;
                    PCSrc[1:0]=(zero==1)?2'b00:2'b01;
                    ALUOp[2:0]=3'b001;
                end
                6'b000001:begin //bltz
                    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b100x0;
                    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b100x1;
                    PCSrc[1:0]=(sign==0)?2'b00:2'b01;
                    ALUOp[2:0]=3'b000;
                end
                6'b000010:begin //j
                    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b1xxx0;
                    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b100xx;
                    PCSrc[1:0]=2'b10;
                    ALUOp[2:0]=3'b000;
                end
                6'b111111:begin //halt
                    {PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre}=5'b0xxx0;
                    {InsMemRW,mRD,mWR,RegDst,ExtSel}=5'b100xx;
                    PCSrc[1:0]=2'b11;
                    ALUOp[2:0]=3'b000;
                end
            endcase
        end
    endmodule
```

4. **ALU**

ALU 是组合逻辑。引脚说明：见实验内容

```
module ALU(
    input [31:0] ALUSrcA,
    input [31:0] ALUSrcB,
    input [2:0] ALUOp,
    output reg sign,
    output reg zero,
    output reg [31:0] result
    );
    always@(ALUOp or ALUSrcA or ALUSrcB)begin
        case(ALUOp)
            3'b000:result = ALUSrcA+ALUSrcB;
            3'b001:result = ALUSrcA-ALUSrcB;
            3'b010:result = ALUSrcB<<ALUSrcA;
            3'b011:result = ALUSrcA|ALUSrcB;
            3'b100:result = ALUSrcA&ALUSrcB;
            3'b101:result = (ALUSrcA<ALUSrcB)?1:0;
            3'b110:begin
                if(ALUSrcA<ALUSrcB&&(ALUSrcA[31]==ALUSrcB[31]))
                    result=1;
                else if(ALUSrcA[31]==1&&ALUSrcB[31]==0)
                    result=1;
                else
                    result=0;
            end
            3'b111:result=ALUSrcA^ALUSrcB;
        endcase
        zero=(result==0)?1:0;
        sign=result[31];
    end
endmodule
```

5. DataMemory

读数据存储器为组合逻辑，写数据存储器为时序逻辑。前者用 assign 赋值，后者在 CLK 下降沿触发写入。引脚说明：见实验内容

```
module DataMemory(
    input CLK,
    input RD,
    input WR,
    input [31:0] DAddr,
    input [31:0] DataIn,
    output [31:0] DataOut
    );
    reg [7:0] ram [0:99];
    assign DataOut[7:0]=(RD==1)?ram[DAddr+3]:8'bz;
    assign DataOut[15:8]=(RD==1)?ram[DAddr+2]:8'bz;
    assign DataOut[23:16]=(RD==1)?ram[DAddr+1]:8'bz;
    assign DataOut[31:24]=(RD==1)?ram[DAddr]:8'bz;
    always@(negedge CLK)begin
        if(WR==1)begin
            ram[DAddr]<=DataIn[31:24];
            ram[DAddr+1]<=DataIn[23:16];
            ram[DAddr+2]<=DataIn[15:8];
            ram[DAddr+3]<=DataIn[7:0];
        end
    end
endmodule
```

6. **ExtendUnit**

执行立即数扩展或符号数扩展，引脚说明：

ExtSel，当 ExtSel 为 0 时执行立即数扩展，1 执行符号数扩展。

```
module ExtendUnit(
    input ExtSel,
    input [15:0] ExtIn,
    output reg [31:0] ExtOut
    );
    always@(*)begin
        ExtOut[15:0]=ExtIn;
        if(ExtSel==0)begin
            ExtOut[31:16]=16'h0000;
        end
        else begin
            if(ExtIn[15]==0) begin
                ExtOut[31:16]=16'h0000;
            end
            else begin
                ExtOut[31:16]=16'hFFFF;
            end
        end
    end
endmodule
```

7. 多路选择器

MUX_nextPC:

```
module MUX_4_32bits(
    input [1:0] select,
    input [31:0] in0,
    input [31:0] in1,
    input [31:0] in2,
    input [31:0] in3,
    output reg [31:0] out
    );
    always@(select or in0 or in1 or in2 or in3)begin
        case(select)
            2'b00:out=in0;
            2'b01:out=in1;
            2'b10:out=in2;
            2'b11:out=in3;
        endcase
    end
endmodule
```

MUX_WriteReg:

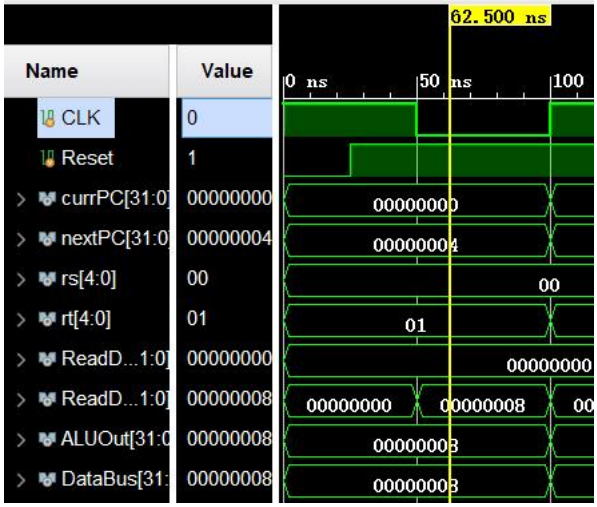
```
module MUX_2_5bits(
    input Enable,
    input [4:0] in0,
    input [4:0] in1,
    output [4:0] out
    );
    assign out = (Enable==0)?in0:in1;
endmodule
```

MUX_WriteData、MUX_ALU_inA、MUX_ALU_inB:

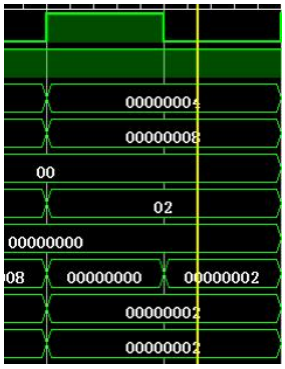
```
module MUX_2_32bits(
    input Enable,
    input [31:0] in0,
    input [31:0] in1,
    output [31:0] out
    );
    assign out = (Enable==0)?in0:in1;
endmodule
```

这里的三种情况共用一个多路选择器，以节省代码工作量，实际是该有三个不同位置的同功能多路选择器。

实验指令：
1) addiu $1, $0, 8

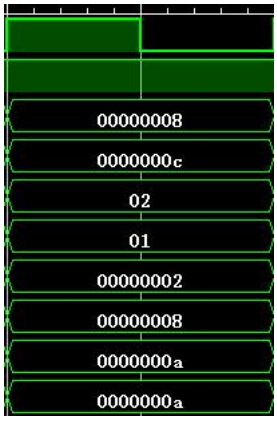


寄存器 0 的值为 0，寄存器 1 的值更新为（0+8）= 8。
2) ori $2, $0, 2



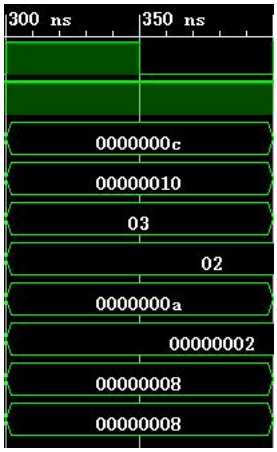
寄存器 0 的值为 0，寄存器 2 的值更新为（00 or 10）= 10b = 2

3) add $3, $2, $1



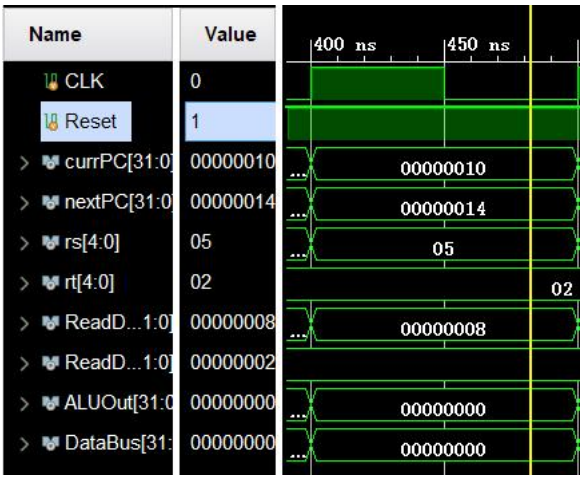
寄存器 1 的值为 8，寄存器 2 的值为 2，寄存器 3 的值更新为（2+8）= 10。

4) sub $5, $3, $2



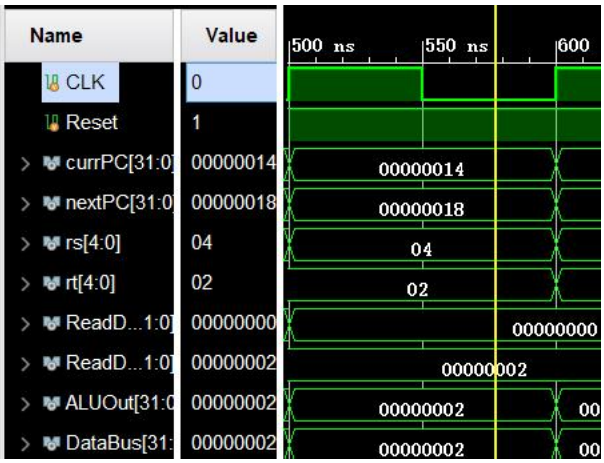
寄存器 3 的值为 10，寄存器 2 的值为 2，寄存器 5 的值更新为（10 - 2）= 8。

5) and $4, $5, $2



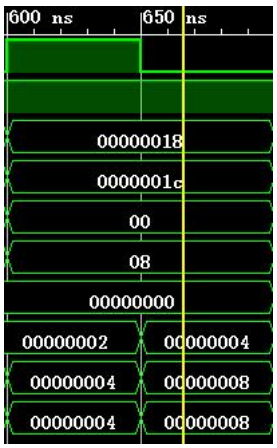
寄存器 2 的值为 2，寄存器 5 的值为 8，寄存器 4 的值更新为（1000 and 0010）= 0。

6) or $8, $4, $2



寄存器 4 的值为 0，寄存器 2 的值为 2，寄存器 8 的值更新为（00 or 10）= 10b = 2。

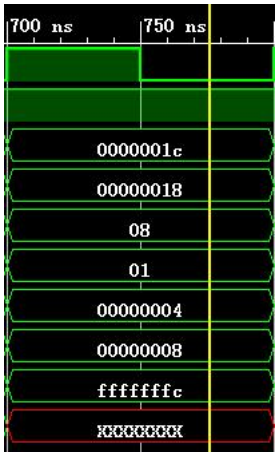
7) sll $8, $8, 1



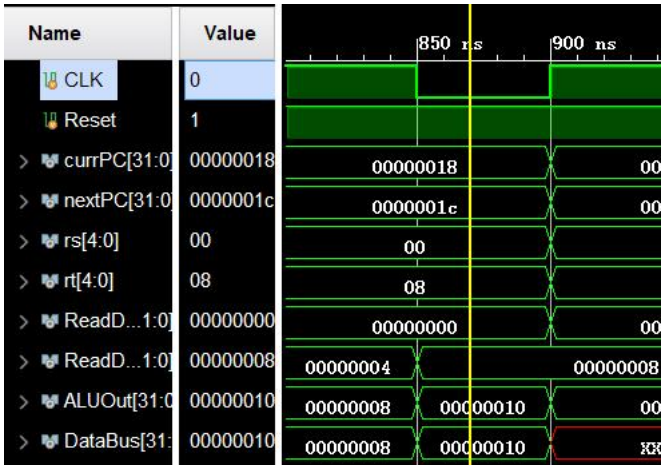
sa = 1，rd 和 rt 都为 8，即左移寄存器 8 内的内容 1 位，由 4 左移为 8，但是需要在指令执

行后才左移。

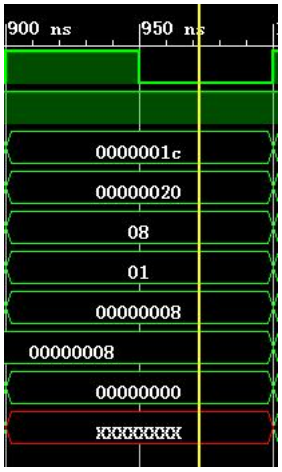
8) bne $8, $1, -2 （≠）



由于寄存器 8 内的值为 4（左移指令后的结果还未写入寄存器 8），而寄存器 1 的值为 8，令 rs ≠ rt，所以下一条地址为
nextPC + sign_extend(offset)<<2 = 00000018 而不是 00000020。
9) sll $8, $8, 1

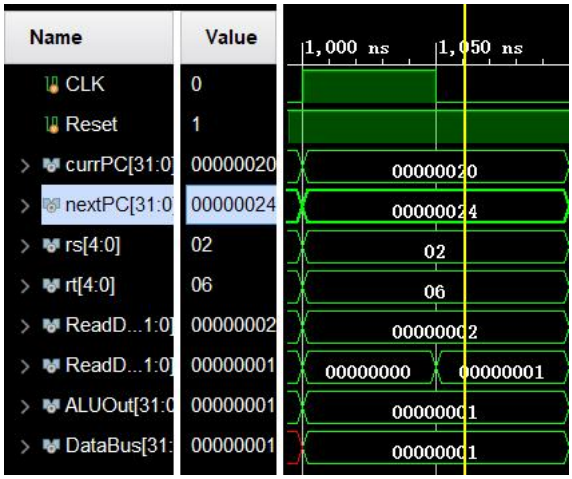


sa = 1，rd 和 rt 都为 8，即左移寄存器 8 内的内容 1 位，由 8 左移为 16，即 10h，但是需
要在指令执行后才左移。
10) bne $8, $1, -2　(=)



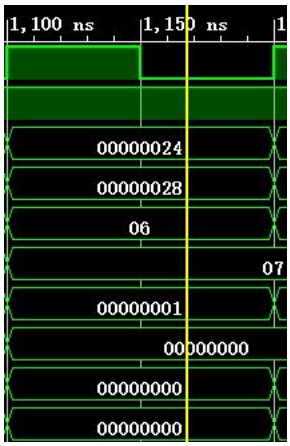
由于寄存器 8 内的值为 8（判断时左移指令结果未写入寄存器 8），而寄存器 1 的值为 8，令 rs = rt，所以正常执行，下条指令为 00000020。

11) slti $6, $2, 4



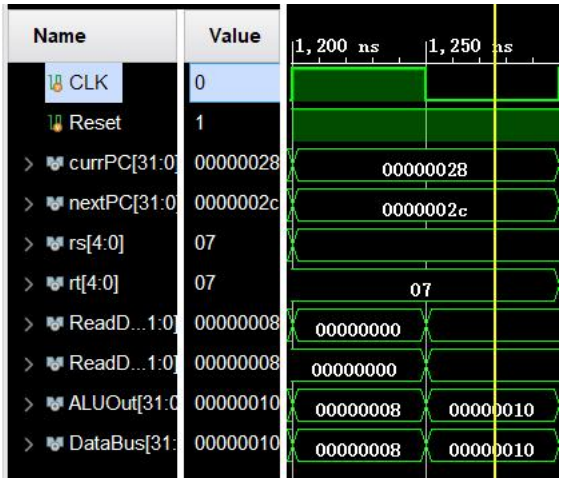
比较立即数是否比寄存器 2 内的数值大，结果为 4>2，令寄存器 6 写入数值 1。

12) slti $7, $6, 0



比较立即数是否比寄存器 6 内的数值大，结果为 1>0，令寄存器 7 写入数值 0。

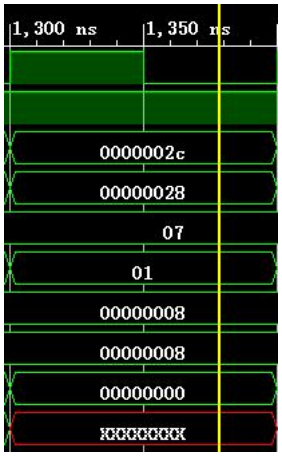
13) addiu $7, $7, 8



寄存器 7 的值为 0，寄存器 7 的值更新为（8+8）= 16 = 10h，因为此时 RD1 在执行指令

前为 0，执行指令后为 8，而 RD2 为立即数 8，所以 ALUout 为两个立即数。

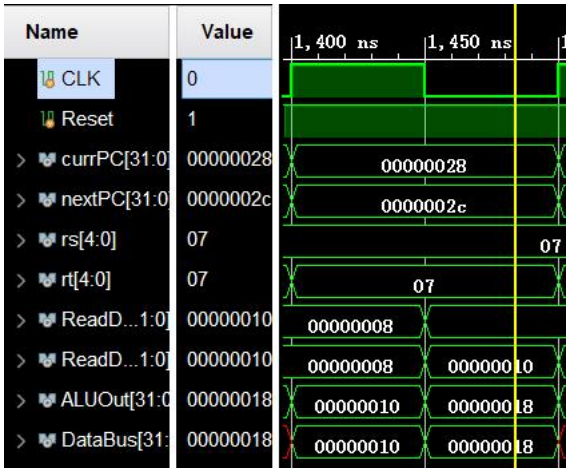
14) beq $7, $1, 2 （=）



由于寄存器 7 内的值为 8（加法指令后的结果还未写入寄存器 7），而寄存器 1 的值为 8，令 rs = rt，所以下一条地址为

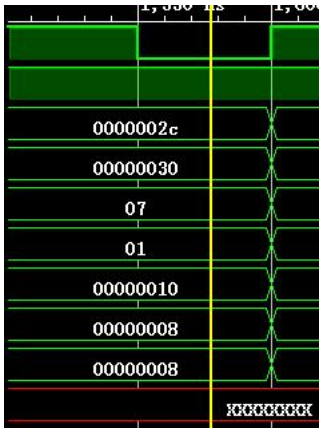
nextPC + sign_extend(offset)<<2 = 0000002c 而不是 00000030。

15) addiu $7, $7, 8



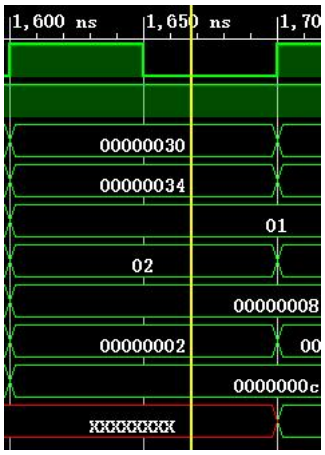
寄存器 7 的值为 16，寄存器 7 的值更新为（16+8）= 24 = 18h。

16) beq $7, $1, 2 （≠）



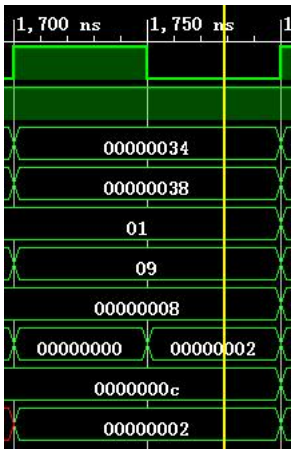
由于寄存器 7 内的值为 10（判断时加法指令结果未完全写入寄存器 7），而寄存器 1 的值为 8，令 rs = rt，所以下一条地址为 00000030。

17) sw $2, 4($1)



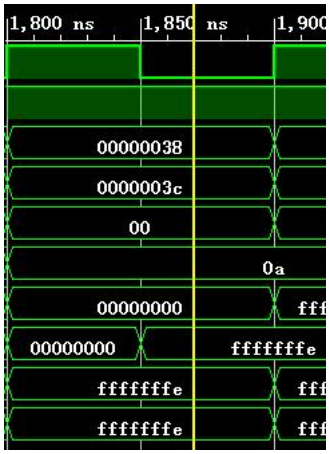
ReadData1 寄存器 1 数值 = 00000008, ReadData2 = 寄存器 2 数值 = 00000002
所以 A = rs，B = immediate = 00000004，所以
ALUout = A + B = 0000000c，DB= 0000000c，并将 12 写入寄存器 2。

18) lw $9, 4($1)



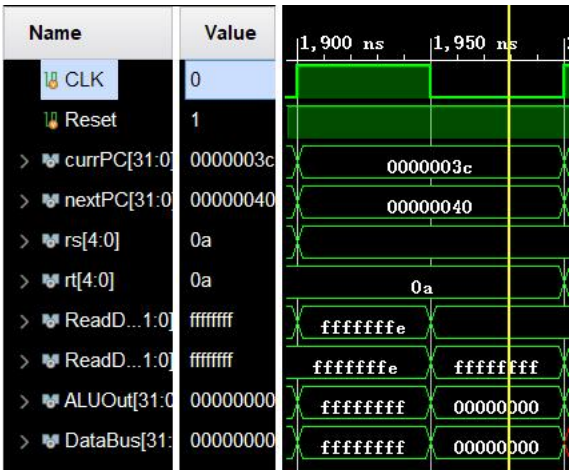
ReadData1 = 寄存器 1 数值 = 00000008，ReadData2 = 寄存器 9
所以 A = rs，B = immediate = 00000004，所以
ALUout = A + B = 0000000c，DB= 00000002，并将 12 写入寄存器 9。

19) addiu $10, $0, -2



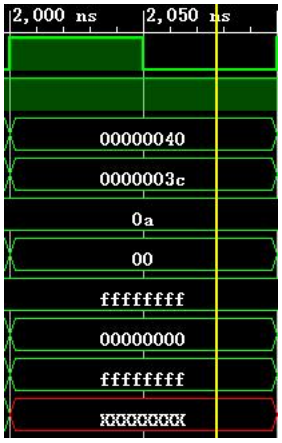
寄存器 0 的值为 0，寄存器 10 的值更新为（0-2）= -2 = fffffffe h 补。

20) addiu $10, $10, 1



寄存器 10 的值为-2，寄存器 10 的值更新为（-1+（-1））= 0，因为此时 RD1 在执行指令前为-2，执行指令后为-1，而 RD2 为立即数 1。

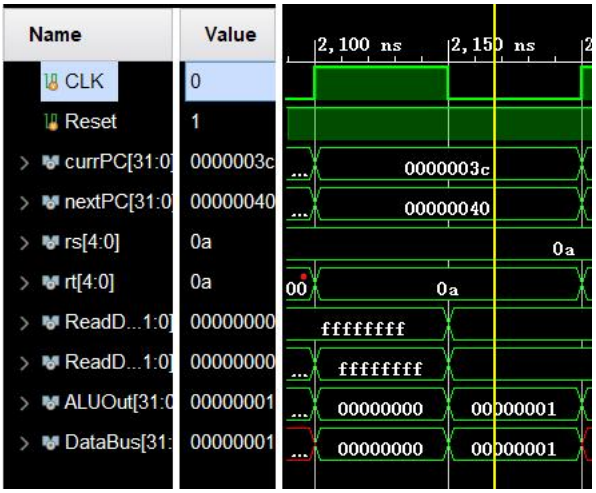
21) bltz $10, 2(>0)



由于寄存器 10 内的值为-1（判断时加法指令结果未完全写入寄存器 10）比 0 小，令 rs < 0，所以下一条地址为

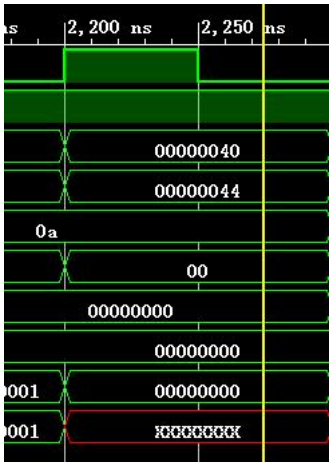
nextPC + sign_extend(offset)<<2 = 0000003c 而不是 00000044。

22) addiu $10, $10, 1



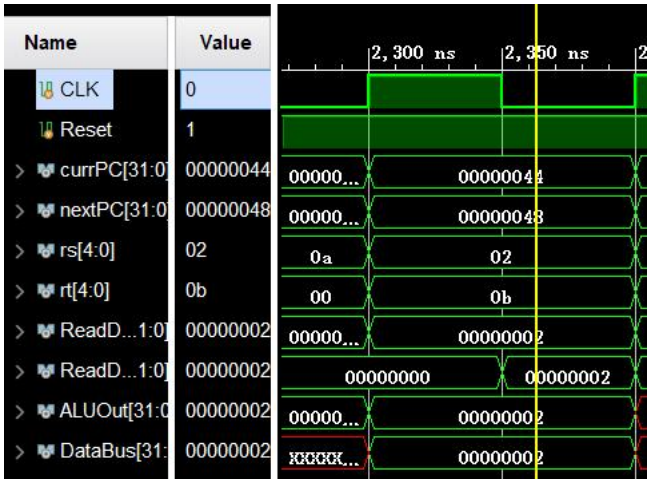
寄存器 10 的值为 0，寄存器 10 的值更新为（0 +1）= 1。

23) bltz $10, 2(>0)



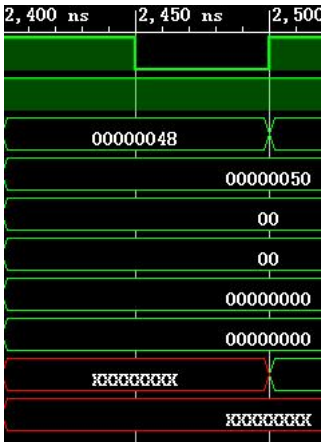
由于寄存器 10 内的值为 0（判断时加法指令结果未完全写入寄存器 10）不比 0 小（只有小于 0 才需要跳回以前的指令），令 0 = rs，所以下一条地址为 00000044。

24) andi $11, $2, 2



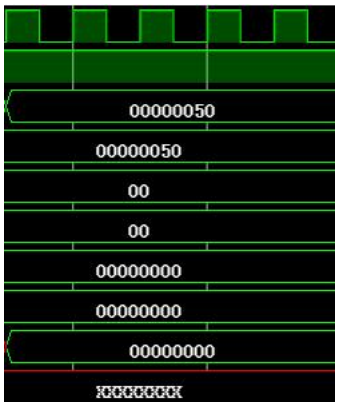
寄存器 2 的值为 2，寄存器 11 的值更新为（2 and 2）= 10b = 2。

25) j 0x00000050



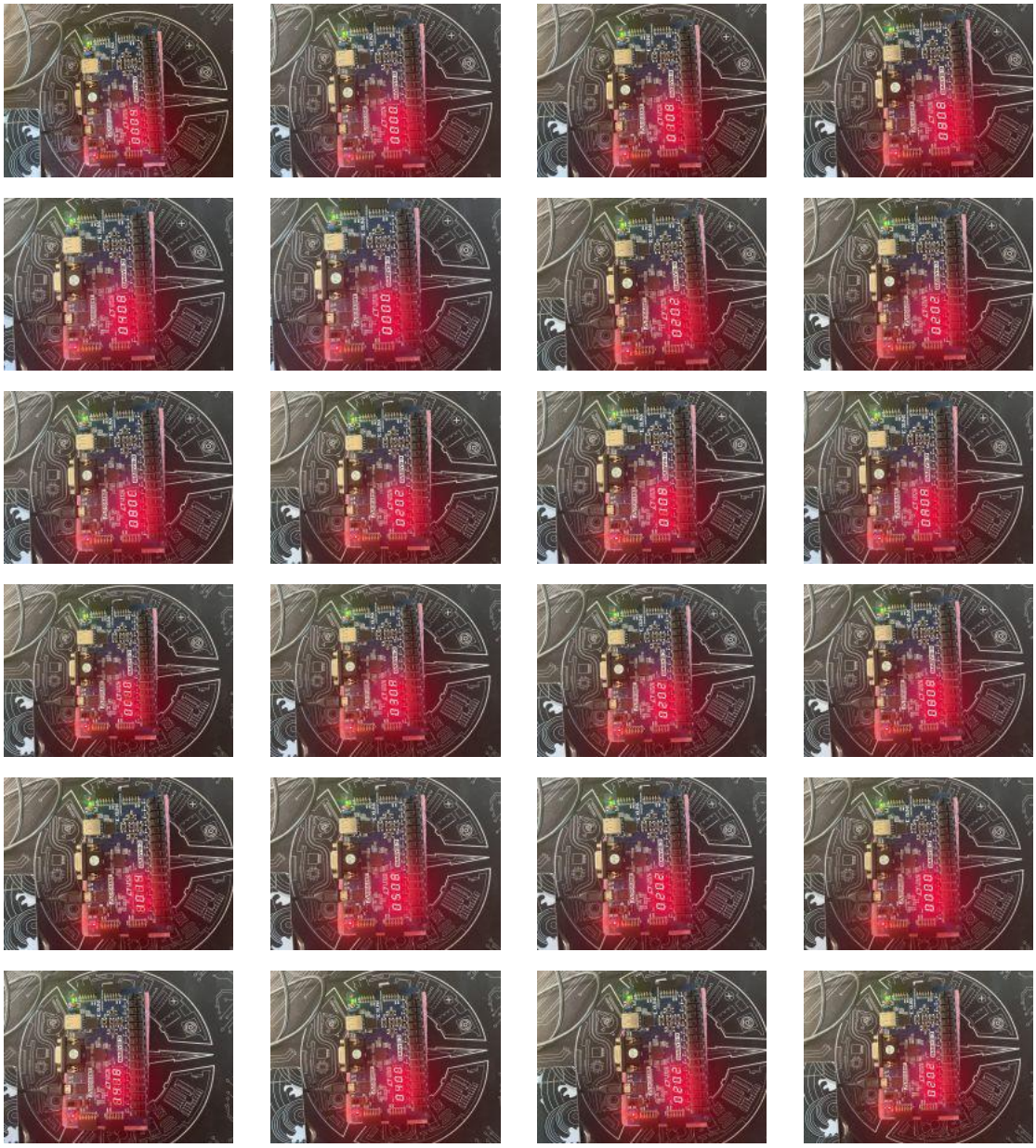
这里下条指令无条件跳转至 50h 指令，没有其他输入，所以 rs，rt，ALUout 和 DB 无值。

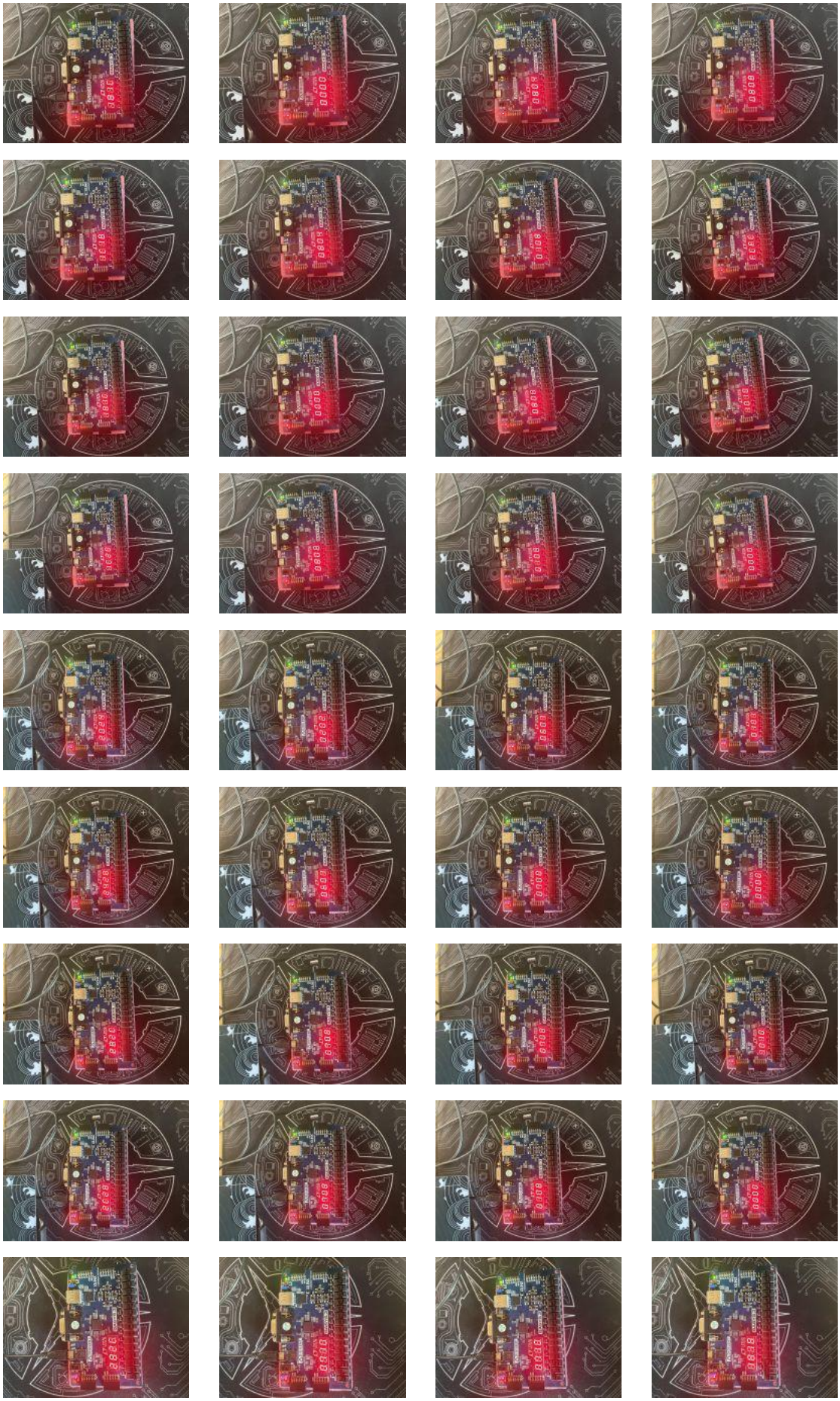
26) halt

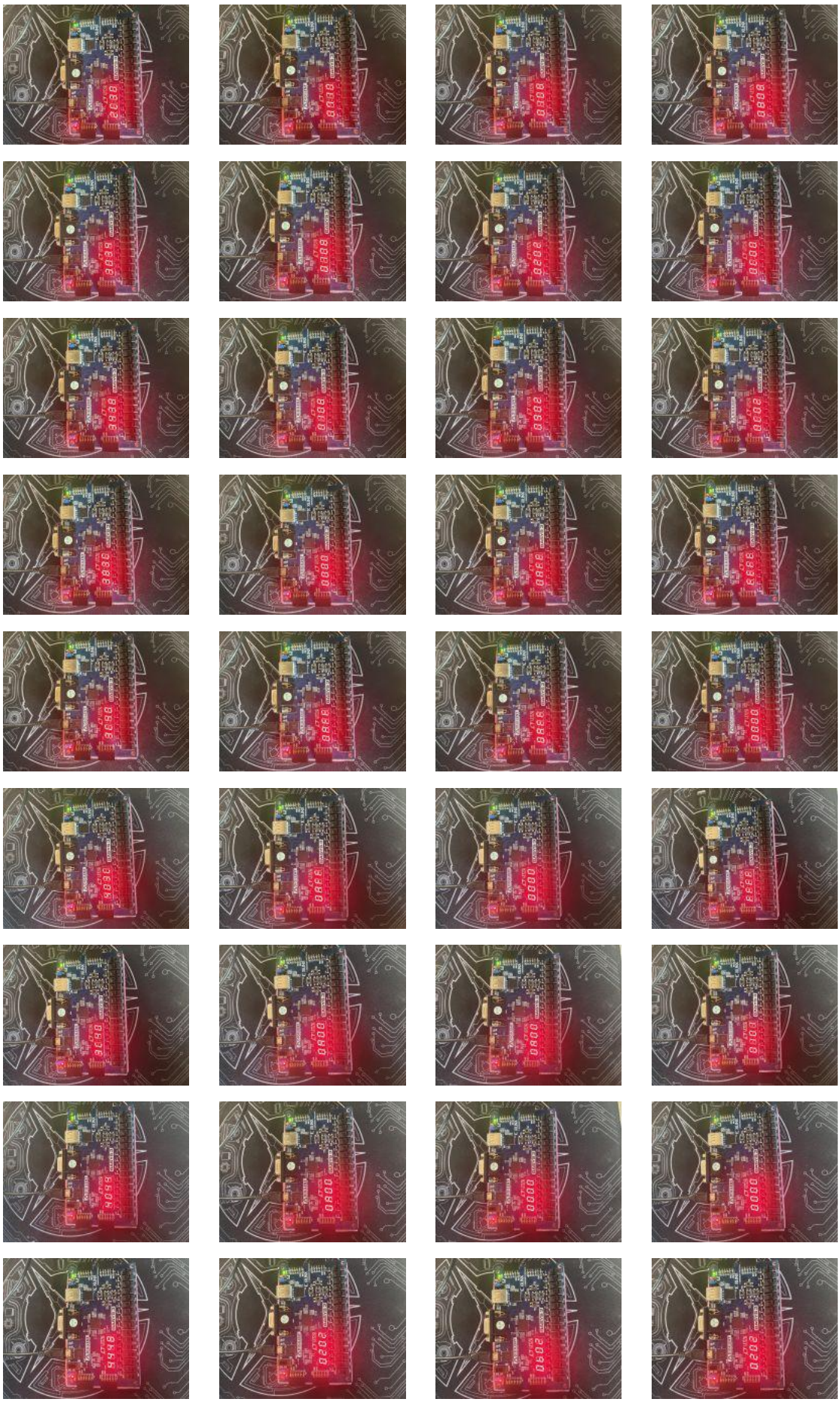


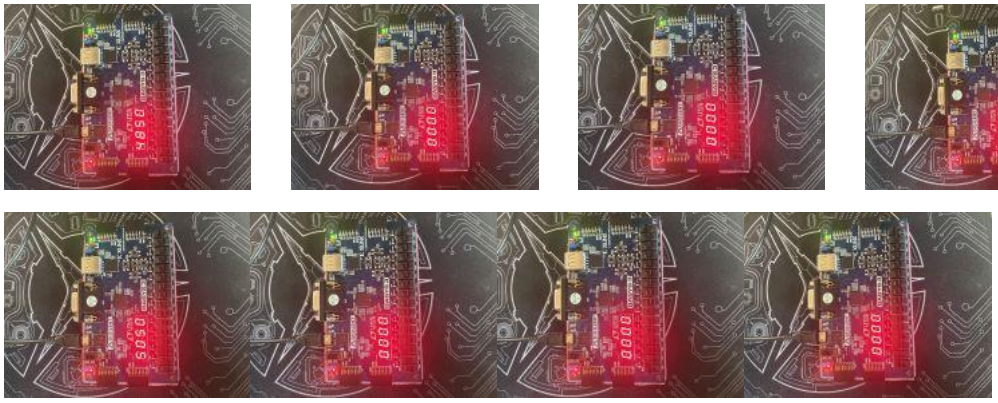
停机指令，之后再输入任何时钟也不运作。

实图（从左至右 一：左为当前 PC，右为下一条 PC 二：选中 rs 寄存器和对应的值 三：选中 rt 寄存器和对应的值 四：ALU 输出和总线数值）









六、实验心得体会

此实验是我接触并使用 verilog 语言完成的第一个实验。在此次实验中，我主要遇到了两方面的困难：一是需要根据资料从零开始建构 MIPS 单周期 CPU，二是掌握 verilog 语言的简单语法及变量使用方法。

为解决第一方面的困难，我询问了学长学姐以及助教前辈，决定采用模块化的方式进行 CPU 的架构，在模块实例化的过程中，我认为最难的部分是处理各种控制信号间的关系，因此我在 ControlUnit 部分花了最长的时间，在设计 ControlUnit 时，我也出现了混淆变量名（例如 PCWre 和 PCSrc）的情况。烧板阶段，出现了需要按住 CLK 同时将 Reset 置为高电平才能为第一条指令提供一个时钟下降沿的情况，后来询问同学，得知可以将信号取反纠正；

为解决第二方面的困难，在此实验前我预先完成了 display_7seg 的实验，初步了解了 verilog 语言的语法，在实验过程中出现 bug 询问助教前辈后学会了给予单独仿真文件进行 debug，同时也会询问同学和学长学姐进行帮助，最终得以完成实验并通过验收。

这次实验让我实在掌握了单周期 CPU 的设计原理以及 MIPS 指令集的内容，为理论课程提供了很大的助益。非常感谢帮助过我的老师、助教前辈和同学们。