《计算机组成原理与接口技术实验》

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）：	16 软件工程四（7）班
学 生 姓 名 ：	徐伟元
学　　号 ：	16340261
时　　间 ：	2018　年　5　月　21　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；

(5) 掌握单周期 CPU 的实现方法。

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 （说明：以助记符表示，是汇编指令；以代码表示，是机器指令）**

000000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

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

（2）addi rt , rs ,**immediate**

000001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs + (sign-extend)**immediate；immediate** 符号扩展再参加“加”运算。

（3）sub rd , rs , rt

000010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs - rt

==> 逻辑运算指令

（4）ori rt , rs ,**immediate**

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

功能：rt←rs | (zero-extend)**immediate；immediate** 做“0”扩展再参加“或”运算。

（5）and rd , rs , rt

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

功能：rd←rs & rt；逻辑与运算。

（6）or rd , rs , rt

010010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

==>移位指令

（7）sll rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved

功能：rd<－rt<<(zero-extend)sa，左移 sa 位 ，(zero-extend)sa

==>比较指令

（8）slti　rt, rs,immediate　　带符号

011011	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

（9）sw　rt ,**immediate**(rs) 写存储器

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

功能：memory[rs+ (sign-extend)**immediate**]←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。

（10) lw　rt , **immediate**(rs) 读存储器

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

功能：rt ← memory[rs + (sign-extend)**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

（11）beq　rs,rt,**immediate**

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

功能：if(rs=rt) pc←pc + 4 + (sign-extend)**immediate** <<2　else pc ←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”。

（12）bne　rs,rt,**immediate**

110001	rs(5 位)	rt(5 位)	**immediate**

功能：if(rs!=rt) pc←pc + 4 + (sign-extend)immediate <<2　else pc ←pc + 4

特别说明：与 beq 不同点是，不等时转移，相等时顺序执行。

==>跳转指令

（13）j　addr

111000	addr[27..2]

功能：pc <－{(pc+4)[31..28],addr[27..2],2{0}}，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 位地址了，剩下最高 4 位由 pc+4 最高 4 位拼接上。

==> 停机指令

（14）halt

111111	00000000000000000000000000(26 位)

功能：停机；不改变 PC 的值，PC 保持不变。

补充:

1、PC 和寄存器组写状态使用时钟触发。

2、指令存储器和数据存储器存储单元宽度一律使用 8 位，即一个字节的存储单位。不能使用 32 位作为存储器存储单元宽度。

3、控制器部分要学会用控制信号真值表方法分析问题并写出逻辑表达式；或者用 case 语句方法逐个产生各指令控制信号。注意：控制信号的产生不能使用时钟触发!

4、必须写一段测试用的汇编程序，而且必须包含所要求的所有指令，slti 指令必须检查两种情况："小于"和"大于等于"；beq、bne："不等"和"等"。

5、Always@（…）的敏感信号表中，时序触发和电平触发不能同时出现，即不能混用。

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。

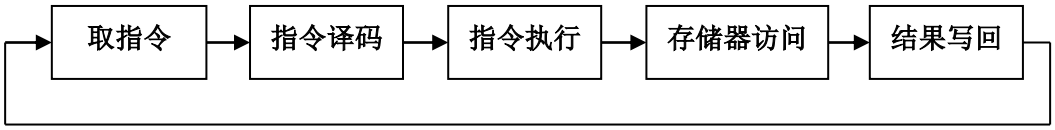
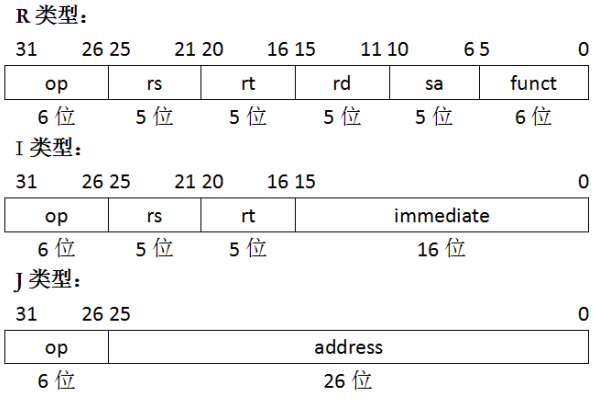


图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式:



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。

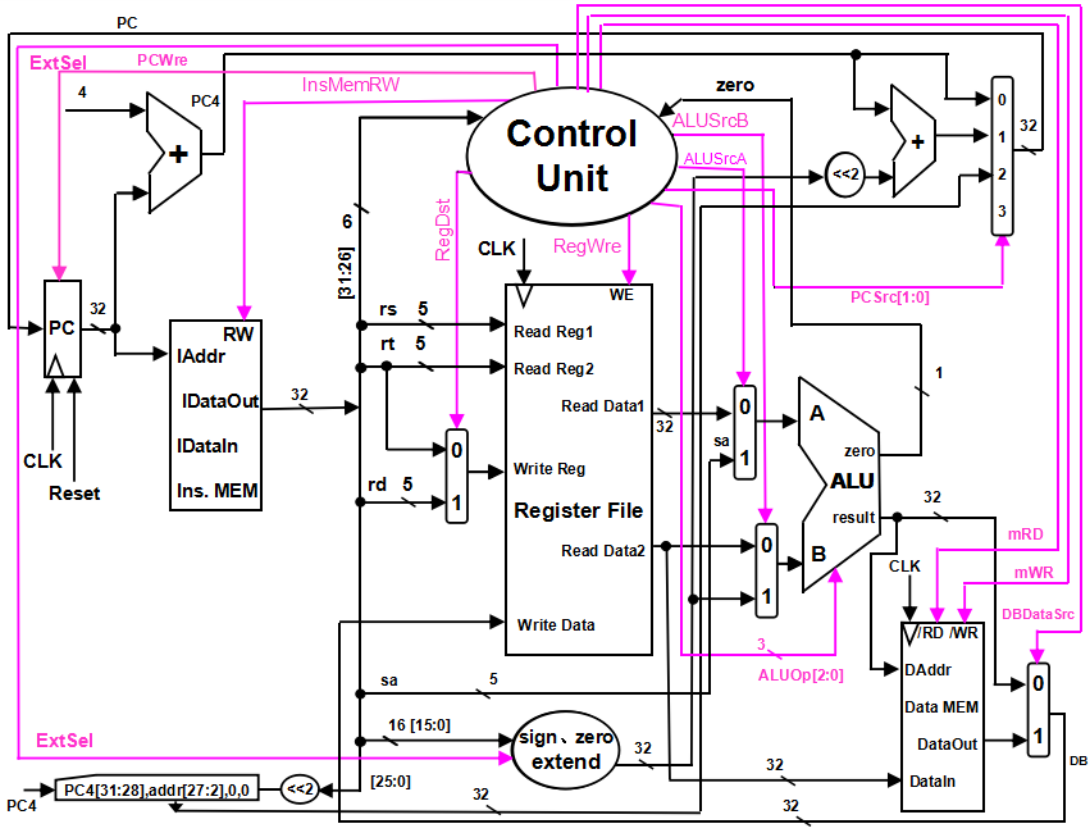


图 2　单周期 CPU 数据通路和控制线路图

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，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

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、addi、or、and、ori、beq、bne、slti、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、sll、beq、bne	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、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、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、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：addi、ori、lw、slti	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：ori	(sign-extend)**immediate（符号扩展）**，相关指令：addi、slti、sw、lw、beq、bne
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)； 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bne(zero=0)； 10：pc<－{(pc+4)[31:28],addr[27:2],2{0}}，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	

相关部件及引脚说明：

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

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(当然，以上指令没有完全用到提供的 ALU 所有功能，但至少必须能实现以上指令功能操作)。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、层次化的思想方法设计，关于如何划分模块、如何整合成一个系统等等，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 设计时，主要参考的流程图是数据通路和控制线路图。从图左边的 PC 模块开始，从左向右，从上到下进行各个模块的设计。

A. 模块设计：

1、 PC(程序计数器)

PC 输出下一指令地址；在时钟上升沿到来时，给出下一条指令的地址，或者在重置信号下降沿时，将下一条指令地址置零。需要注意的一个地方是 halt 指令需将 PC 值维持不变。

I. Module PC：

输入：CLK，Reset，PCWre，NextPC

输出：IAddress

PC 真值表			
输入			输出
Reset	PCWre	PCSrc	NextPC
0	X	x	0
1	0	x	不变
1	1	00	PC+4
1	1	01	PC+4+(extend)immediate
1	1	10	{(PC+4)[31:28],address[27:2],0,0}
1	1	11	不变

```
module PC(
    input CLK,                  // 时钟输入
    input Reset,                // 重置信号
    input PCWre,                // PC 更改信号，为 0 时不更改
    input [31:0] NextPC,        // 新指令
    output reg [31:0] IAddress      // 输出指令地址
    );
    // 初始化
    initial begin
        IAddress = 0;
    end
    // 时钟上升沿或重置下降沿触发
    always @(posedge CLK or negedge Reset) begin
        if (!Reset) begin
            IAddress <= 32'hFFFFFFFC;   // 重置为-4
        end
        else if (PCWre || !NextPC) begin
            IAddress <= NextPC;
        end
    end
endmodule   // PC
```

PC 模块中的下一条指令地址，可能来自正常加四指令地址，分支跳转指令地址或 j 指令跳转地址，所以我们设计一个 NextPC 模块(可视为选择器)，来辅助选择下一条指令地址。

II. Module NextPC

输入：Reset，PCSrc，PC，Immediate，JPC

输出：NextPC

NextPC 真值表	
输入	输出
PCSrc	NextPC
00	PC + 4
01	PC + 4 + (Immediate << 2)
10	JPC
其他	PC + 4

```
module NextPC(
    input Reset,                // 重置信号
    input [1:0] PCSrc,          // 选择信号
    input [31:0] PC,            // PC 地址
    input [31:0] Immediate,     // 立即数
    input [31:0] JPC,           // 跳转 PC
    output reg [31:0] NextPC        // 下一条 PC
    );

    always @(Reset or PCSrc or PC or Immediate or JPC) begin
        if (!Reset) begin
            NextPC = PC + 4;
        end
        else begin
            case (PCSrc)
              2'b00: NextPC = PC + 4;
              2'b01: NextPC = PC + 4 + (Immediate << 2);
              2'b10: NextPC = JPC;
              default: NextPC = PC + 4;
            endcase
        end
    end
endmodule   // Next PC
```

接下来，设计有关 j 指令的指令跳转地址计算。其中，计算公式如下：

PC <— {(PC+4)[31：28], addr[27：2], 2{0}}

III. Module JPC

输入：PC，IAddress

输出：JPC

```
module JPC(
    input [31:0] PC,            // PC 地址
    input [25:0] IAddress,      // 跳转地址
    output reg [31:0] JPC       // 跳转 PC
    );

    wire [27:0] temp;
    assign temp = IAddress << 2;
    always @(PC or IAddress) begin
        JPC[31:28] = PC[31:28];
        JPC[27:2] = temp[27:2];
        JPC[1:0] = 0;
    end
endmodule   // Jump PC
```

2、 InstructionMemory(指令存储器)

指令存储器存储着我们需要执行的指令及其对应地址。所以，需要存储指令的存储单元，并在指令读写选择信号的控制下进行指令(IDataIn)的写入或指令的读取。本实验只有读取指令操作。

输入：InsMemRW，IAddress，IDataIn

输出：IDataOut

```
module InstructionMemory(
    input InsMemRW,                 // 指令读写选择信号，1 为读，0 为写
    input [31:0] IAddress,          // 指令地址输入
    input [31:0] IDataIn,           // 指令寄存器输入数据
    output reg [31:0] IDataOut      // 指令存储器输出数据
    );
    // 8 位长的指令存储单元，共 128 个
    reg [7:0] Memory[0:127];
    initial begin
        $readmemb("Path/instruction.txt", Memory);
    end
    // 从地址取值后输出指令
    always @(IAddress or InsMemRW) begin
        if (InsMemRW) begin
            IDataOut[31:24] = Memory[IAddress];
            IDataOut[23:16] = Memory[IAddress + 1];
            IDataOut[15:8] = Memory[IAddress + 2];
            IDataOut[7:0] = Memory[IAddress + 3];
        end
    end
endmodule
```

3、 Control Unit(控制单元)

控制单元，根据指令发出针对其余模块的控制信号，以使得其余模块按照指令正常工作。控制信号与指令的真值表如下：

OpCode	指令	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	InsMemRW
000000	add	1	0	0	0	1	1
000001	addi	1	0	1	0	1	1
000010	sub	1	0	0	0	1	1
010000	ori	1	0	1	0	1	1
010001	and	1	0	0	0	1	1
010010	or	1	0	0	0	1	1
011000	sll	1	1	0	0	1	1
011011	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
111000	j	1	x	x	x	0	1
111111	halt	0	x	x	x	0	1

OpCode	指令	mRD	mWR	RegDst	ExtSel	PCSrc	ALUOp
000000	add	0	0	1	x	00	000
000001	addi	0	0	0	1	00	000
000010	sub	0	0	1	x	00	001
010000	ori	0	0	0	0	00	011
010001	and	0	0	1	x	00	100
010010	or	0	0	1	x	00	011
011000	sll	0	0	1	x	00	010
011011	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	00(zero=1); 01(zero=0)	001
111000	j	0	0	x	x	10	010
111111	halt	0	0	x	x	xx	xxx

如果每次都输入指令的二进制码，太过繁琐，所以构造了一个 head.v 头文件，将所有二进制码定义为了其对应指令。(具体见代码文件下的 head.v 文件)

输入：OpCode，Zero，Sign

输出：PCWre，ALUSrcA，ALUSrcB，DBDataSrc，RegWre，InsMemRW，mRD，mWR，RegDst，ExtSel，PCSrc，ALUOp

```
module ControlUnit(
    input [5:0] OpCode,     // 操作符
    input Zero,            // ALU 的 Zero 输出
    input Sign,            // ALU 的 Sign 输出
    output PCWre,          // PC 更改信号
    output ALUSrcA,        // ALU 左操作数选择信号
    output ALUSrcB,        // ALU 右操作数选择信号
    output DBDataSrc,      // 写入寄存器数据选择信号
    output RegWre,         // 寄存器组写使能信号
    output InsMemRW,       // 指令存储器读写信号
    output mRD,            // 内存读信号
    output mWR,            // 内存写信号
    output RegDst,         // 写寄存器组地址
    output ExtSel,         // 拓展方式选择信号
    output [1:0] PCSrc,     // 指令分支选择信号
    output [2:0] ALUOp      // ALU 功能选择信号
    );

    assign PCWre = (OpCode == `opHalt) ? 0 : 1;
    assign ALUSrcA = (OpCode == `opSll) ? 1 : 0;
```

```
    assign ALUSrcB = (OpCode == `opAddi || OpCode == `opOri || OpCode == `opSlti
|| OpCode == `opSw || OpCode == `opLw) ? 1 : 0;
    assign DBDataSrc = (OpCode == `opLw) ? 1 : 0;
    assign RegWre = (OpCode == `opSw || OpCode == `opBeq || OpCode == `opBne
|| OpCode == `opJ || OpCode == `opHalt) ? 0 : 1;
    assign InsMemRW = 1;
    assign mRD = (OpCode == `opLw) ? 1 : 0;
    assign mWR = (OpCode == `opSw) ? 1 : 0;
    assign RegDst = (OpCode == `opAdd || OpCode == `opSub || OpCode == `opAnd
|| OpCode == `opOr || OpCode == `opSll) ? 1 : 0;
    assign ExtSel = (OpCode == `opOri) ? 0 : 1;
    assign PCSrc[1] = (OpCode == `opJ || OpCode == `opHalt) ? 1 : 0;
    assign PCSrc[0] = ((OpCode == `opBeq && Zero) || (OpCode == `opBne && !Zero)
|| OpCode == `opHalt) ? 1 : 0;
    assign ALUOp[2] = (OpCode == `opAnd || OpCode == `opSlti) ? 1 : 0;
    assign ALUOp[1] = (OpCode == `opOri || OpCode == `opOr || OpCode == `opSll
|| OpCode == `opSlti || OpCode == `opJ) ? 1 : 0;
    assign ALUOp[0] = (OpCode == `opSub || OpCode == `opOr || OpCode == `opOri
|| OpCode == `opBeq || OpCode == `opBne) ? 1 : 0;
endmodule
```

4、 RegisterFile(寄存器组)

模拟 MIPS 中的32个寄存器，并注意在写入数据时保护 0 号寄存器。读取寄存器数据时，无需时钟信号；当写使能信号为 1 且时钟到达下降沿时，将数据写入寄存器。当重置信号为 0 时，重置所有寄存器数据。

输入：CLK，WE，Reset，ReadReg1，ReadReg2，WriteReg，WriteData

输出：ReadData1，ReadData2

```
module RegisterFile(
    input CLK,                  // 时钟输入
    input WE,                   // 写使能信号
    input Reset,                // 重置信号
    input [4:0] ReadReg1,       // 读寄存器 1 地址
    input [4:0] ReadReg2,       // 读寄存器 2 地址
    input [4:0] WriteReg,       // 写寄存器地址
    input [31:0] WriteData,     // 写数据
    output [31:0] ReadData1,    // 读数据 1
    output [31:0] ReadData2     // 读数据 2
    );


    // 0 号寄存器值恒为 0
    // 初始化寄存器的值
    integer i;
    reg [31:0] Register[0:31];
```

```
    initial begin
        for (i = 0; i < 32; i = i + 1) begin
            Register[i] = 0;
        end
    end

    // 读寄存器
    assign ReadData1 = Register[ReadReg1];
    assign ReadData2 = Register[ReadReg2];

    // 写寄存器
    // 注意保护 0 号寄存器
    always @(negedge CLK) begin
        if (!Reset) 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
```

5、 SignZeroExtend(符号位或零拓展)

根据拓展选择信号，选择拓展方式：符号位拓展或零拓展，并对 16 位立即数进行拓展，输出 32 位数。其中，拓展选择信号为 0 时，进行零拓展，为 1 时，进行符号位拓展。

```
module SignZeroExtend(
    input ExtSel,              // 拓展选择信号，为 0 则全补 0，否则进行符号位拓展
    input Sign,                // 符号位
    input [15:0] Immediate,    // 16 位立即数
    output [31:0] Extend       // 拓展输出
    );

    // 拓展立即数
    assign Extend[15:0] = Immediate[15:0];
    assign Extend[31:16] = (ExtSel && Immediate[15]) ? 16'hFFFF : 16'h0000;
endmodule
```

6、 ALU(算逻运算单元)

算术逻辑运算单元，根据控制信号的不同，选择对输入操作数进行不同的算术或逻辑运算，得到结果。算术功能包括加，减和移位，输出包含结果与符号位；逻辑功能包括或，与，比较和异或，输出包含结果与标志位。具体功能与控制信号对应如下表：

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

输入：ALUop，A，B

输出：Zero，Sign，Y

```
module ALU(
    input [31:0] A,     // 左操作数
    input [31:0] B,     // 右操作数
    input [2:0] ALUop,  // ALU 控制选择
    output Zero,        // 运算结果标志，结果为 0 输出 1，否则为 0
    output Sign,        // 符号位
    output reg [31:0] Y     // 计算结果
    );

    // 根据运算控制信号选择对应功能
    always @(ALUop or A or B) begin
        case (ALUop)
            3'b000 : Y = (A + B);
            3'b001 : Y = (A - B);
            3'b010 : Y = (B << A);
            3'b011 : Y = (A | B);
            3'b100 : Y = (A & B);
            3'b101 : Y = (A < B) ? 1 : 0;
            3'b110 : Y = (((A < B) && (A[31] == B[31])) || ((A[31] && !B[31]))) ?
1 : 0;
            3'b111 : Y = (A ^ B);
            default : Y = 0;
        endcase
    end

    assign Zero = (Y == 0) ? 1 : 0;
    assign Sign = Y[31];
endmodule
```

7、 DataMemory(数据存储器)

数据存储器，采用 8 位一字节，大端模式模拟内存。当读使能信号为 1 时，读取数据；当写使能信号为 1 且时钟到达下降沿时，写入数据。

输入：CLK，mRD，mWR，DAddr，DataIn

输出：DataOut

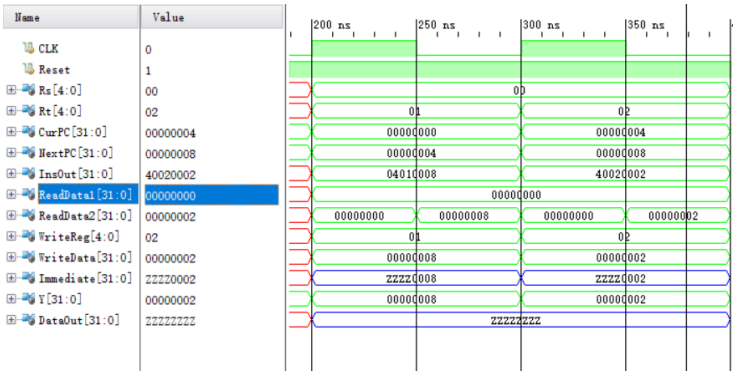
```
module DataMemory(
    input CLK,              // 时钟输入
    input mRD,              // 读数据使能输入
    input mWR,              // 写数据使能输入
    input [31:0] DAddr,     // 数据内存地址
    input [31:0] DataIn,    // 输入数据
    output [31:0] DataOut          // 输出数据
    );

    // 8 位一字节，模拟内存
    // 采用大端模式存储数据
    reg [7:0] memory[0:127];

    // 初始化内存
    integer i;
    initial begin
        for (i = 0; i < 128; i = i + 1) begin
            memory[i] = 0;
        end
    end

    // 读数据
    assign DataOut[7:0] = (mRD) ? memory[DAddr + 3] : 8'bz;
    assign DataOut[15:8] = (mRD) ? memory[DAddr + 2] : 8'bz;
    assign DataOut[23:16] = (mRD) ? memory[DAddr + 1] : 8'bz;
    assign DataOut[31:24] = (mRD) ? memory[DAddr] : 8'bz;

    // 写数据
    always @(negedge CLK) begin
        if (mWR) 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
```

B. 仿真验证：

1、 addi $1,$0,8

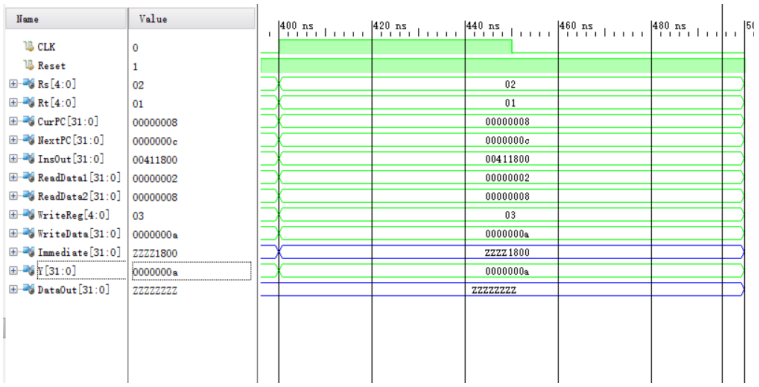
当前指令地址 CurPC 为 0，下一条 NextPC 为 4。立即数 Immediate 为 8。Rs 为 0 号寄存器，对应值 ReadData1 保持为 0；Rt 为 1 号寄存器，对应值 ReadData2 在时钟下降沿时由 0 变为 8。ALU 运算结果 Y 为 8。写入寄存器 WriteReg 为 1 号寄存器，写入数据 WriteData 为 8。指令执行正确($1 = $0 + 8 = 0 + 8 = 8)。

2、 ori $2,$0,2



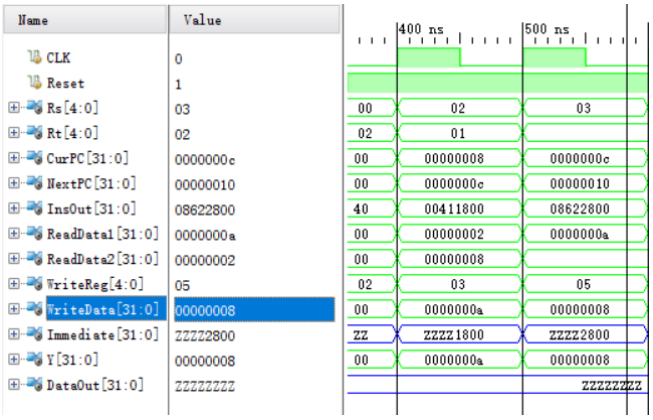
Rs 为 0 号寄存器，对应值 ReadData1 为 0；Rt 为 2 号寄存器，对应值 ReadData2 在时钟下降沿时由 0 变为 2。ALU 运算结果 Y 为 2。写入寄存器 WriteReg 为 2 号寄存器，写入数据 WriteData 为 2。指令执行正确($2 = $0 | 2 = 0 | 2 = 2)。

3、 add $3,$2,$1



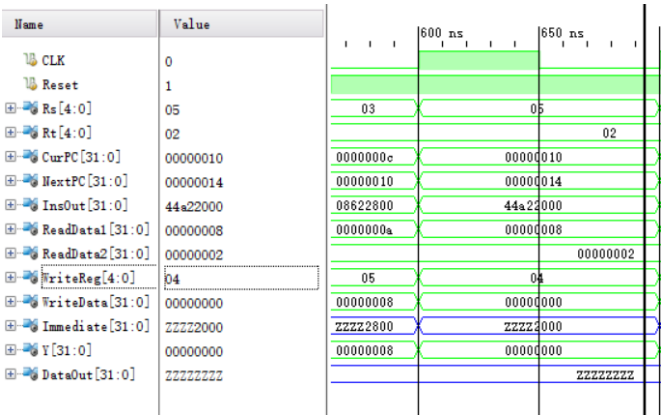
Rs 为 2 号寄存器，对应值 ReadData1 为 2；Rt 为 1 号寄存器，对应值 ReadData2 为 8。ALU 运算结果 Y 为 10。写入寄存器 WriteReg 为 3 号寄存器，写入数据 WriteData 为 10。指令执行正确($3 = $2 + $1 = 2 + 8 = 10)。

4、 sub $5,$3,$2



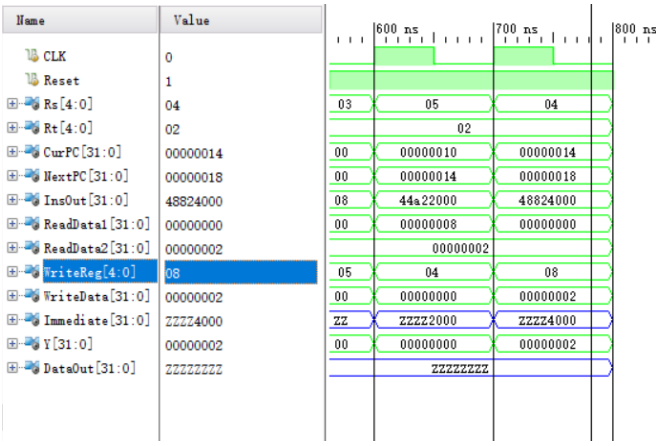
Rs 为 3 号寄存器，对应值 ReadData1 为 10；Rt 为 2 号寄存器，对应值 ReadData2 为 2。ALU 运算结果 Y 为 8。写入寄存器 WriteReg 为 5 号寄存器，写入数据 WriteData 为 8。指令执行正确($5 = $3 - $2 = 10 - 2 = 8)。

5、 and $4,$5,$2



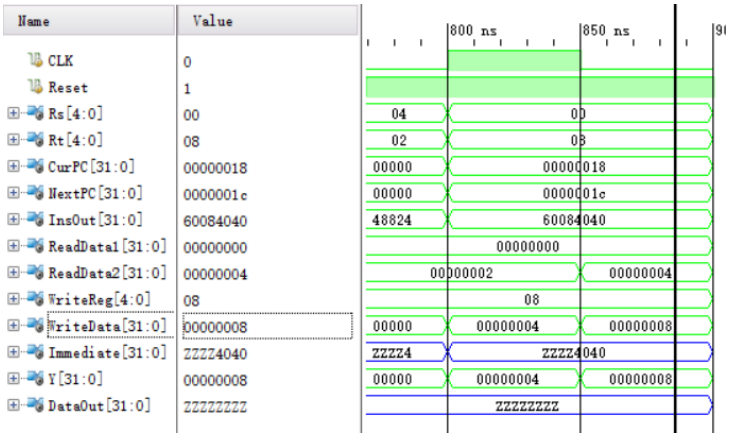
Rs 为 5 号寄存器，对应值 ReadData1 为 8；Rt 为 2 号寄存器，对应值 ReadData2 为 2。ALU 运算结果 Y 为 0。写入寄存器 WriteReg 为 4 号寄存器，写入数据 WriteData 为 0。指令执行正确($4 = $5 & $2 = 8 & 2 = 0)。

6、 or $8,$4,$2



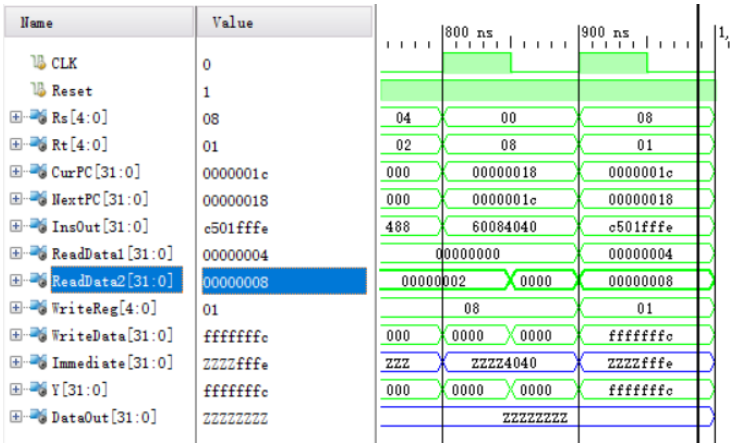
Rs 为 4 号寄存器，对应值 ReadData1 为 0；Rt 为 2 号寄存器，对应值 ReadData2 为 2。ALU 运算结果 Y 为 2。写入寄存器 WriteReg 为 8 号寄存器，写入数据 WriteData 为 2。指令执行正确($8 = $4 | $2 = 0 | 2 = 2)。

7、 sll $8,$8,1



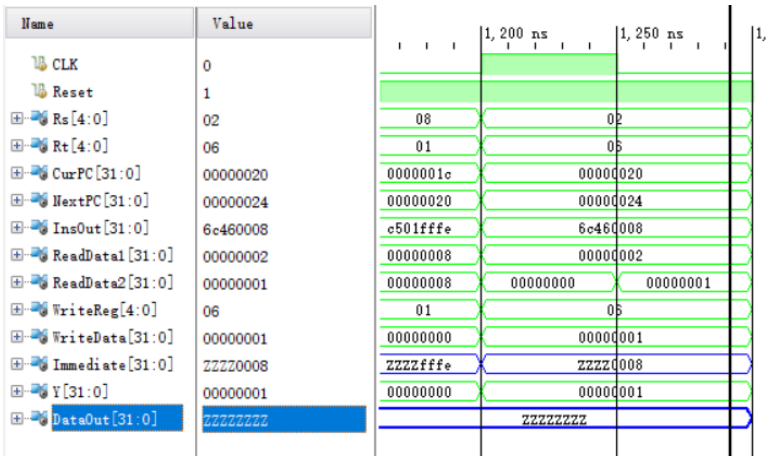
Rt 为 8 号寄存器，对应值 ReadData2 在时钟下降沿时由 2 变为 4。指令执行正确($8 = $8 << 1 = 2 << 1 = 4)。

8、 bne $8,$1,-2 (≠,转18)



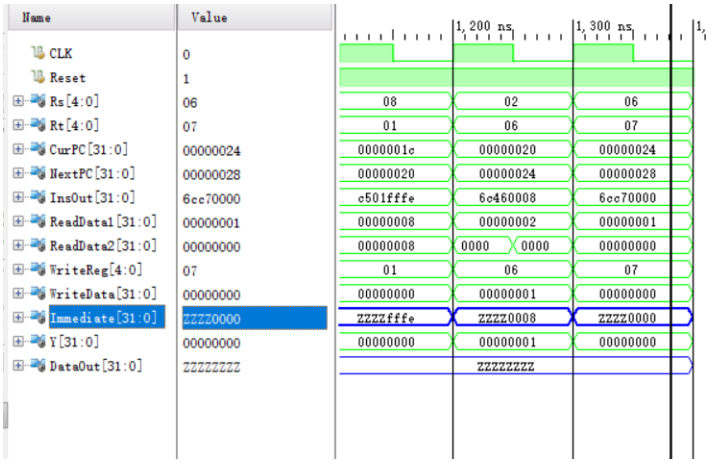
当前指令地址 CurPC 为 1C，下一条 NextPC 为 18，符合预期分支跳转情况。立即数 Immediate 为 -2。Rs 为 8 号寄存器，对应值 ReadData1 保持为 4；Rt 为 1 号寄存器，对应值为 8。指令执行正确($8 ！ = $1 <-> 4 ！ = 8，PC = PC - 2)。

9、 slti $6,$2,8



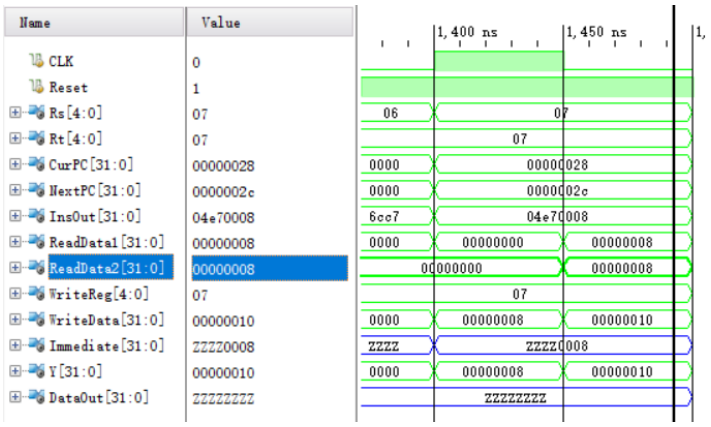
Rs 为 2 号寄存器，对应值 ReadData1 为 2，立即数 Immediate 为 8，Rt 为 6 号寄存器，对应值 ReadData2 为 1。ALU 运算结果 Y 为 1。指令执行正确($2 < Immediate => 2 < 8 => $6 = 1)。

10、 slti $7,$6,0



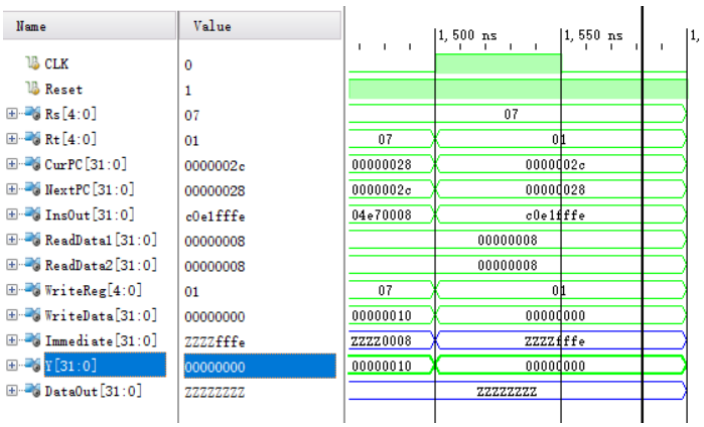
Rs 为 6 号寄存器，对应值 ReadData1 为 1，立即数 Immediate 为 0，Rt 为 7 号寄存器，对应值 ReadData2 为 0。ALU 运算结果 Y 为 0。指令执行正确($6 > Immediate => 1 > 0 => $7 = 0)。

11、 addi $7,$7,8



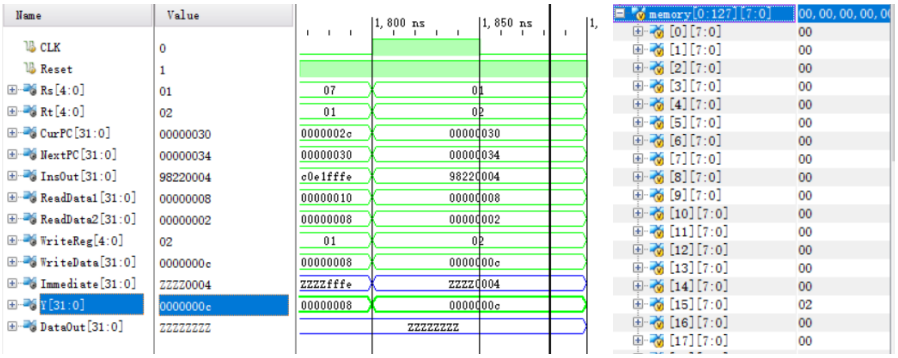
立即数 Immediate 为 8。Rs，Rt 均为 7 号寄存器，对应值 ReadData1，ReadData2 在时钟下降沿时由 0 变为 8。ALU 运算结果 Y 为 8。写入寄存器 WriteReg 为 7 号寄存器，写入数据 WriteData 为 8。指令执行正确($7 = $7 + 8 = 0 + 8 = 8)。

12、 beq $7,$1,-2 (=,转28)



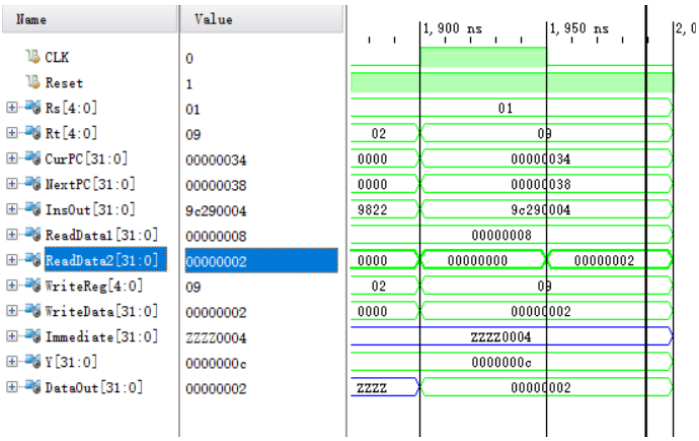
当前指令地址 CurPC 为 2C，下一条 NextPC 为 28，符合预期分支跳转情况。立即数 Immediate 为 -2。Rs 为 7 号寄存器，对应值 ReadData1 为 8；Rt 为 1 号寄存器，对应值为 8。指令执行正确($7 = $1 <-> 8 = 8，PC = PC - 2)。

13、 sw $2,4($1)



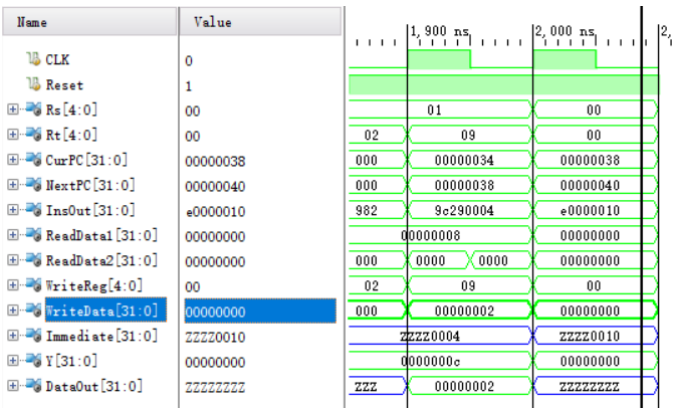
立即数 Immediate 为 4。Rs 为 1 号寄存器，对应值 ReadData1 为 8；Rt 为 2 号寄存器，对应值 ReadData2 为 2。ALU 运算结果为 12。由于采用 8 位一字节的大端模式存储数据，所以在 12，13，14，15 号内存内写入数据 2。查看内存如图，指令执行正确(memory[$1 + (sign-extend)immediate] = $2 => memory[8 + 4] = 2 => memory[12:15] = 2)。

14、 lw $9,4($1)



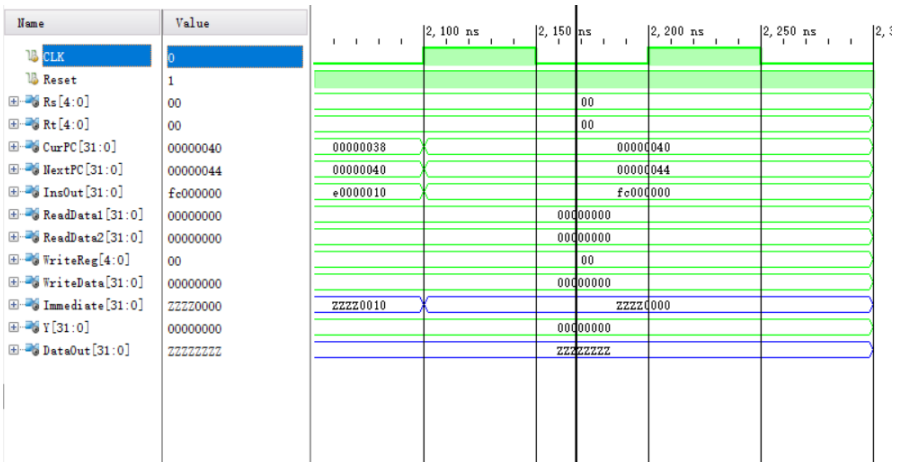
立即数 Immediate 为 4。Rs 为 1 号寄存器，对应值 ReadData1 为 8；Rt 为 9 号寄存器，对应值 ReadData2 在时钟下降沿由 0 变为 2。ALU 运算结果为 12。由于采用 8 位一字节的大端模式存储数据，所以在 12，13，14，15 号内存读出数据 2，写入 Rt，9号寄存器，指令执行正确($9 = memory[$1 + (sign-extend)immediate] => $9 = memory[8 + 4] => $9 = memory[12:15] = 2)。

15、 j 0x00000040



当前指令地址 CurPC 为 38，下一条 NextPC 为 40，符合预期分支跳转情况。

16、 halt



指令地址由 38 跳到 40 后，执行 halt 指令。PC 在两个时钟周期内，均维持 40 不变，符合预期。指令执行正确。

C. 烧板设计：

在这里简述思路与大致实现过程，代码略去，因为这个部分代码并非难以实现。

1、 时钟分频

将Basys3 实验板上的时钟频率进行分频，实现方法采用时钟上升沿触发初始为 0 的二进制数加一，然后选择合适位数下的二进制位作为选择的时钟频率(类似二分法)。

2、 数据选择

输入选择信号，输出选择的数据，如 00 => PC ： NextPC。四选一选择器。

3、 数码管显示

输入选择数据，并选择每个数码管显示不同数字数据。同时，在选择的时钟频率下，对四个数码管进行“刷新”，使得四个数码管被同时点亮。这里的时钟频率，决定了整个时钟频率，因为频率太高或太低，数码管的显示都会出现问题，太快看不清或太慢而逐个显示。

4、 按键仿 CPU 周期

每按下一次按键，将信号值取反，做为 CPU 时钟。这里要处理按键的防抖动问题。通过查询资料按键的防抖动可以采用在时钟内，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，则输出正信号或负信号，这样可以避免按键信号的抖动问题，这样输出的信号则一定是稳定状况下的按键信号。

D. 烧板验证：

图片按照顺时针方向，依次为：

PC：NextPC，RS：RsData，RT：RTData，ALU 结果 Y：DB总线数据

1、 addi $1,$0,8

2、 ori $2,$0,2

3、 add $3,$2,$1

4、 sub $5,$3,$2

5、 and $4,$5,$2

6、 or $8,$4,$2

7、 sll $8,$8,1

8、 bne $8,$1,-2 (≠,转18)

此时已经进行过一次跳转，$8 (RS)的值为 8。

9、 slti $6,$2,8

10、 slti $7,$6,0

11、 beq $7,$1,-2 (=,转28)

12、 sw $2,4($1)

13、 lw $9,4($1)

14、 j 0x00000040

15、 halt

六. 实验心得

开始设计 CPU 的时候，感觉无从下手，但在仔细数据通路图后，知晓了 CPU 的设计方法，类似与面向对象的设计：将 CPU 的各个模块进行分离，实现每个模块的时候，只关注这个模块本身的输入输出与逻辑功能。设计完成所有模块后，就是设计顶层文件，将模块整合在一起，完成整体 CPU 的设计。

在进行模块设计的时候，感触最深的就是控制单元的设计了。针对不同指令的输入给出真值表。开始的时候，直接在代码文件中使用指令二进制码进行真值选择，然后每次针对一个输出，就需要仔细观察二进制码，避免出错。这样书写实在是太不方便了，于是在网上查找了 Verilog 的文件引入与宏定义资料后，自行设计了 head.v 文件，使用宏将指令二进制码定义在文件中，之后在使用这些指令二进制码的文件中引入该文件并使用宏定义名替代，大大提高了书写代码的准确性，避免硬编码问题。

在寄存器组中，传入的是寄存器的地址，虽然代码上看来是数组下标，但应该区分这两者，所以在模块中设计变量名时，对于地址变量，大部分会写明 Address。我们采用的是 MIPS 的指令集，所以寄存器组中有 32 个寄存器。虽然无法做到真的如同 MIPS 中对应寄存器对应用途使用，但针对 0 号寄存器，进行了保护：写入数据时，会判断是否为 0 号寄存器，若是，则禁止写入。

第一次仿真的时候，输出全是高阻抗状态或者不确定值。但是指令地址与指令二进制码都与测试文件中相同。进行了模块的重新审视，但未发现问题。最终在顶层模块中发现问题所在：对于一个变量，如果有两个变量同时对其进行赋值，则其值会变为不确定。即，我的顶层模块中，出现了将一个变量作为某一模块的输出，同时将另一个变量赋值给了它。所以直接在顶层模块中，使用大写变量名作为它的输入输出，同时，使用小写变量名重新定义了所有的模块使用到的变量作为局部变量，让局部变量在内部运转，并将局部变量通过 assign 语句实时将值赋值给输出变量。这样之后的仿真输出大部分都正确了。

对于仿真输出确定之后，进行每条指令的 debug。在 j 跳转指令处出现了问题。j 跳转指令使用下面的赋值式：pc <－{(pc+4)[31..28],addr[27..2],2{0}}。从二进制数的方面考虑，25位的 addr 左移两位补零，然后我们截取的长度仍旧是 25 位，所以直接将 addr 赋值即可。但是这么做的话，仿真时跳转的地址就不对了，这个问题不知道在什么地方出错，并没有解决，改为左移后截取则跳转正常。

在烧板的时候，需要处理按键防抖动。但在设计时，开始忘记了处理这个问题，直接采取按键直接取反，但貌似时钟频率取值比较高且符合要求，这么处理的代码在进行验证的时候依旧可以运行。然后在重新查看烧板验证的 pdf 后发现了这个问题，于是通过查询资料得知，按键的防抖动可以采用在时钟上升沿时，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，则输出正信号或负信号，这样可以避免按键信号的抖动问题，且输出的信号则一定是稳定状况下的按键信号。