



《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）：18 软件工程 2 班

时间：2019 年 11 月 20 日

组 员：贺恩泽 17364025

陈志远 17338020

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)	reserved
--------	---------	---------	---------	----------

功能: $rd \leftarrow rs + rt$ 。**reserved** 为预留部分，即未用，一般填“0”。

(2) sub rd , rs , rt

000001	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能: $rd \leftarrow rs - rt$ 。

(3) addiu rt , rs ,immediate

000010	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能: $rt \leftarrow rs + (\text{sign-extend})immediate$ ；**immediate** 符号扩展再参加“加”运算。

==> 逻辑运算指令

(4) andi rt , rs ,immediate

010000	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能: $rt \leftarrow rs \& (\text{zero-extend})immediate$ ；**immediate** 做“0”扩展再参加“与”运算。

(5) and rd , rs , rt

010001	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能: $rd \leftarrow rs \& rt$ ；逻辑与运算。

(6) ori rt , rs ,immediate

010010	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能: $rt \leftarrow rs | (\text{zero-extend})immediate$ ；**immediate** 做“0”扩展再参加“或”运算。

(7) or rd , rs , rt

010011	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能: $rd \leftarrow rs | rt$ ；逻辑或运算。

==>移位指令

(8) sll rd, rt, sa

011000	未用	rt(5 位)	rd(5 位)	sa(5 位)	reserved
--------	----	---------	---------	---------	----------

功能: $rd \leftarrow rt \ll (\text{zero-extend})sa$, 左移 sa 位, (zero-extend)sa。**==>比较指令**(9) slti rt, rs, **immediate** 带符号数

011100	rs(5 位)	rt(5 位)	immediate (16 位)
--------	---------	---------	-------------------------

功能: if (rs < (sign-extend)**immediate**) rt = 1 else rt = 0, 具体请看表 2 ALU 运算功能表, 带符号。**==> 存储器读/写指令**(10) sw rt, **immediate**(rs) 写存储器

100110	rs(5 位)	rt(5 位)	immediate (16 位)
--------	---------	---------	-------------------------

功能: $\text{memory}[rs + (\text{sign-extend})\text{immediate}] \leftarrow rt$; **immediate** 符号扩展再相加。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。(11) lw rt, **immediate**(rs) 读存储器

100111	rs(5 位)	rt(5 位)	immediate (16 位)
--------	---------	---------	-------------------------

功能: $rt \leftarrow \text{memory}[rs + (\text{sign-extend})\text{immediate}]$; **immediate** 符号扩展再相加。即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数, 然后保存到 rt 寄存器中。**==> 分支指令**(12) beq rs, rt, **immediate**

110000	rs(5 位)	rt(5 位)	immediate (16 位)
--------	---------	---------	-------------------------

功能: if (rs == rt) $pc \leftarrow pc + 4 + (\text{sign-extend})\text{immediate} \ll 2$ else $pc \leftarrow pc + 4$

特别说明: **immediate** 是从 PC+4 地址开始和转移到的指令之间指令条数。**immediate** 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 **immediate** 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, **immediate**

110001	rs(5 位)	rt(5 位)	immediate (16 位)
--------	---------	---------	-------------------------

功能: if (rs != rt) $pc \leftarrow pc + 4 + (\text{sign-extend})\text{immediate} \ll 2$ else $pc \leftarrow pc + 4$

特别说明: 与 beq 不同点是, 不等时转移, 相等时顺序执行。

(14) bltz rs, **immediate**

110010	rs(5 位)	00000	immediate (16 位)
--------	---------	-------	-------------------------

功能: if (rs < \$zero) $pc \leftarrow pc + 4 + (\text{sign-extend})\text{immediate} \ll 2$ else $pc \leftarrow pc + 4$ 。**==>跳转指令**

(15) j addr

111000	addr[27:2]				
--------	------------	--	--	--	--

功能： $pc \leftarrow \{(pc+4)[31:28], addr[27:2], 2'b00\}$ ，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	00000000000000000000000000000000(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：

- (1) 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。
- (2) 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。
- (3) 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。
- (4) 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。
- (5) 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	2625	2120	1615	1110	65	0
op	rs	rt	rd	sa	funct	
6 位	5 位	5 位	5 位	5 位	6 位	

I 类型:

31	2625	2120	1615	0
op	rs	rt	immediate	
6 位	5 位	5 位	16 位	

J 类型:

31	2625	0
op	address	
6 位	26 位	

其中,

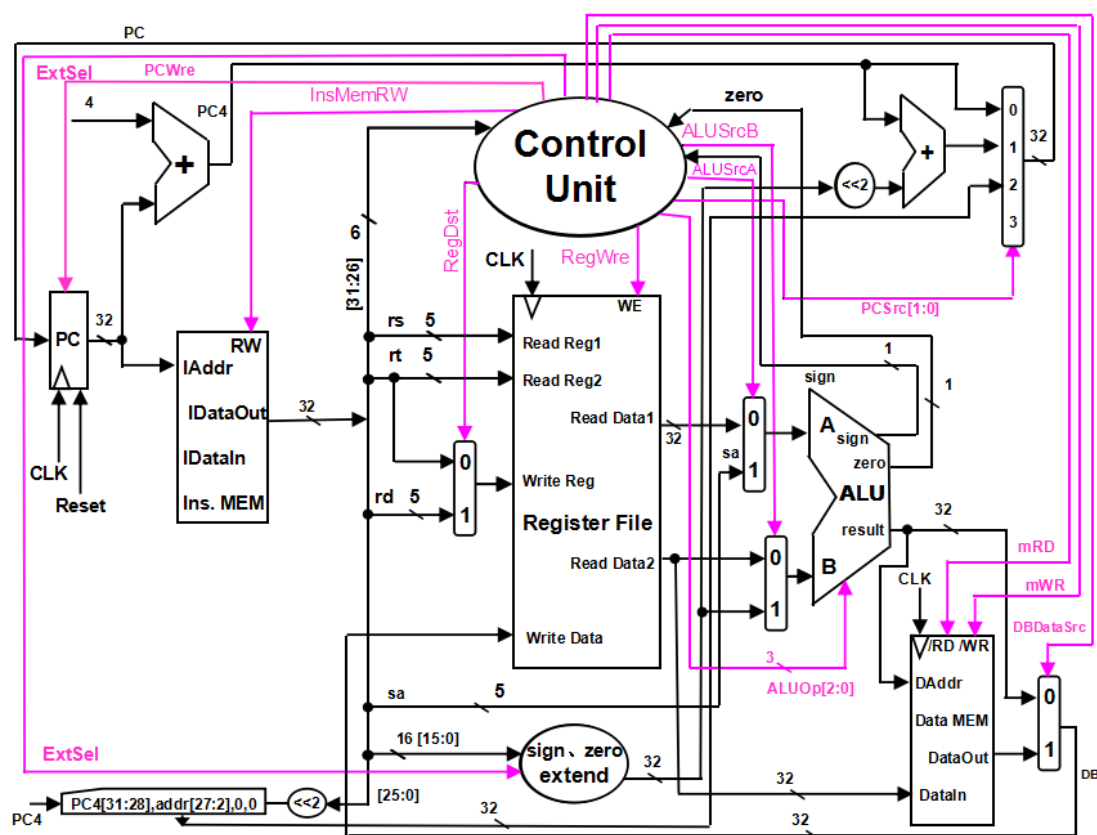
op: 为操作码;**rs:** 只读。为第 1 个源操作数寄存器, 寄存器地址 (编号) 是 00000~11111, 00~1F;**rt:** 可读可写。为第 2 个源操作数寄存器, 或目的操作数寄存器, 寄存器地址 (同上);**rd:** 只写。为目的操作数寄存器, 寄存器地址 (同上);**sa:** 为位移量 (shift amt), 移位指令用于指定移多少位;**funct:** 为功能码, 在寄存器类型指令中 (R 类型) 用来指定指令的功能与操作码配合使用;**immediate:** 为 16 位立即数, 用作无符号的逻辑操作数、有符号的算术操作数、数据加载 (Load) / 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：andi、ori	(sign-extend)immediate (符号扩展)，相关指令：addiu、slti、sw、lw、beq、bne、bltz
PCSrc[1..0]	00: $pc \leftarrow pc+4$ ，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: $pc \leftarrow pc+4+(sign\text{-}extend)immediate \ll 2$ ，相关指令： beq(zero=1)、 bne(zero=0)、bltz(sign=1)； 10: $pc \leftarrow \{(pc+4)[31:28],addr[27:2],2'b00\}$ ，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	$Y = B \ll A$	B 左移 A 位
011	$Y = A \vee B$	或
100	$Y = A \wedge B$	与
101	$Y = (A < B) ? 1 : 0$	比较 A<B 不带符号
110	$Y = (((A < B) \& \& (A[31] == B[31])) \vee (((A[31] == 1 \& \& B[31] == 0))) ? 1 : 0$	比较 A<B 带符号
111	$Y = A \oplus B$	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，关于 ALU 设计、存储器设计、寄存器组设计等等，也是必须认真考虑的问题。

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

单周期CPU的设计是根据数据通路图来进行设计，根据数据通路图中每个模块的输入输出，设计和实现对应的模块。

验证设计的CPU是否正确，可以通过

1. 观察所设计CPU的数据通路是否与原理数据通路图一致。
2. 通过仿真，让CPU读取并执行预先定义的指令，对比CPU各模块输出与理论输出是否一致。
3. 将设计烧板并在真实机器上运行，观察输出对比理论输出进行验证。

CPU设计

一、模块设计

a. 指令存储器

指令存储器存储了需要执行的指令和地址，本模块将从一个存储着指令的文件当中读入指令。

该模块的输入为指令读写信号 insRW、指令地址 address。

```
module InstructionMemory(
    input insRW,
    input [31:0] address,
    output reg [31:0] dataOut);

    reg [7:0] memory[0:127];
    initial
        $readmemb("memory.txt", memory);

    always @(*) begin
        if (insRW) begin
            dataOut[31:24] = memory[address];
            dataOut[23:16] = memory[address + 1];
        end
    end
endmodule
```



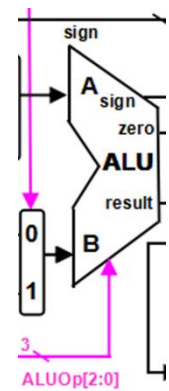
```
dataOut[15:8] = memory[address + 2];
dataOut[7:0] = memory[address + 3];
end
end
endmodule
```

b. 算术和逻辑运算单元

该模块是一个32位的ALU单元，会根据控制信号对输入的操作数进行不同的运算，例如加、减、与、或等。

输入为：操作数 rega、操作数 regb 和操作符 op；

输出包括运算结果 result、零标志 zero 和符号标志 sign。



ALU有以下功能：

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	$Y=A \vee B$	或
100	$Y=A \wedge B$	与
101	$Y=(A<B)?1:0$	比较A<B 不带符号
110	$Y=((A<B)\&\&(A[31]==B[31])) \vee ((A[31]==1\&\&B[31]==0)))?1:0$	比较A<B 带符号
111	$Y=A \oplus B$	异或

```
module ALU(
    input [31:0] rega,
    input [31:0] regb,
    input [2:0] op,
    output reg [31:0] result,
    output zero,
    output sign);

    always @(op or rega or regb) begin
        case (op)
            `ALU_OP_ADD: result = (rega + regb);
            `ALU_OP_SUB: result = (rega - regb);
```

```

        `ALU_OP_SLL: result = (regb << rega);
        `ALU_OP_OR: result = (rega | regb);
        `ALU_OP_AND: result = (rega & regb);
        `ALU_OP_LT: result = (rega < regb) ? 1 : 0;
        `ALU_OP_SLT: result = (((rega < regb) && (reg
a[31] == regb[31])) || ((rega[31] && !regb[31]))) ? 1 : 0
;
        `ALU_OP_XOR: result = (rega ^ regb);
        default: result = 0;
    endcase
end

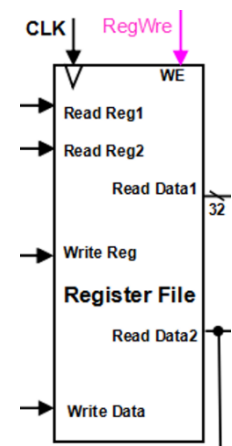
assign zero = (result == 0) ? 1 : 0;
assign sign = result[31];
endmodule

```

c. 寄存器组

该模块为一个32位而拥有32个寄存的寄存器组。

输入为：时钟信号 clk、写使能信号 we（为1时在时钟边沿触发写入）、重置信号 rst、rs寄存器地址 readReg1、rt 寄存器地址 readReg2、将数据写入的寄存器 writeReg、写入寄存器的数据 writeData；
输出为读出的寄存器rs和rt的数据 readData1 和 readData2。



```

module Register(
    input clk,
    input we,
    input rst,
    input [4:0] readReg1,
    input [4:0] readReg2,
    input [4:0] writeReg,
    input [31:0] writeData,
    output [31:0] readData1,
    output [31:0] readData2);

    reg [31:0] register[1:31];
    integer i;

    assign readData1 = readReg1 == 0 ? 0 : register[readR
eg1];

```

```

    assign readData2 = readReg2 == 0 ? 0 : register[readReg2];

    always @(negedge clk or negedge rst) begin
        if (!rst) begin
            for (i = 1; i < 32; i = i + 1) begin
                register[i] = 0;
            end
        end
        else if (we && writeReg) begin
            register[writeReg] <= writeData;
        end
    end
endmodule

```

d. 存储器

该存储器模块存储单元为8位二进制数长度，由时钟边沿触发写。

输入为：时钟信号 clk、数据存储器地址输入端口 dAddr、数据存储器数据输入端口 dataIn、读控制信号 rd、写控制信号 wr;

```

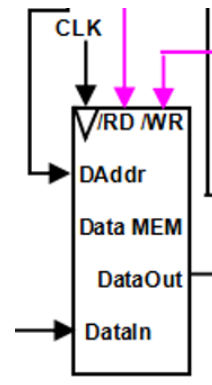
module DataMemory(
    input clk,
    input [31:0] dAddr,
    input [31:0] dataIn,
    input rd,
    input wr,
    output [31:0] dataOut);

    reg [7:0] memory [0:68];

    assign dataOut[7:0] = rd ? memory[dAddr + 3] : 8'bz;
    assign dataOut[15:8] = rd ? memory[dAddr + 2] : 8'bz;
    assign dataOut[23:16] = rd ? memory[dAddr + 1] : 8'bz;
    assign dataOut[31:24] = rd ? memory[dAddr] : 8'bz;

    always@(negedge clk) begin
        if (wr) begin
            memory[dAddr] <= dataIn[31:24];
            memory[dAddr + 1] <= dataIn[23:16];
            memory[dAddr + 2] <= dataIn[15:8];
            memory[dAddr + 3] <= dataIn[7:0];
        end
    end
endmodule

```



```

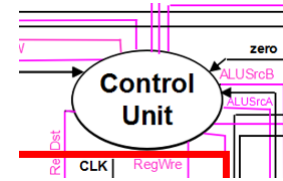
        end
    end
endmodule

```

e. CPU 控制单元

该模块根据控制信号解码出指令并控制各模块使其正常工作。

控制信号与指令对应关系如下：



Op	Ins	pcWre	aluSrcA	aluSrcB	regDataSrc	regWre	insRW
000000	add	1	0	0	0	1	1
000001	sub	1	0	0	0	1	1
000010	addiu	1	0	1	0	1	1
010000	andi	1	0	1	0	1	1
010001	and	1	0	0	0	1	1
010010	ori	1	0	1	0	1	1
010011	or	1	0	0	0	1	1
011000	sll	1	1	0	0	1	1
011100	slti	1	0	1	0	1	1
100110	sw	1	0	1	X	0	1
100111	lw	1	0	1	1	1	1
110000	beq	1	0	0	X	0	1
110001	bne	1	0	0	X	0	1
110010	bltz	1	0	0	X	0	1
111000	j	1	X	X	X	0	1
111111	halt	0	X	X	X	0	1
Op	Ins	rd	wr	regDst	extSel	pcSrc	aluOp
000000	add	0	0	1	X	00	000
000001	sub	0	0	1	X	00	001
000010	addiu	0	0	0	1	00	000
010000	andi	0	0	0	0	00	100
010001	and	0	0	1	X	00	100

010010	ori	0	0	0	0	00	011
010011	or	0	0	1	X	00	011
011000	sll	0	0	1	X	00	010
011100	slti	0	0	0	1	00	110
100110	sw	0	1	X	1	00	000
100111	lw	1	0	0	1	00	000
110000	beq	0	0	X	1	00(zero=0) 01(zero=1)	001
110001	bne	0	0	X	1	01(zero=0) 00(zero=1)	001
110010	bltz	0	0	X	1	00(sign=0) 01(sign=1)	001
111000	j	0	0	X	X	10	010
111111	halt	0	0	X	X	11	XXX

代码实现：

```

module ControlUnit(
    input [5:0] op,
    input zero,
    input sign,
    output pcWre,
    output aluSrcA,
    output aluSrcB,
    output regDataSrc,
    output regWre,
    output insRW,
    output rd,
    output wr,
    output regDst,
    output extSel,
    output [1:0] pcSrc,
    output [2:0] aluOp);

    assign pcWre = (op == `OP_HALT) ? 0 : 1;

    assign aluSrcA = (op == `OP_SLL) ? 1 : 0;

```

```

    assign aluSrcB = (op == `OP_ADDIU || op == `OP_ANDI ||
    op == `OP_ORI || op == `OP_SLTI || op == `OP_SW || op ==
    `OP_LW) ? 1 : 0;

    assign regDataSrc = (op == `OP_LW) ? 1 : 0;
    assign regWre = (op == `OP_SW || op == `OP_BEQ || op ==
    `OP_BNE || op == `OP_BLTZ || op == `OP_J || op == `OP_HA
    LT) ? 0 : 1;

    assign insRW = 1;

    assign rd = (op == `OP_LW) ? 1 : 0;
    assign wr = (op == `OP_SW) ? 1 : 0;

    assign regDst = (op == `OP_ADD || op == `OP_SUB || op
    == `OP_AND || op == `OP_OR || op == `OP_SLL) ? 1 : 0;
    assign extSel = (op == `OP_ANDI || op == `OP_ORI) ? 0
    : 1;

    assign pcSrc[1] = (op == `OP_HALT || op == `OP_J) ? 1
    : 0;
    assign pcSrc[0] = (op == `OP_HALT || (op == `OP_BEQ &&
    zero) || (op == `OP_BNE && !zero) || (op == `OP_BLTZ && s
    ign)) ? 1 : 0;

    assign aluOp[2] = (op == `OP_ANDI || op == `OP_AND ||
    op == `OP_SLTI) ? 1 : 0;
    assign aluOp[1] = (op == `OP_ORI || op == `OP_OR || op
    == `OP_SLL || op == `OP_SLTI || op == `OP_J) ? 1 : 0;
    assign aluOp[0] = (op == `OP_SUB || op == `OP_ORI || o
    p == `OP_OR || op == `OP_BEQ || op == `OP_BNE || op == `OP
    _BLTZ) ? 1 : 0;

endmodule

```

f. 扩展模块

该模块实现符号位扩展或零扩展。

输入为：扩展方式选择信号 extSel (0为0扩展，1为符号位扩展)、立即数 immediate。

```

module Extend(
    input [15:0] immediate,
    input extSel,

```

```
output [31:0] extImmediate);

assign extImmediate[15:0] = immediate[15:0];
assign extImmediate[31:16] = (extSel && immediate[15])
? 16'hFFFF : 16'h0000;

endmodule
```

g. 程序计数器

在时钟上升沿处给出下条指令的地址，或在重置信号下降沿处将PC归零。
输入为：时钟信号 clk、重置信号 rst、PC 更改信号 pcWre (为 0 表示不更改 PC) 和新 PC 指令 newPC。

程序计数器的真值表如下：

Rst	PCWre	PCSrc	NewPC
0	X	XX	0
1	0	XX	PC
1	1	00 (Next)	PC+4
1	1	01 (RelJump)	PC+4+(Immediate<<2)
1	1	10 (AbsJump)	{(PC+4)[31:28],Address[27:2],00}
1	1	11 (HALT)	PC

而因PC的下一条指令可能是当前PC+4，也可能是跳转指令地址，还有可能因为停机而不变。

因此还需要设计一个选择器来选择下一条指令地址的计算方式，为此创建了 JumpPCHelper 用于计算 j 指令的下一条 PC 地址，和 NextPCHelper 用于根据指令选择不同的计算方式，具体指令地址的计算参考上述真值表内容。

```
module PC(
    input clk,
    input rst,
    input pcWre,
    input [31:0] newPC,
    output reg [31:0] address);

    initial
        address = 0;
```

```
always @(posedge clk or negedge rst)
begin
    if (!rst) begin
        address <= 32'hFFFFFFFC;
    end
    else if (pcWre || !newPC) begin
        address <= newPC;
    end
end
endmodule

module JumpPCHelper(
    input [31:0] pc,
    input [25:0] address,
    output reg [31:0] jumpPC);

    wire [27:0] tmp;
    assign tmp = address << 2; // address * 4

    always @(*) begin
        jumpPC[31:28] = pc[31:28];
        jumpPC[27:2] = tmp[27:2];
        jumpPC[1:0] = 0;
    end
endmodule

module NextPCHelper(
    input rst,
    input [1:0] pcSrc,
    input [31:0] pc,
    input [31:0] immediate,
    input [31:0] jumpPC,
    output reg [31:0] nextPC);

    always @(*) begin
        if (!rst) begin
            nextPC = pc + 4;
        end
        else begin
            case (pcSrc)
                `PC_NEXT: nextPC = pc + 4;
                `PC_REL_JUMP: nextPC = pc + 4 + (immediate
<< 2);
                `PC_ABS_JUMP: nextPC = jumpPC;
```



```
        `PC_HALT: nextPC = pc;
    default:
        nextPC = pc + 4;
    endcase
end
end
endmodule
```

h. 时钟分频模块

该模块对时钟信号进行16分频，输入为时钟信号 clk、重置信号 rst，输出分时后的时钟信号，用于数码管显示和按键防抖。

```
module Clock(
    input clk,
    input rst,
    output divClk);

    reg [16:0] cnt;
    always @(posedge clk or negedge rst) begin
        if (!rst) cnt <= 0;
        else cnt <= cnt + 1;
    end

    assign divClk = cnt[16];
endmodule
```

i. 数码管相关模块

根据数码管各位和对应管位置关系得到下述对应表：

显示数值	DigCodeSrc
0	1110_0000001
1	1110_1001111
2	1110_0010010
3	1110_0000110
4	1110_1001100
5	1110_0100100
6	1110_0100000
7	1110_0001111
8	1110_0000000
9	1110_0000100

A	1110_0001000
B	1110_1100000
C	1110_0110001
D	1110_1000010
E	1110_0110000
F	1110_0111000

根据上述表，实现显示模块。

```
module Display(  
    input clk,  
    input rst,  
    input [15:0] data,  
    output reg [3:0] digSrc,  
    output reg [6:0] digCodeSrc);  
  
    reg [1:0] digBit;  
    reg [3:0] dig;  
    wire [3:0] src;  
    assign src = 4'b1111;  
  
    always @(posedge clk or negedge rst) begin  
        if (!rst) begin  
            digBit <= 0;  
        end  
        else begin  
            digBit <= digBit + 1;  
        end  
    end  
  
    always @(*) begin  
        case (digBit)  
            0: dig = data[3:0];  
            1: dig = data[7:4];  
            2: dig = data[11:8];  
            3: dig = data[15:12];  
        endcase  
    end  
  
    always @(*) begin  
        case (dig)  
            4'h0: digCodeSrc = 7'b0000001; //0  
            4'h1: digCodeSrc = 7'b1001111; //1
```

```

4'h2: digCodeSrc = 7'b0010010; //2
4'h3: digCodeSrc = 7'b0000110; //3
4'h4: digCodeSrc = 7'b1001100; //4
4'h5: digCodeSrc = 7'b0100100; //5
4'h6: digCodeSrc = 7'b0100000; //6
4'h7: digCodeSrc = 7'b0001111; //7
4'h8: digCodeSrc = 7'b0000000; //8
4'h9: digCodeSrc = 7'b0000100; //9
4'hA: digCodeSrc = 7'b0001000; //A
4'hB: digCodeSrc = 7'b1100000; //B
4'hC: digCodeSrc = 7'b0110001; //C
4'hD: digCodeSrc = 7'b1000010; //D
4'hE: digCodeSrc = 7'b0110000; //E
4'hF: digCodeSrc = 7'b0111000; //F
    endcase
end

always @(*) begin
    digSrc = 4'b1111; // close all
    if (src[digBit]) digSrc[digBit] = 0; // display
end
endmodule

```

二、CPU 验证

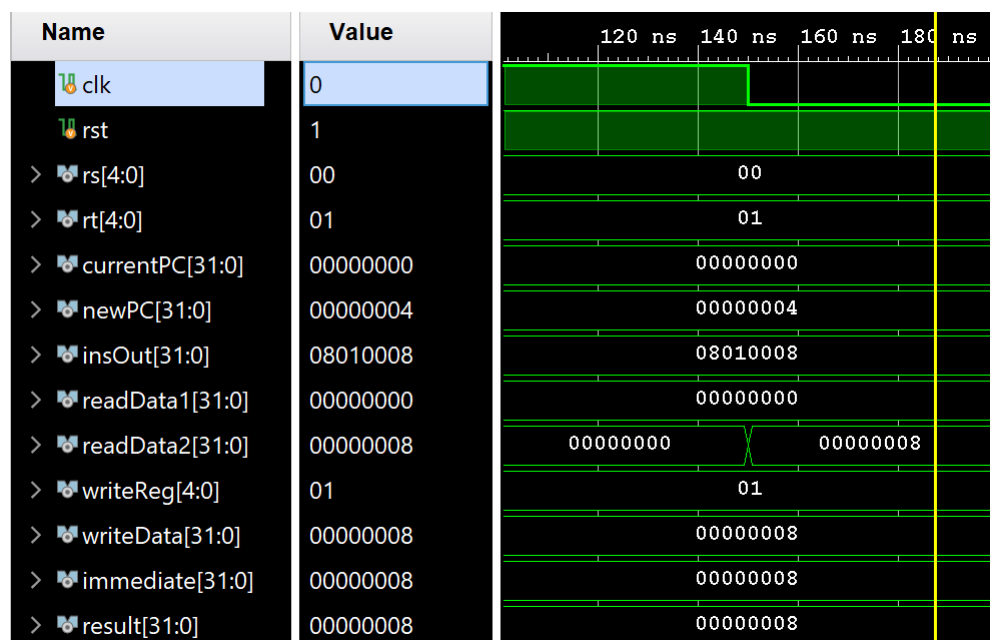
用于测试的指令如下：

PC	指令
0x00000000	addi \$1, \$0, 8
0x00000004	ori \$2, \$0, 2
0x00000008	add \$3, \$2, \$1
0x0000000C	sub \$5, \$3, \$2
0x00000010	and \$4, \$5, \$2
0x00000014	or \$8, \$4, \$2
0x00000018	sll \$8, \$8, 1
0x0000001C	bne \$8, \$1, -2 (≠, 转18)
0x00000020	slti \$6, \$2, 4
0x00000024	slti \$7, \$6, 0

0x00000028	addiu \$7, \$7, 8
0x0000002C	beq \$7, \$1, -2 (=, 转28)
0x00000030	sw \$2, 4(\$1)
0x00000034	lw \$9, 4(\$1)
0x00000038	addiu \$10, \$0, -2
0x0000003C	addiu \$10, \$10, 1
0x00000040	bltz \$10, -2 (<0, 转3C)
0x00000044	andi \$11, \$2, 2
0x00000048	j 0x00000050
0x0000004C	or \$8, \$4, \$2
0x00000050	halt

a. 仿真验证

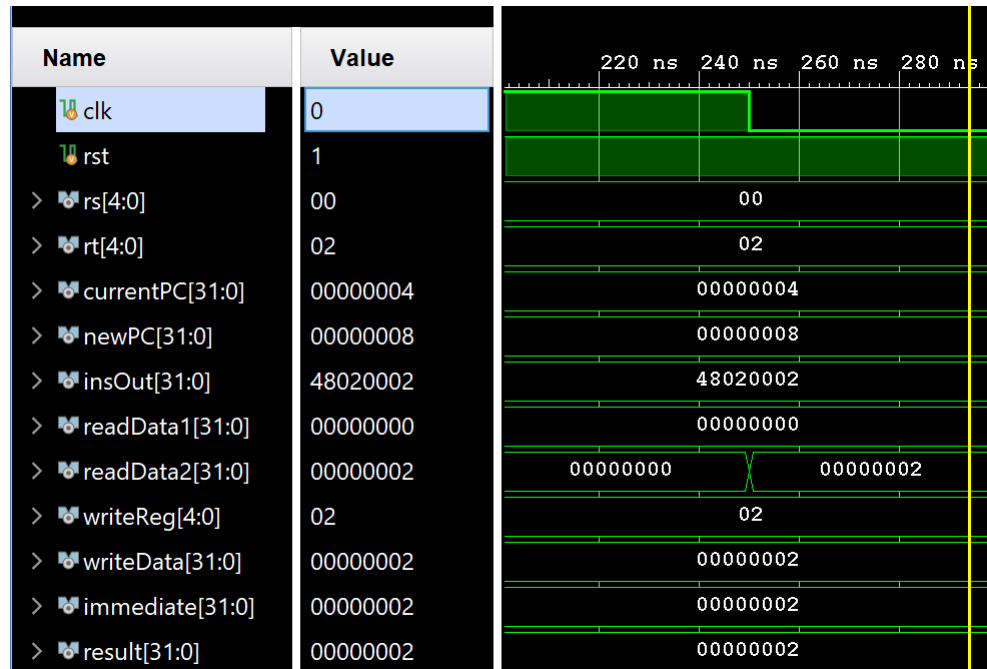
1. addiu \$1, \$0, 8



结果: $\$1 = 8$

解释: readData1为\$0寄存器的值0, readData2为\$1寄存器的值, 在时钟下降沿处从0变为8; immediate为立即数8; ALU运算结果result为8; writeReg表示写入到\$1寄存器, writeData表示写入值为8。结果正确。

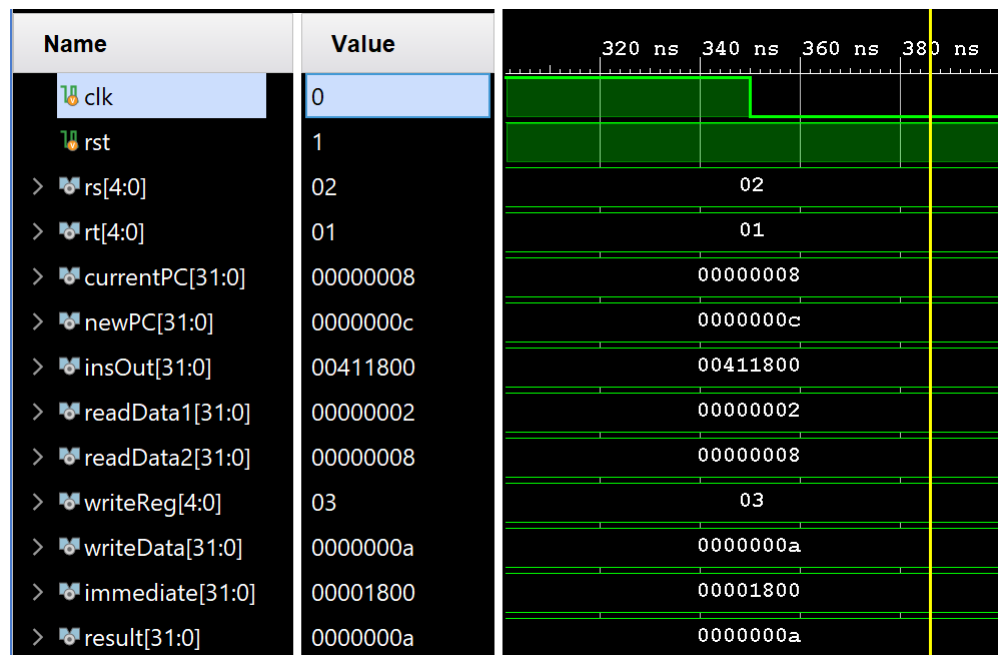
2. ori \$2, \$0, 2



结果: $\$2 = 0 + 2 = 2$

解释: readData1为\$0寄存器的值0, readData2为\$2寄存器的值, 在时钟下降沿处从0变为2; immediate为立即数2; ALU运算结果result为2; writeReg表示写入到\$2寄存器, writeData表示写入值为2。结果正确。

3. add \$3, \$2, \$1



结果: $\$3 = 2 + 8 = 10$

解释: readData1和readData2 分别为 rs (\$2)、rt (\$1) 的值2、8; ALU

sub \$5, \$3, \$2

Name	Value	
clk	0	
rst	1	
> rs[4:0]	03	
> rt[4:0]	02	
> currentPC[31:0]	0000000c	
> newPC[31:0]	00000010	
> insOut[31:0]	04622800	
> readData1[31:0]	0000000a	
> readData2[31:0]	00000002	
> writeReg[4:0]	05	
> writeData[31:0]	00000008	
> immediate[31:0]	00002800	
> result[31:0]	00000008	

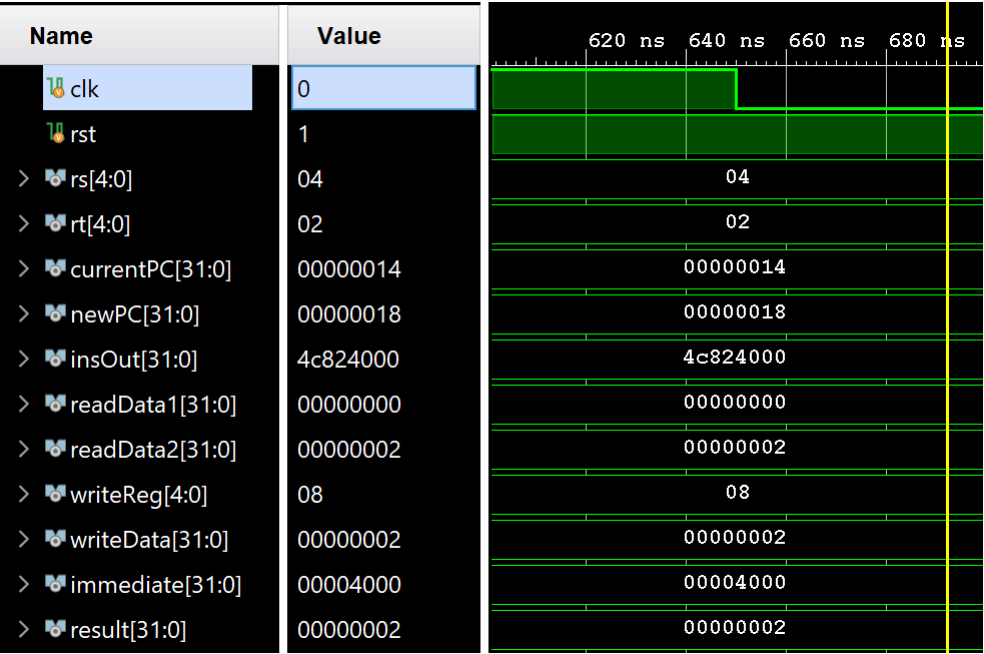
and \$4, \$5, \$2

Name	Value	520 ns	540 ns	560 ns	580 ns	ns
clk	0					
rst	1					
> rs[4:0]	05					
> rt[4:0]	02					
> currentPC[31:0]	00000010					
> newPC[31:0]	00000014					
> insOut[31:0]	44a22000					
> readData1[31:0]	00000008					
> readData2[31:0]	00000002					
> writeReg[4:0]	04					
> writeData[31:0]	00000000					
> immediate[31:0]	00002000					
> result[31:0]	00000000					

结果: $\$4 = 8 \& 2 = 0$

解释: readData1和readData2 分别为 rs (\$5)、rt (\$2) 的值8、2; ALU 运算结果result为0; writeReg表示写入到\$4寄存器, writeData表示写入值为0。结果正确。

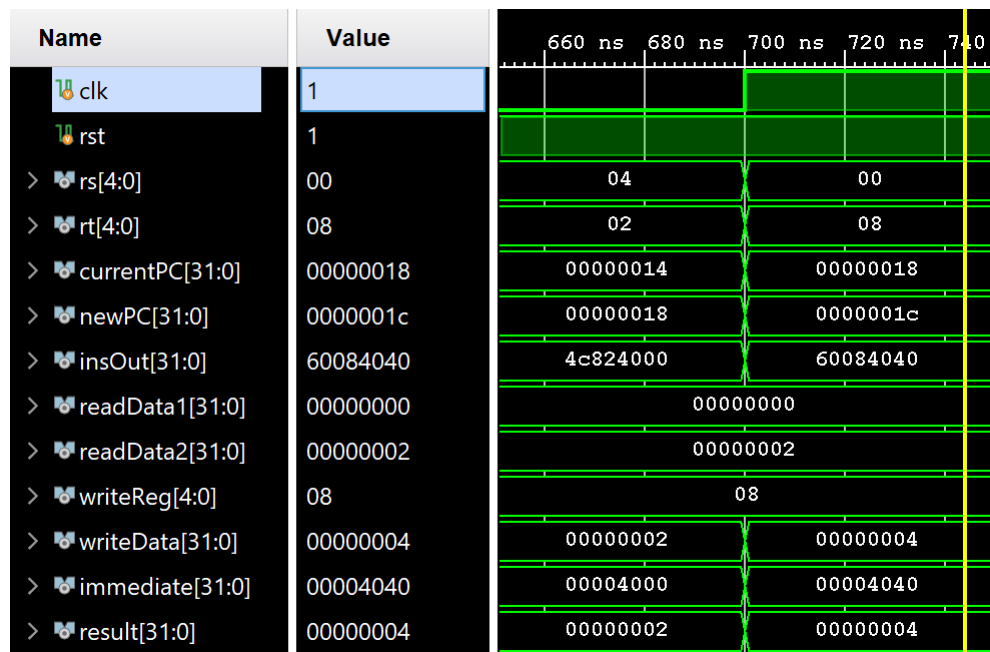
6. or \$8, \$4, \$2



结果: $\$8 = 0 \mid 2 = 2$

解释: readData1和readData2 分别为 rs (\$4)、rt (\$2) 的值0、2; ALU 运算结果result为2; writeReg表示写入到\$8寄存器, writeData表示写入值为2。结果正确。

7. sll \$8, \$8, 1



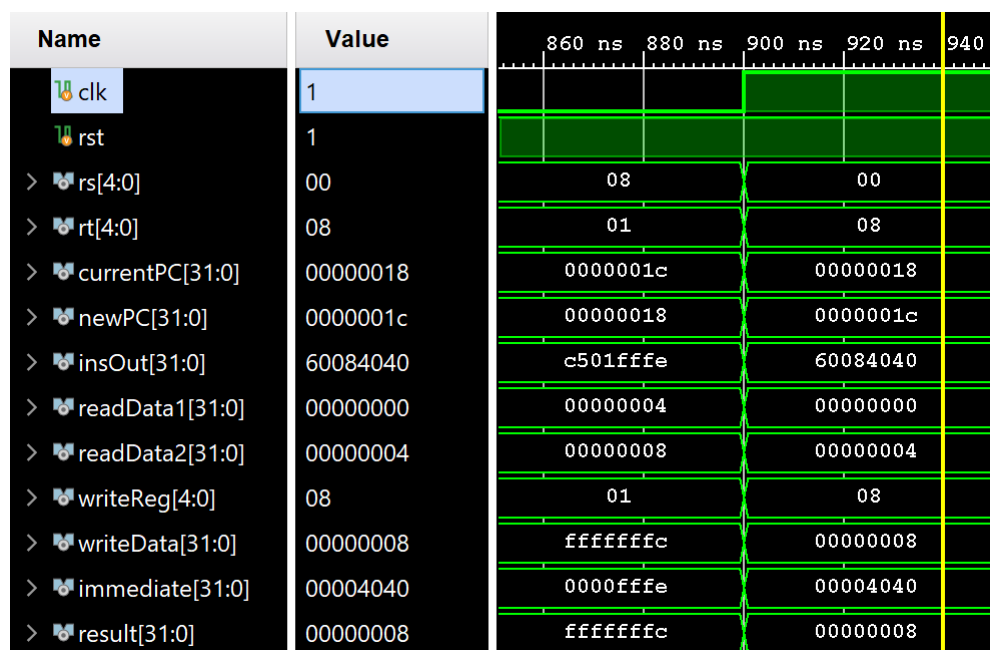
结果: $\$8 = 2 \ll 1 = 4$

解释: readData2 为 rt (\$8) 的值2, 然后在时钟下降沿处变为4; ALU运算

结果result为4; writeReg表示写入到\$8寄存器, writeData表示写入值为4。

结果正确。

8. bne \$8, \$1, -2



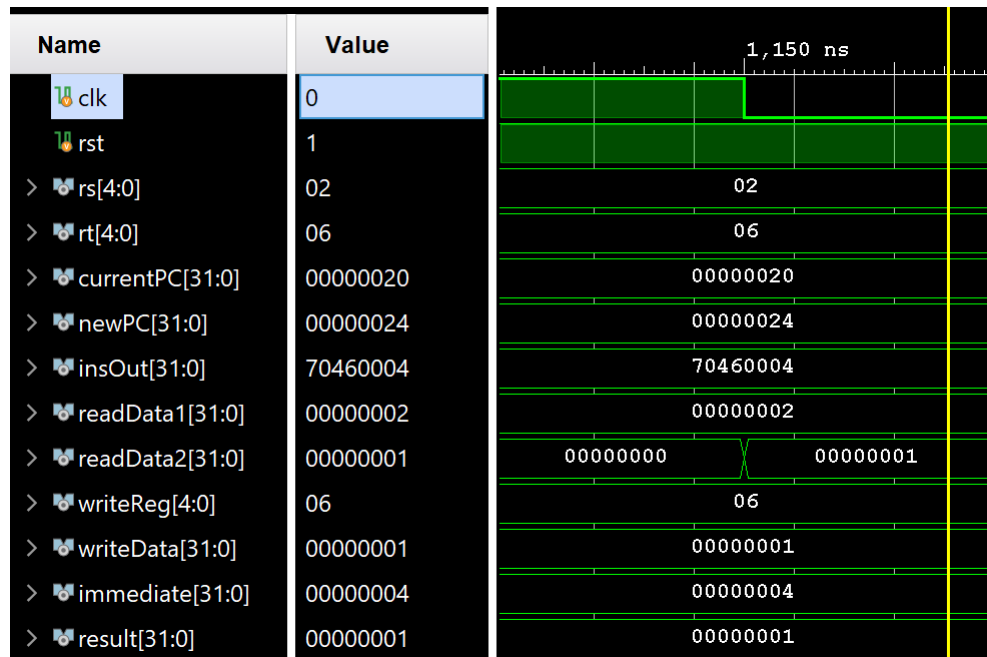
结果: $4 \neq 8 \rightarrow PC = PC - 2$

解释: readData1和readData2 分别为 rs (\$8) 、 rt (\$1) 的值4、8; 立即

数immediate为-2; ALU运算结果result为-4; 由于 $-4 \neq 0$ (即 $4 \neq 8$) , PC下

一个地址newPC为当前PC-2。结果正确。

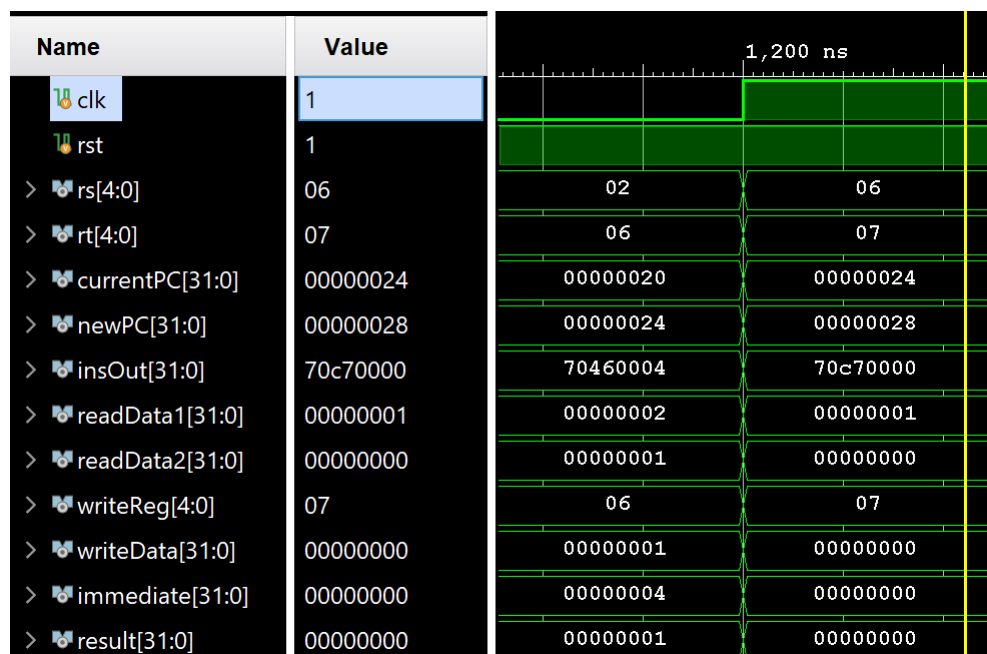
9. slti \$6, \$2, 4



结果: $\$2 < 4 \rightarrow \$6 = 1$

解释: readData1为 rs (\$2) 的值2; 立即数immediate为4; readData2为 rt (\$6) 的值, 由于 $2 < 4$ 因此在时钟下降沿处变为1; writeReg表示写入到\$6寄存器, writeData表示写入值为1。结果正确。

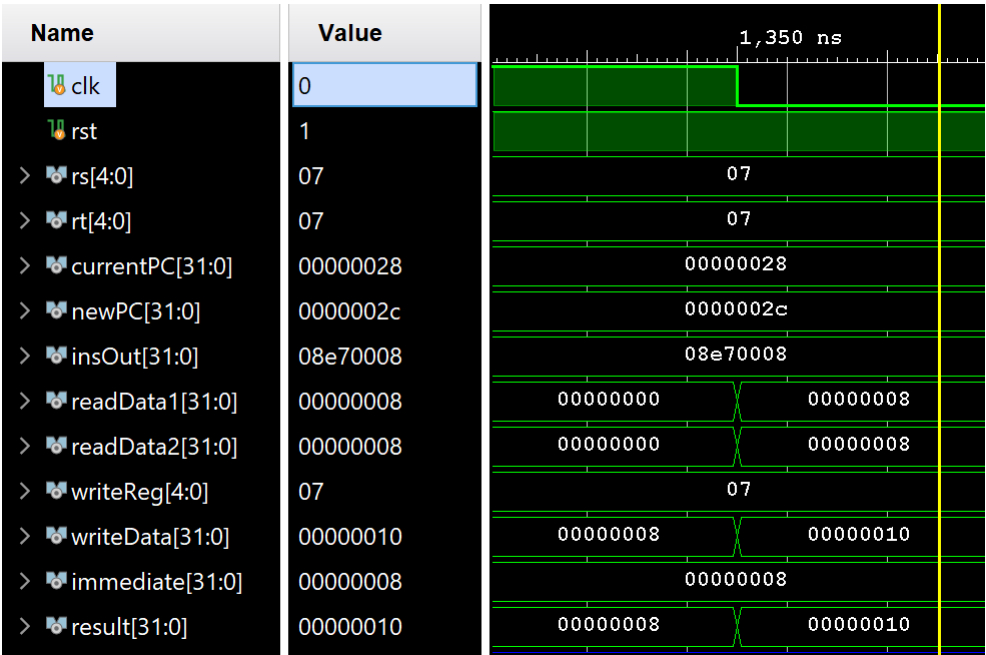
10. slti \$7, \$6, 0



结果: $\$6 > 0 \rightarrow \$7 = 0$

解释: readData1为 rs (\$6) 的值1; 立即数immediate为0; readData2为 rt (\$7) 的值, 由于1>0因此在时钟下降沿处变为0; writeReg表示写入到\$7寄存器, writeData表示写入值为0。结果正确。

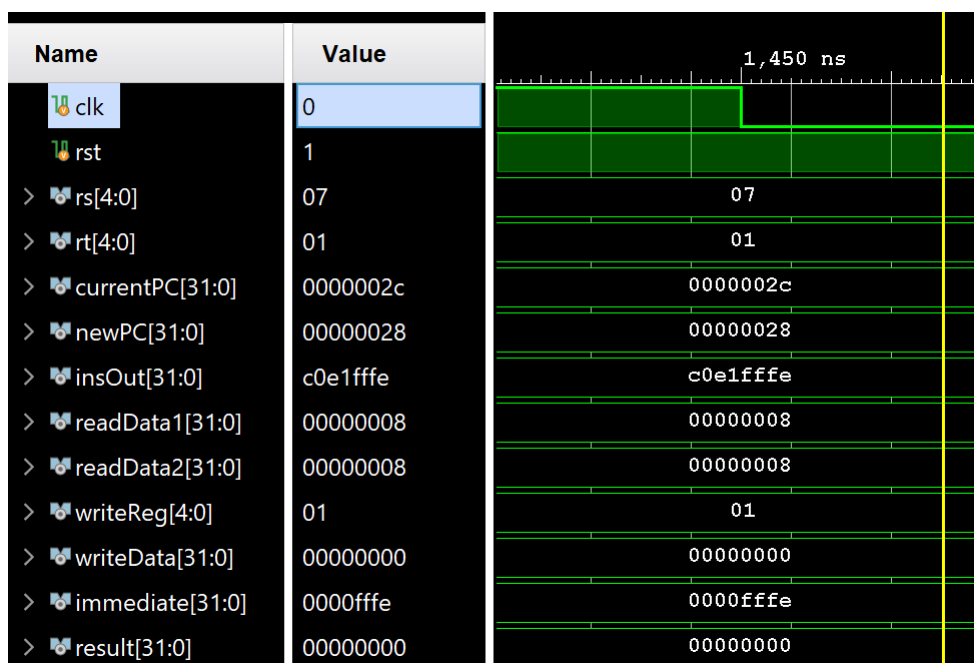
11.addiu \$7, \$7, 8



结果: $\$7 = 0 + 8 = 8$

解释: readData1为\$7寄存器的值0, readData2为\$7寄存器的值0, 在时钟下降沿处从0变为8; immediate为立即数8; ALU运算结果result为8; writeReg表示写入到\$7寄存器, writeData表示写入值为8。结果正确。

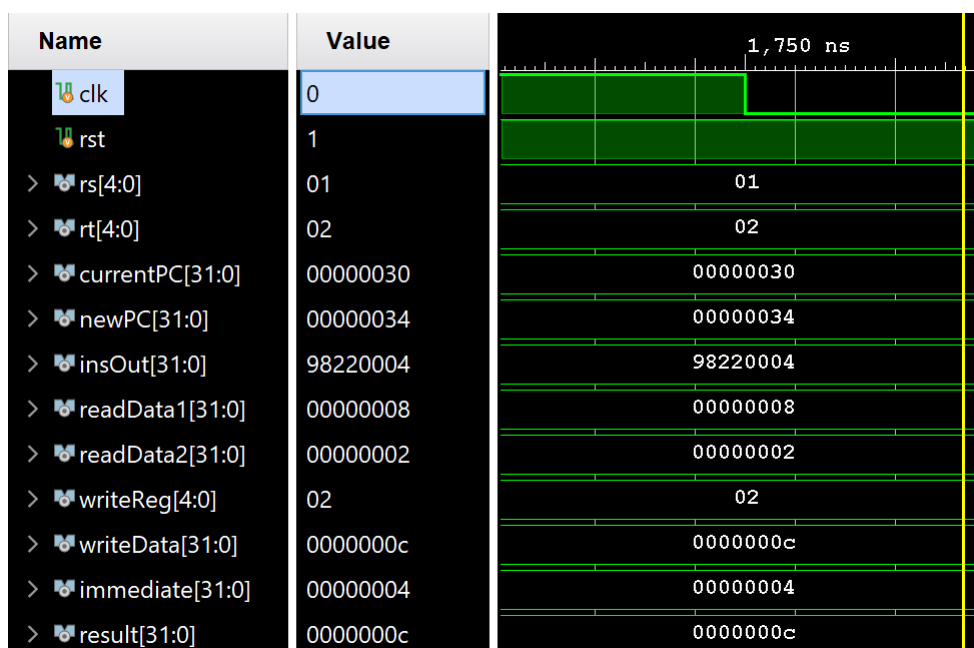
12.beq \$7, \$1, -2



结果: $8 = 8 \rightarrow PC = PC - 2$

解释: readData1、readData2分别为\$7、\$1寄存器的值8、8; immediate为立即数-2; ALU运算结果result为0; 由于 $8 = 8$, 下一个PC地址为当前PC-2结果正确。

13. sw \$2, 4(\$1)



Name	Value
memory[0:68][7]	XX,XX,X
> [12][7:0]	00
> [13][7:0]	00
> [14][7:0]	00
> [15][7:0]	02

结果: memory[12:15]=2

解释: readData1、readData2分别为\$1、\$2寄存器的值8、2; immediate为立即数4; ALU运算结果result为12; 由于采用8位一字节的大端方式存储数据, 因此会在12-15号内存写入数据2。结果正确。

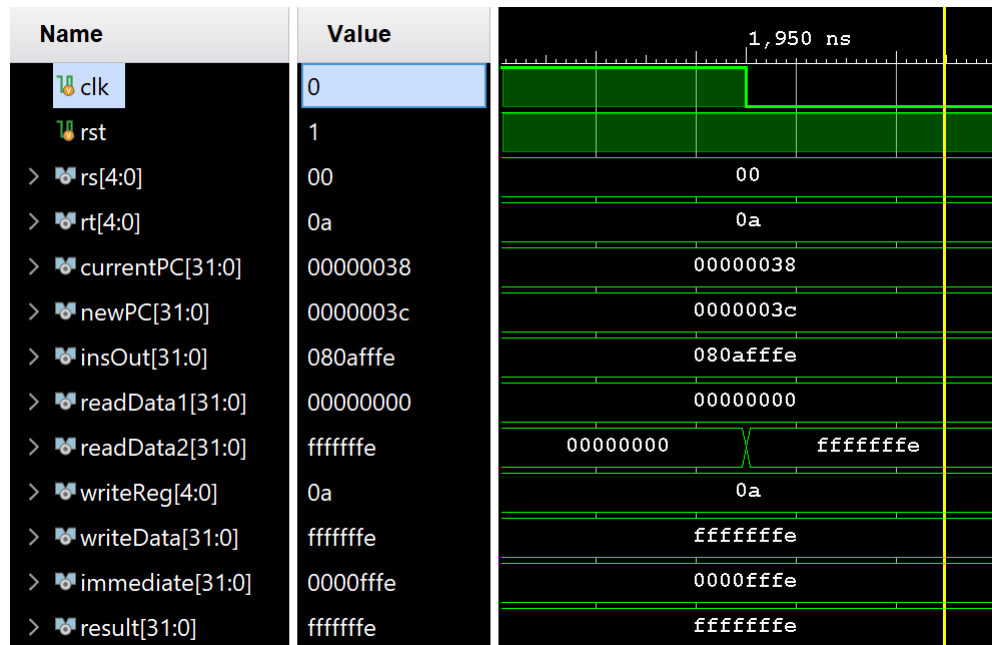
14.lw \$9, 4(\$1)

Name	Value	
clk	0	
rst	1	
> rs[4:0]	01	
> rt[4:0]	09	
> currentPC[31:0]	00000034	
> newPC[31:0]	00000038	
> insOut[31:0]	9c290004	
> readData1[31:0]	00000008	
> readData2[31:0]	00000000	
> writeReg[4:0]	09	
> writeData[31:0]	00000002	
> immediate[31:0]	00000004	
> result[31:0]	0000000c	

结果: \$9 = memory[12:15] = 2

解释: readData1为\$1寄存器的值8; immediate为立即数4; ALU运算结果result为12; 由于采用8位一字节的大端方式存储数据, 因此读入12-15号内存的数据2, 并在时钟下降沿写入\$9寄存器; writeReg表示写入到\$2寄存器; writeData表示写入的值为2。结果正确。

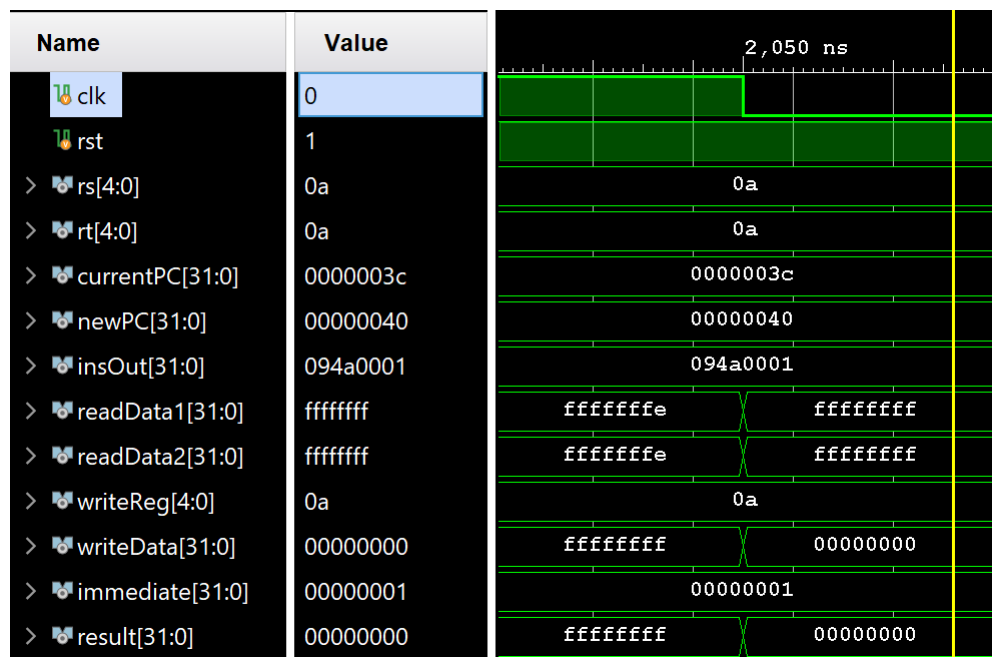
15.addiu \$10, \$0, -2



结果: $\$10 = 0 + (-2) = -2$

解释: readData1为\$0寄存器的值0, readData2为\$10寄存器的值, 在时钟下降沿处从0变为-2; immediate立即数经过符号扩展后为-2; ALU运算结果result为-2; writeReg表示写入到\$10寄存器, writeData表示写入值为-2。结果正确。

16.addiu \$10, \$10, 1

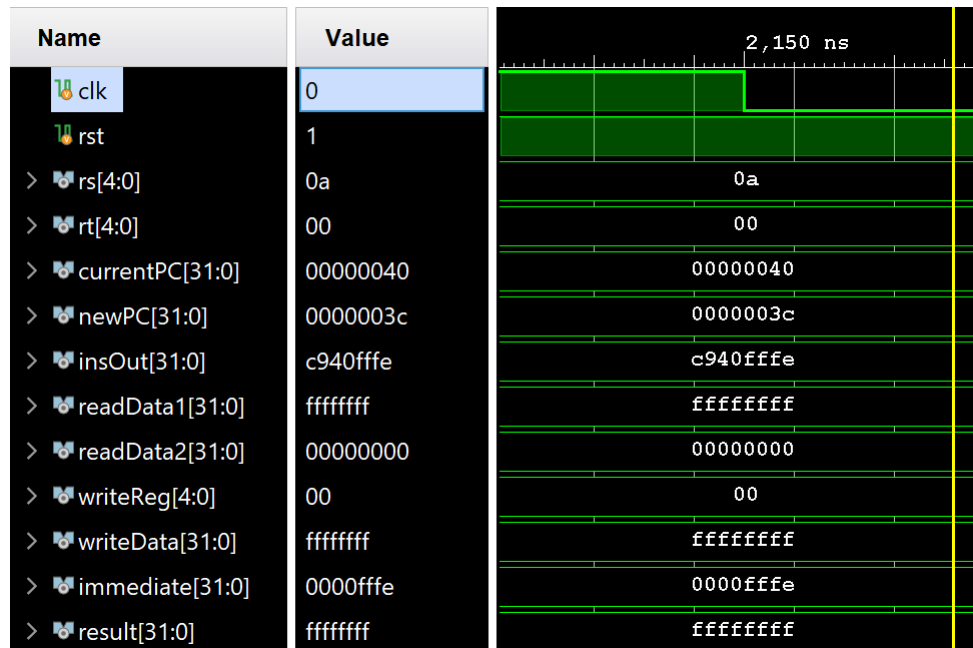


结果: $\$10 = -2 + 1 = -1$

解释: readData1为\$10寄存器的值-2, readData2为\$10寄存器的值-2, 在

时钟下降沿处从-2变为-1; immediate为立即数1; ALU运算结果result为-1;
writeReg表示写入到\$10寄存器, writeData表示写入值为-1。结果正确。

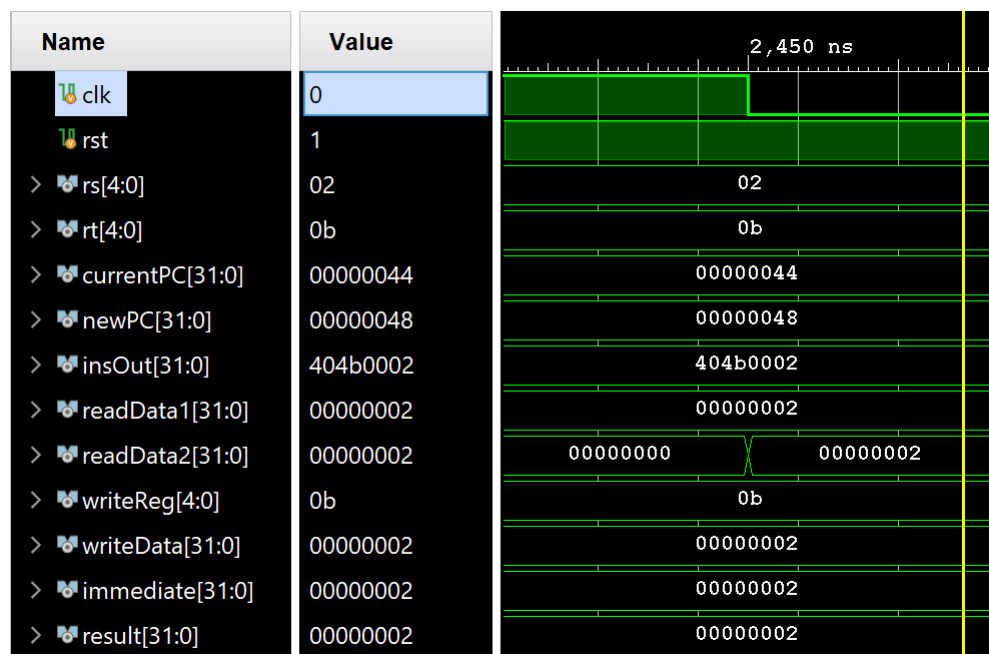
17. bltz \$10, -2



结果: $PC = PC - 2$

解释: readData1为\$10寄存器的值-1; immediate为立即数-2; ALU运算结果result为-1; 由于 $-1 < 0$, 因此下一个PC地址为当前PC-2。结果正确。

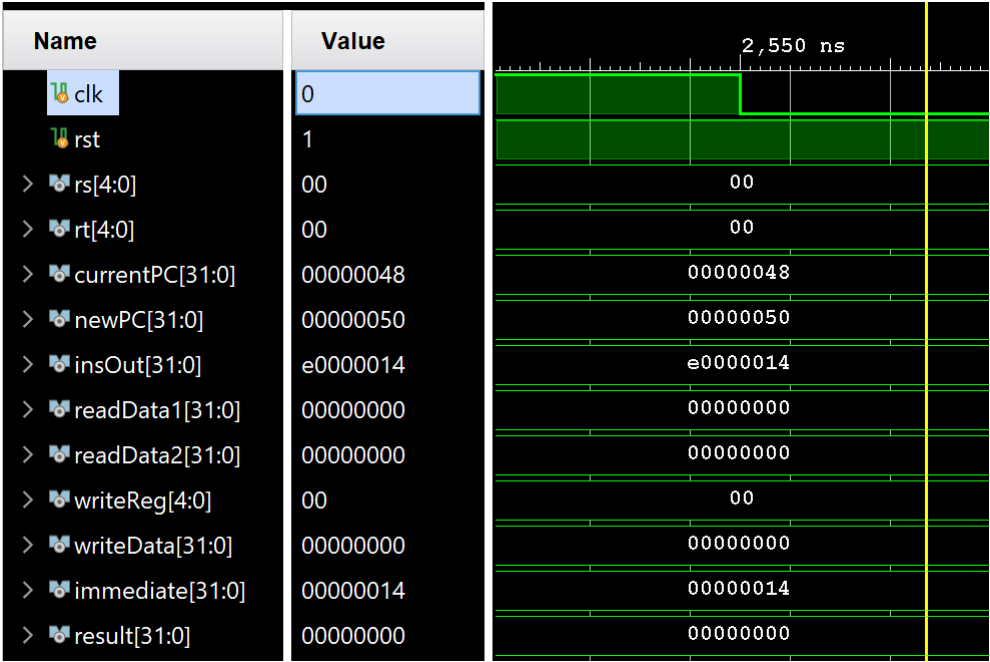
18. andi \$11, \$2, 2



结果: $\$11 = 2 \& 2 = 2$

解释: readData1为\$2寄存器的值2, readData2为\$11寄存器的值0, 在时钟下降沿处从0变为2; immediate为立即数2; ALU运算结果result为2; writeReg表示写入到\$11寄存器, writeData表示写入值为2。结果正确。

19.j 0x00000050



结果: PC = 50

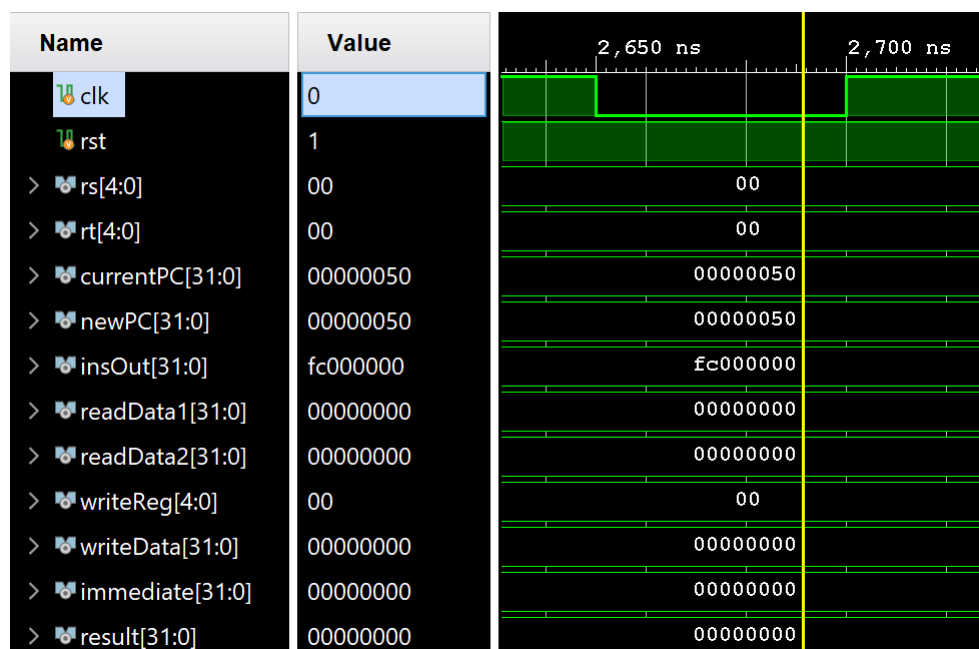
解释: 当前PC地址为48, 下一个PC地址为50。结果正确。

20.or \$8, \$4, \$2

结果: 无

解释: 由于上一步的 j 0x00000050, 该指令被跳过。

21.halt



结果：PC = 50

解释：停机指令，PC将一直停留在50，不再变化。

仿真验证结论：CPU实现正确。

b. 烧板验证

烧板验证，通过每按一下按键将信号值取反的方式作为CPU时钟，除此之外，还需要做按键防抖处理，可以在始终内对按键的信号进行采样，如果达到预设的周期则输出对应的信号，防止按一下按钮产生多次信号的问题，避免按键抖动。

另外，数码管需要合适的频率显示，如果频率不合适会导致数码管闪动过快而看不清，也可能会因为速度太慢导致逐个显示的情况。

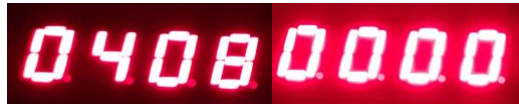
具体的时钟分频模块实现已在（一、模块设计）中给出。

以下图片中数位管两位为一组，依次为：当前PC、下一个PC、rs寄存器、rs寄存器数据、rt寄存器、rt寄存器数据、ALU运算结果、DB总线数据。

1. addiu \$1, \$0, 8



2. ori \$2, \$0, 2



04080000



02000200

3. add \$3, \$2, \$1

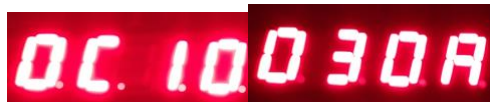


080C0202



01080A00

4. sub \$5, \$3, \$2



0C10030A



02020800

5. and \$4, \$5, \$2



10140508



02020000

6. or \$8, \$4, \$2



14180400



02020200

7. sll \$8, \$8, 1



181C0000



08020400

8. bne \$8, \$1, -2



1C 18 08 04
0 108 FC 00

9. slti \$6, \$2, 4



20 24 02 02
06 00 0 100

10. slti \$7, \$6, 0



24 28 06 0 1
0 700 0000

11. addiu \$7, \$7, 8



28 2C 0 700
0 700 0800

12. beq \$7, \$1, -2



2C 28 0 708
0 108 0000

13. sw \$2, 4(\$1)

3034 0108

0202 0C00

14. lw \$9, 4(\$1)

3438 0108

0900 0C02

15. addiu \$10, \$0, -2

383C 0000

0A00 FE00

16. addiu \$10, \$10, 1

3C40 0AFE

0AFE FF00

17. bltz \$10, -2

403C 0AFF



18. `andi $11, $2, 2`



19. `j 0x00000050`



20. `or $8, $4, $2`

由于上一条指令 `j 0x00000050`，该指令被跳过。

21. `halt`



烧板验证结论：CPU实现正确。

六. 实验心得

本次实验是本课程第一次比较复杂一点的实验设计，主要对理论课上的执行进行实践，加深对CPU方面的知识理解和运用。尽管拥有了第一次实验的实践基础，但在实验的过程中还是遇到了一系列问题，在解决问题的过程中也收获了许多。

1. 首先在使用Vivado软件的时候遇到了几个问题：

- (1) 在打开已有工程的时候遇到问题，本以为是在软件File选项卡会有打开选项，后来发现是在工程内部有一个用于打开整个工程的文件，和以前C，C++ 编程有些许不同之处。
- (2) 另一个就是在进行第一次仿真的时候，会报一个证书过期的错误，然后搜索了资料之后，发现要在Xilinx官网上下载需要的证书类型，才能使用相应的功能。
2. 其次就是在开始编写代码的时候，刚开始看着提供的几个参考文件根本无从下手。慢慢地在观察 CPU 的数据通路和控制线路图的过程中，发现了每个部件都有输入端口和输出端口，通过这些端口，各个部件相互联系，共同工作。抓住这点关系，就可以逐个击破，根据端口为每个部件创建变量，编写各自的功能代码。
3. 而且在控制单元的编写过程中，因为根据不同的指令的输入会形成一张真值表，如果直接使用二进制码进行真值选择，实在是太麻烦了，而且出错的概率也大大上升。所以为了解决这个问题，可以效仿 C 和 C++ 的写法，把复杂的内容通过宏定义将其简化，来避免复杂的代码编写。所以在工程中除了每个部件文件之外，还添加了一个宏定义文件 defines.v，需要使用到宏定义时就引用该文件。
4. 为了在烧板工程中达到按键防抖动的效果，不能直接采用按键直接取反。查找资料后发现，可以在始终上升沿时，对按键正信号或负信号进行取样，如果取样周期达到了预设周期，则输出正信号或者负信号，从而避免按键抖动，且输出稳定。

所以通过本次实验，我们对计算机的硬件结构更加了解，这种知识会对我们以后的编程有帮助：让我们更合理安排代码顺序，优化代码结构，提高代码的执行效率。同时单周期的CPU 实现也为我们接下来的多周期 CPU 实现打好基础，对我们以后的计算机原理实践有所帮助。