成 绩 ：

实验三 ： 多周期CPU设计与实现

一. 实验目的

1. 认识和掌握多周期数据通路图的构成、原理及其设计方法；
2. 掌握多周期 CPU 的实现方法，代码实现方法；
3. 编写一个编译器，将 MIPS 汇编程序编译为二进制机器码；
4. 掌握多周期 CPU 的测试和实现方法。

二. 实验内容

设计一个多周期 CPU，至少实现以下指令功能操作，需设计的指令与格式如下：

==>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<－rs + rt。

（2）sub rd, rs, rt

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

功能：rd<－rs － rt。

（3）addi rt, rs, **immediate**

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

功能：rt<－rs + (sign-extend)**immediate**。

==>逻辑运算指令

（4）and rd , rs , rt

010000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs & rt；逻辑与运算。

（5）andi rt , rs ,**immediate**

010001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs & (zero-extend)immediate；immediate 做“0”扩展再“与”。

（6）ori rt , rs ,**immediate**

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

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

（7）xori　rt , rs , **immediate**

010011	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs ⊕ (zero-extend)immediate；immediate 做“0”扩展再“异或”。

==>移位指令

（8）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

（9）slti　rt, rs,immediate　　带符号

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

功能：if (rs <(sign-extend)**immediate)**　rt =1 else　rt=0，具体看表 2ALU 运算功能表，带符号。

（10）slt　rd, rs, rt　　带符号

100111	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：if (rs<rt)　rd =1 else　rd=0，具体请看表 2 ALU 运算功能表，带符号。

==>存储器读写指令

（11）sw　rt, **immediate(**rs)

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

功能：memory[rs+ (sign-extend)immediate]<－rt。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。

（12）lw　rt, **immediate**(rs)

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

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

(13)beq　rs,rt, **immediate** (说明:**immediate 从 pc+4 开始和转移到的指令之间间隔条数**)

110100	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。

(14)bne　rs,rt,**immediate** (说明:**immediate 从 pc+4 开始和转移到的指令之间间隔条数**)

110101	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。

（15）bltz　rs,**immediate**

110110	rs(5 位)	00000	**immediate**

功能：if(rs<$0) pc←pc + 4 +((sign-extend)**immediate** <<2)　else pc ←pc + 4。

==>跳转指令

（16）j　addr

111000	addr[27:2]

功能：pc <－{(pc+4)[31:28],addr[27:2],2'b00}，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。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。

（17）jr　rs

111001	rs(5 位)	未用	未用	reserved

功能：pc <－ rs，跳转。

==>调用子程序指令

（18）jal　addr

111010	addr[27:2]

功能：调用子程序，pc <－ {(pc+4)[31:28],addr[27:2],2'b00}；$31<－pc+4，返回地址设置；子程序返回，需用指令 jr $31。跳转地址的形成同 j addr 指令。

==>停机指令

（19）halt (停机指令)

111111	00000000000000000000000000(26 位)

不改变 pc 的值，pc 保持不变。

三．实验原理

多周期 CPU 指的是将整个 CPU 的执行过程分成几个阶段，每个阶段用一个时钟去完成，然后开始下一条指令的执行，而每种指令执行时所用的时钟数不尽相同。CPU 在处理指令时，一般需要经过以下几个阶段：

(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，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。

实验按照这五个阶段进行设计，一条指令的执行最长需要五个(小)时钟周期才能完成，但具体情况要根据各指令而定，有些指令可能不需要五个时钟周期。

图 1 多周期 CPU 指令处理过程

MIPS 指令的三种格式：

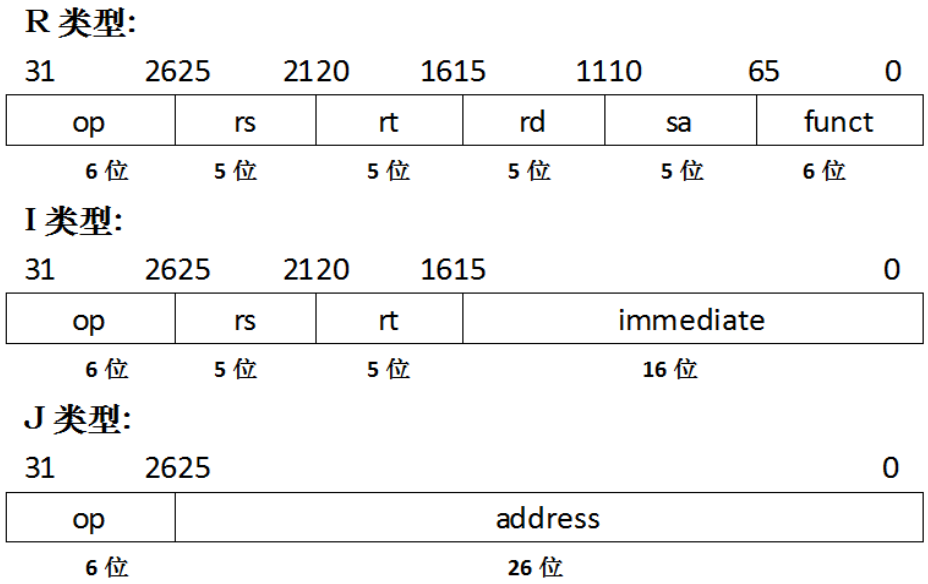

图 2 MIPS 指令的三种格式

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（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。

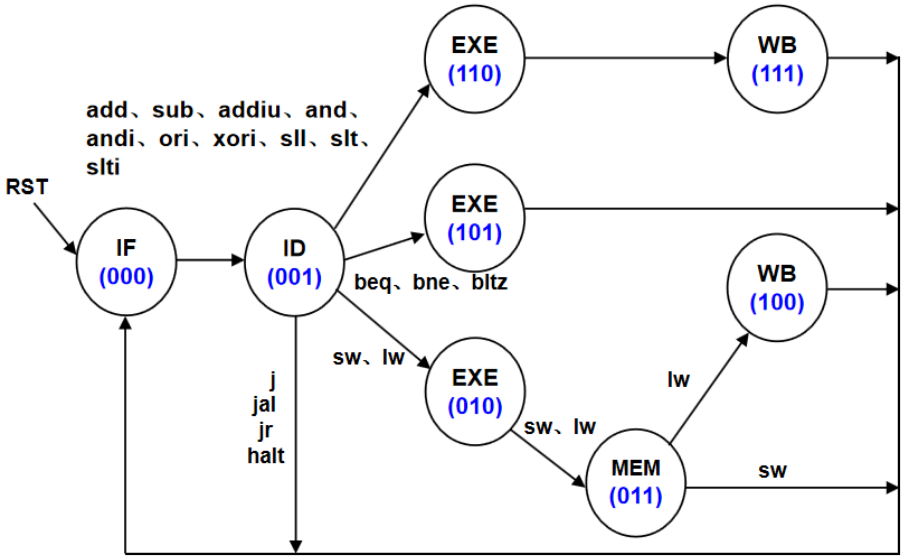

图 3 多周期 CPU 状态转移图

状态的转移有的是无条件的，例如从 sIF 状态转移到 sID 就是无条件的；有些是有条件的，例如 sEXE 状态之后不止一个状态，到底转向哪个状态由该指令功能，即指令操作码决定。每个状态代表一个时钟周期。

图 4 是多周期 CPU 控制部件的电路结构，三个 D 触发器用于保存当前状态，是时序逻辑电路，RST 用于初始化状态“000”，另外两个部分都是组合逻辑电路，一个用于产生下一个阶段的状态，另一个用于产生每个阶段的控制信号。从图上可看出，下个状态取决于指令操作码和当前状态；而每个阶段的控制信号取决于指令操作码、当前状态和反映运算结果的状态 zero 标志和符号 sign 标志。

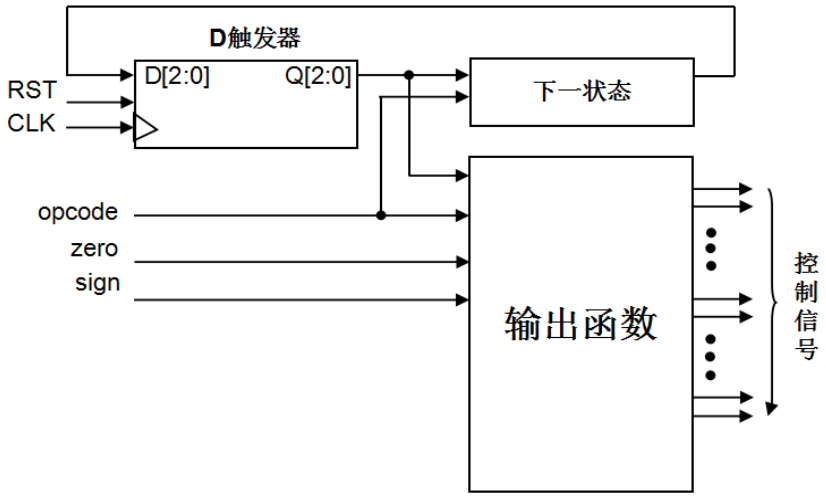

图 4 多周期 CPU 控制部件的原理结构图

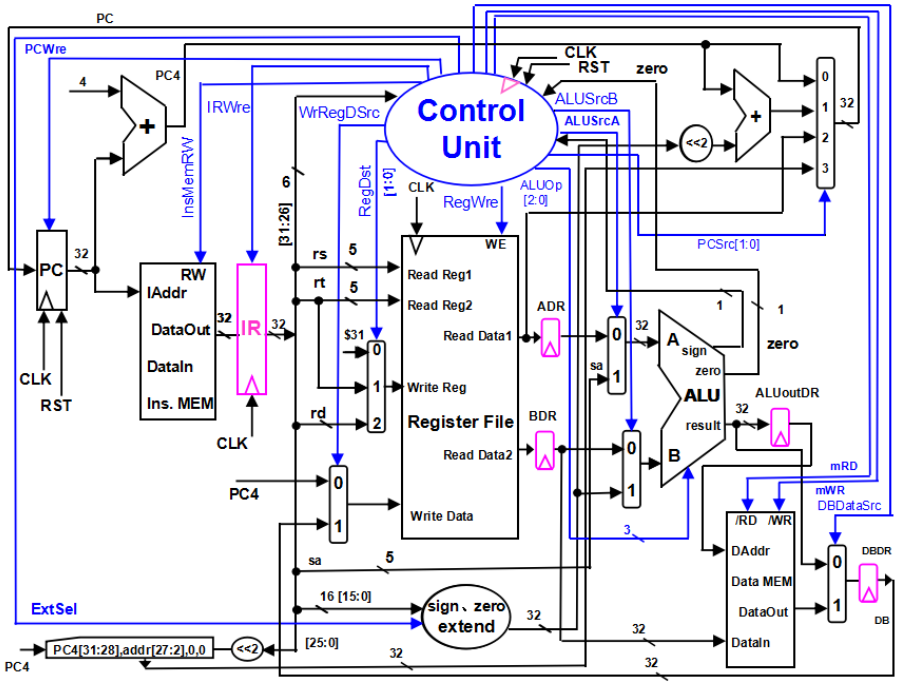

图 5 多周期 CPU 数据通路和控制线路图

图 5 是一个简单的基本上能够在多周期 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，给出寄存器地址（编号），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。图中控制信号功能如表 1 所示，表 2 是 ALU 运算功能表。

特别提示，图上增加 IR 指令寄存器，目的是使指令代码保持稳定，pc 写使能控制信号 PCWre，是确保 pc 适时修改，原因都是和多周期工作的 CPU 有关。ADR、BDR、ALUoutDR、DBDR 四个寄存器不需要写使能信号，其作用是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟。

表 1 控制信号作用

控制信号名	状态“0”	状态“1”
RST	对于 PC，初始化 PC 为程序首地址	对于 PC，PC 接收下一条指令地址
PCWre	PC 不更改，相关指令：halt，另外，除‘000’状态外，其余状态慎改 PC 的值。	PC 更改，相关指令：除指令 halt 外，另外，在‘000’状态修改 PC 的值合适。
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addiu、and、andi、ori、xori、slt、slti、sw、lw、beq、bne、bltz	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、and、slt、sll、beq、bne、bltz	来自 sign 或 zero 扩展的立即数，相关指令：addiu、andi、ori、xori、slti、lw、sw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、sub、addiu、and、andi、ori、xori、sll、slt、slti	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、j、sw、jr、halt	寄存器组寄存器写使能，相关指令：add、sub、addiu、and、andi、ori、xori、sll、slt、slti、lw、jal
WrRegDSrc	写入寄存器组寄存器的数据来自 pc+4(pc4），相关指令：jal，写 $31	写入寄存器组寄存器的数据来自 ALU 运算结果或存储器读出的数据，相关指令：add、addiu、sub、and、andi、ori、xori、sll、slt、slti、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	存储器输出高阻态	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
IRWre	IR(指令寄存器)不更改	IR 寄存器写使能。向指令存储器发出读指令代码后，这个信号也接着发出，在时钟上升沿，IR 接收从指令存储器送来的指令代码。与每条指令都相关。
ExtSel	(zero-extend)immediate，相关指令：andi、xori、ori;	(sign-extend)immediate，相关指令：addiu、slti、lw、sw、beq、bne、bltz;
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addiu、sub、and、andi、ori、xori、slt、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0); 01：pc<－pc+4+(sign-extend)immediate ×4，相关指令：beq(zero=1)、bne(zero=0)、bltz(sign=1); 10：pc<－rs，相关指令：jr; 11：pc<－{pc[31:28],addr[27:2],2'b00}，相关指令：j、jal;	
RegDst [1..0]	写寄存器组寄存器的地址，来自： 00：0x1F($31)，相关指令：jal，用于保存返回地址（$31<-pc+4）； 01：rt 字段，相关指令：addiu、andi、ori、xori、slti、lw; 10：rd 字段，相关指令：add、sub、and、slt、sll; 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，指令地址输入端口

DataOut，存储器数据输出端口

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，数据地址输入端口

DataIn，存储器数据输入端口

DataOut，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，其地址输入端口（rt、rd）

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

IR：指令寄存器，用于存放正在执行的指令代码

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<B，不带符号
110	Y=(((A<B)&&(A[31]==B[31]))|| ((A[31]==1&& B[31] == 0))) ? 1:0	比较 A<B，带符号
111	Y = A ⊕ B	异或

四. 实验器材

PC机一台，BASYS 3 实验板一块，Xilinx Vivado 开发软件一套。

五. 实验过程与结果

(一) 编译器

编译器用于将MIPS汇编程序编译为二进制机器码。在本实验中，将针对只含有R型、I型和J型指令的MIPS代码编写编译器，代码见附录或compiler.py。（程序运行命令为：**python compiler.py <input_file> <output_file>**；如：*python compiler.py mips_code.asm machine_code.txt*）

(二) CPU设计

分析单周期CPU与多周期CPU数据通路图可以发现他们在设计上有如下异同点：

- **相同点**

  PC模块、Register File模块、ALU模块、符号扩展模块和Data MEM模块。

- **不同点**

  (1) 多周期CPU中增加了用于使指令代码保持稳定的IR指令寄存器和ADR、BDR、ALUoutDR、DBDR四个用于切分数据通路的寄存器。

  (2) 多周期CPU的Control Unit模块需要时钟输入CLK进行控制；控制模块按照各指令对应的状态转移图输出对应的控制信号。

  (3) 多周期CPU的PC Adder模块增加一个状态，用于jr指令将程序从子程序跳回主程序（PC <- rs）。

  (4) 多周期CPU部分模块输入端的变化（适配jal指令）

  1) Register File输入端端口Write Reg新增了寄存器$31的输入，用于执行jal指令时，将主程序下一地址（PC+4）暂存入寄存器$31中

  2) Register File输入端端口Write Data新增了输入选择端PC+4，用于执行jal指令时，输入主程序下一地址。

依照图 4 多周期CPU控制部件的原理结构图，将CPU划分为十一个功能模块（底层模块）和一个用于连接各模块的顶层模块（CPU），其中将部分数据选择器（MUX）整合至功能模块中，化简了CPU的总体设计。CPU各模块划分结果如图 6所示。

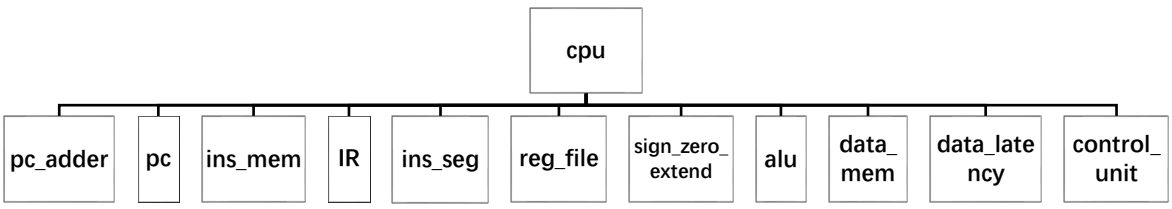

图 6 多周期 CPU 模块划分图

1. 底层模块设计

(1) pc_adder

模块功能：根据控制信号PCSrc，计算得到下一个pc地址。

实现思路：使用时钟下降沿作为敏感信号，触发计算下一条pc地址的功能。

主要实现代码

```
always@(negedge CLK or negedge Reset) begin
    if(Reset == 0) nextPC <= 0;
    else begin
        pc <= curPC + 4;
        case(PCSrc)
            2'b00: nextPC <= curPC + 4;
            2'b01: nextPC <= curPC + 4 + immediate * 4;
            2'b10: nextPC <= ReadData;
            2'b11: nextPC <= {pc[31:28],addr,2'b00};
        endcase
    end
end
```

(2) pc

模块功能：本模块是时序逻辑。根据控制信号PCWre，判断pc是否改变以及根据Reset信号判断pc是否重置。

实现思路：用时钟上升沿和Reset作为敏感信号，在时钟上升沿改变pc。

主要实现代码

```
always@(posedge CLK or posedge Reset) begin
    if(!Reset) curPC <= 0;
    else begin
        if(PCWre == 1) curPC <= nextPC;
        else curPC <= curPC;
    end
end
```

(3) ins_mem

模块功能：本模块是组合逻辑。当控制信号InsMemRW为1（读信号）时，根据当前pc（IAddr），读取指令寄存器中相对应的指令。（本次实验中，测试指令从test_data.txt一次性写入，不涉及单条指令写入）

实现思路

1) CPU初始化时，将指令读入并预存至内含128个字节的ROM中；
2) 使用PC作为敏感信号，PC改变时，从ROM中读出地址对应指令；

3) 即该模块输入一个32位长的地址，输出对应地址中的32位机器指令。

主要实现代码

```
reg [7:0] rom[0:127];
initial $readmemb("…(link url)", rom);

always@(IAddr or InsMemRW) begin
    if(InsMemRW) begin
        IDataOut[7:0] = rom[IAddr + 3];
        IDataOut[15:8] = rom[IAddr + 2];
        IDataOut[23:16] = rom[IAddr + 1];
        IDataOut[31:24] = rom[IAddr];
    end
end
```

(4) ins_seg

模块功能：对ins_mem读出指令进行分割，获取不同类型指令（R类型、I类型、J类型）对应的指令信息。

主要实现代码

```
always@(instruction) begin
    // R type instruction
    op = instruction[31:26];
    rs = instruction[25:21];
    rt = instruction[20:16];
    rd = instruction[15:11];
    sa = instruction[10:6];
    // I type instruction
    immediate = instruction[15:0];
    // J type instruction
    addr = instruction[25:0];
end
```

(5) IR

模块功能：使指令代码保持稳定。

实现思路：依据信号量IRWre，在时钟上升沿对IR寄存器进行写入。

主要实现代码

```
always@(posedge CLK) begin
    if(IRWre == 1)  out_data = in_data;
end
```

(6) reg_file

模块功能：本模块是组合逻辑。寄存器存储器。按控制信号RegWre对寄存器进行读或写操作。同时，该模块整合实现了与Write Reg写入端口相连的数据选择器。

实现思路

1) CPU初始化时，初始化内含32个32位寄存器的寄存器组regFile；
2) 读操作（RegWre=0），按寄存器编号（rs和rt）输出寄存器的值；
3) 写操作（RegWre=1），根据控制信号RegDst选择被写入寄存器的编号（rt或rd），时钟上升沿将数据（Write Data）写入对应寄存器；
4) 根据控制信号WrRegDSrc选择Write Reg端口输入的寄存器。

主要实现代码

```
reg [31:0] regFile[0:31];
integer i;
initial begin
    for(i = 0; i < 32; i = i + 1) regFile[i] <= 0;
end
// rs -> ReadReg1; rt -> ReadReg2
always@(ReadReg1 or ReadReg2) begin
    ReadData1 = regFile[ReadReg1];
    ReadData2 = regFile[ReadReg2];
end
// MUX
always@(RegDst) begin
    if(RegDst == 2'b00) WriteReg = 5'b11111;
    else if(RegDst == 2'b01) WriteReg = rt;
    else if(RegDst == 2'b10) WriteReg = rd;
    else WriteReg = 5'bzzzzz;
end

always@(posedge CLK) begin
    if(WriteReg!=0 && RegWre) regFile[WriteReg] <= WriteData;
end
```

(7) sign_zero_extend

模块功能：根据控制指令ExtSel，对立即数进行“0”扩展或符号扩展。

主要实现代码

```
assign extendImmediate[15:0] = immediate;
assign extendImmediate[31:16] = (ExtSel == 1 ? (immediate[15] ?
16'hffff : 16'h0000) : 16'h0000);
```

(8) alu

模块功能：算术逻辑单元。根据控制信号ALUOp，对两个输入数据（A、B）进行相对应算术逻辑运算，并输出运算结果和状态量（zero、sign）。

实现思路：

1) alu模块中集成了两个数据选择器，选择器根据控制信号ALUSrcA和控制信号ALUSrcB分别判断alu两个输入端口A、B对应的输入数据；
2) 根据表 2 ALU运算功能表完成设计。

主要实现代码

```
always@(*) begin
    A = (ALUSrcA == 0) ? ReadData1 : sa;   // MUX
    B = (ALUSrcB == 0) ? ReadData2 : extend;
    case(ALUOp)
        3'b000: result = A + B;
        3'b001: result = A - B;
        3'b010: result = B << A;
        3'b011: result = A | B;
        3'b100: result = A & B;
        3'b101: result = (A < B) ? 1 : 0;
        3'b110: result=((A<B)&&(A[31]==B[31]))||((A[31]==1&& B[31]==0)?1:0);
        3'b111: result = A ^ B;
    endcase
    zero = (result == 0) ? 1 : 0;
    sign = result[31];
end
```

(9) data_mem

模块功能：数据存储器。依照控制信号mRD和mWR，对数据存储器进行读或写操作。

实现思路

1) CPU初始化时，初始化内含128个8位寄存器的数据存储器ram；
2) 按照控制信号进行读操作或写操作；
3) data_mem中集成了一个数据选择器，该选择器根据控制信号DBDataSrc，决定写回寄存器的数据（ALU result或DataOut）。

主要实现代码

```
reg [7:0] ram[0:127];
integer i;
initial for(i = 0;i < 128; i = i + 1) ram[i] <= 0;
```

```
// Write Operation
always@(negedge CLK) begin
    if(mWR == 1) begin
        ram[DAddr + 3] = DataIn[7:0];
        ram[DAddr + 2] = DataIn[15:8];
        ram[DAddr + 1] = DataIn[23:16];
        ram[DAddr] = DataIn[31:24];
    end
end
// Read Operation
always@(mRD or DAddr or DBDataSrc) begin
    DataOut[7:0] = mRD ? ram[DAddr + 3] : 8'bz;
    DataOut[15:8] = mRD ? ram[DAddr + 2] : 8'bz;
    DataOut[23:16] = mRD ? ram[DAddr + 1] : 8'bz;
    DataOut[31:24] = mRD ? ram[DAddr] : 8'bz;
    // DataOut is the output of data_mem
    // DAddr is the output of alu
    DB = DBDataSrc ? DataOut : DAddr;
end
```

(10) data_latency（寄存器ADR、BDR、ALUoutDR、DBDR）

模块功能：本模块是时序逻辑。切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟。

实现思路：本质是D-触发器，在时钟下降沿将相应数据写入触发器中。四个模块功能相同，使用同一模块实例化得到。

实现代码

```
module data_latency(
    input CLK,
    input [31:0] in_data,
    output reg [31:0] out_data
    );

    initial out_data = 0;

    always @(negedge CLK) begin
        out_data = in_data;
    end
endmodule
```

(11) control_unit

模块功能：控制单元。根据指令状态转移图对各个模块发出正确控制信号。

实现思路

将控制信号分为“写入”和“读出”两类。

1) 首先考虑涉及写入操作的控制信号PCWre、IRWre、mWR和RegWre:
   a) PCWre：除指令为halt停机，pc不发生改变外（PCWre=0），pc均在在sIF状态的时钟上升沿写入下一指令地址。因此应该让PCWre信号在前一个状态的时钟下降沿处直到IF状态的时钟下降沿处保持有效；
   b) IRWre：IR在ID状态的时钟上升沿写入前一状态IF读取的32位指令。因此应该让IRWre信号在IF状态的时钟下降沿处直到ID状态的时钟下降沿处保持有效；
   c) mWR：存储器写使能信号，存储器只在sw指令的MEM状态将数据写入，mWR控制信号保持有效，其他状态mWR控制信号均保持无效；
   d) RegWre：寄存器写使能信号，寄存器只在R型指令和I型指令的WB状态写入(回写)，同时jal指令的ID状态也需要寄存器写入。因此，RegWre信号只在这两种状态保持有效，其他状态mWR控制信号均保持无效。
2) 然后再考虑读出信号，根据表 1 控制信号作用综合分析得出表 3；

表 3 控制信号量与控制指令对照表

指令	ALU SrcA	ALU SrcB	DBDataSrc	WrRegDSrc	mRD	ExtSel	PCSrc [1:0]	RegDst [1:0]	ALUOp [2:0]
add	0	0	0	1	0	0	00	10	000
sub	0	0	0	1	0	0	00	10	001
addi	0	1	0	1	0	1	00	01	000
and	0	0	0	1	0	0	00	10	100
andi	0	1	0	1	0	1	00	01	100
ori	0	1	0	1	0	0	00	01	011
xori	0	1	0	1	0	0	00	01	111
sll	1	0	0	1	0	0	00	10	010
slti	0	1	0	1	0	1	00	01	110
slt	0	0	0	1	0	0	00	10	110
sw	0	1	0	1	0	1	00	00	000

lw	0	1	1	1	1	1	00	01	000
beq	0	0	0	1	0	1	01(zero) /00	00	001
bne	0	0	0	1	0	1	00(zero) /01	00	001
bltz	0	0	0	1	0	1	01(sign) /00	00	000
j	0	0	0	1	0	0	11	00	000
jr	0	0	0	1	0	0	10	00	000
jal	0	0	0	1	0	0	11	00	000
halt	0	0	0	1	0	0	00	00	000

主要实现代码：为节约篇幅，此处仅给出PCWre控制信号、IRWre控制信号和PCSrc控制信号相关代码（完整代码见control_unit.v），其他控制信号实现代码结构类似。

```
//PCWre, IRWre
always@(negedge CLK) begin
    if(state == 3'b000) IRWre = 1;
    else IRWre = 0;
    case(state)
        3'b111, 3'b101, 3'b100: PCWre = 1;
        3'b011: PCWre = (op == SW ? 1 : 0);
        3'b001: PCWre = ((op == J||op == JAL||op == JR) ? 1 : 0);
        default: PCWre = 0;
    endcase
end
//PCSrc
always@(op or zero or sign or state) begin
    if(op == J || op== JAL) PCSrc = 2'b11;
    else if(op == JR) PCSrc = 2'b10;
    else if(op ==BEQ && zero == 1) PCSrc = 2'b01;
    else if(op == BNE && zero == 0) PCSrc = 2'b01;
    else if(op == BLTZ && sign == 1) PCSrc = 2'b01;
    else PCSrc = 2'b00;
end
//… other states
```

2. 顶层模块设计

在顶层模块中，实例化各个底层模块，并用线（wire）将各个模块按照图 5 多周期CPU数据通路和控制线路图连接（具体代码见cpu.v）。

图 7 多周期 CPU 文件结构

(三) 仿真验证CPU正确性

1. CPU程序代码测试

根据所给文档提供的指令（表 4，该表已将所有分支跳转情况展开，即表中指令按顺序执行）。对指令实际执行过程中各个寄存器变化、指令跳转情况等信息进行综合分析，判断各模块是否正常运行，完成本次测试。

表 4 多周期 CPU 测试指令段

地址	汇编程序	指令代码					
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码	
0x00000000	addi $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002
0x00000008	xori $3,$2,8	010011	00010	00011	0000 0000 0000 1000	=	4C430008
0x0000000C	sub $4,$3,$1	000001	00011	00001	00100 00000 000000	=	04612000
0x00000010	and $5,$4,$2	010000	00100	00010	00101 00000 000000	=	40822800
0x00000014	sll $5,$5,2	011000	00000	00101	00101 00010 000000	=	60052880
0x00000018	beq $5,$1,-2 (=,转 14)	110100	00101	00001	1111 1111 1111 1110	=	D0A1FFFE
0x00000014	sll $5,$5,2	011000	00000	00101	00101 00010 000000	=	60052880
0x00000018	beq $5,$1,-2	110100	00101	00001	1111 1111 1111 1110	=	D0A1FFFE
0x0000001C	jal 0x0000050	111010	00000	00000	0000 0000 0001 0100	=	E8000014
0x00000050	sw $2,4($1)	110000	00001	00010	0000 0000 0000 0100	=	C0220004
0x00000054	lw $13,4($1)	110001	00001	01101	0000 0000 0000 0100	=	C42D0004
0x00000058	jr $31	111001	11111	00000	0000 0000 0000 0000	=	E7E00000
0x00000020	slt $8,$13,$1	100111	01101	00001	01000 00000 000000	=	9DA14000
0x00000024	addi $14,$0,-2	000010	00000	01110	1111 1111 1111 1110	=	080EFFFE

0x00000028	slt $9,$8,$14	100111	01000	01110	01001 00000 000000	=	9D0E4800
0x0000002C	slti $10,$9,2	100110	01001	01010	0000 0000 0000 0010	=	992A0002
0x00000030	slti $11,$10,0	100110	01010	01011	0000 0000 0000 0000	=	994B0000
0x00000034	add $11,$11,$10	000000	01011	01010	01011 00000 000000	=	016A5800
0x00000038	bne $11,$2,-2 (≠,转 34)	110101	01011	00010	1111 1111 1111 1110	=	D562FFFE
0x00000034	add $11,$11,$10	000000	01011	01010	01011 00000 000000	=	016A5800
0x00000038	bne $11,$2,-2	110101	01011	00010	1111 1111 1111 1110	=	D562FFFE
0x0000003C	addi $12,$0,-2	000010	00000	01100	1111 1111 1111 1110	=	080CFFFE
0x00000040	addi $12,$12,1	000010	01100	01100	0000 0000 0000 0001	=	098C0001
0x00000044	bltz $12,-2 (<0,转 40)	110110	01100	00000	1111 1111 1111 1110	=	D980FFFE
0x00000040	addi $12,$12,1	000010	01100	01100	0000 0000 0000 0001	=	098C0001
0x00000044	bltz $12,-2	110110	01100	00000	1111 1111 1111 1110	=	D980FFFE
0x00000048	andi $12,$2,2	010001	00010	01100	0000 0000 0000 0010	=	444C0002
0x0000004C	j 0x000005C	111000	00000	00000	0000 0000 0001 0111	=	E0000017
0x0000005C	halt	111111	00000	00000	0000000000000000	=	FC000000

2. 使用Vivado进行仿真验证

创建仿真文件test_single_cycle_cpu.v，在仿真文件中使用如下输入和输出实例化顶层模块cpu：

表 5 实例化 CPU 的输入和输出

输入	CLK		CPU 时钟信号	
	RST		CPU 复位信号，0 为复位	
输出	curPC	当前 PC 地址	rd	rd 寄存器编号
	nextPC	下一条指令地址	IR_out	IR 指令寄存器输出
	ins	当前 PC 地址对应指令	ADR_out	ADR 寄存器输出
	op	指令操作码	BDR_out	BDR 寄存器输出
	rs	rs 寄存器编号	ALUoutDR_out	ALUoutDR 寄存器输出
	rt	rt 寄存器编号	DBDR_out	DBDR 寄存器输出

仿真文件关键代码如下：

```
initial begin
    // 初始化 CLK 信号和 RST 信号
    CLK = 0;
    RST = 0;
    #50 RST = 1;
        forever #50 CLK = ~CLK;  // 仿真 CLK 周期为 100ns
end
```

以上步骤准备就绪后，使用Vivado开始仿真验证工作。仿真结果如下：

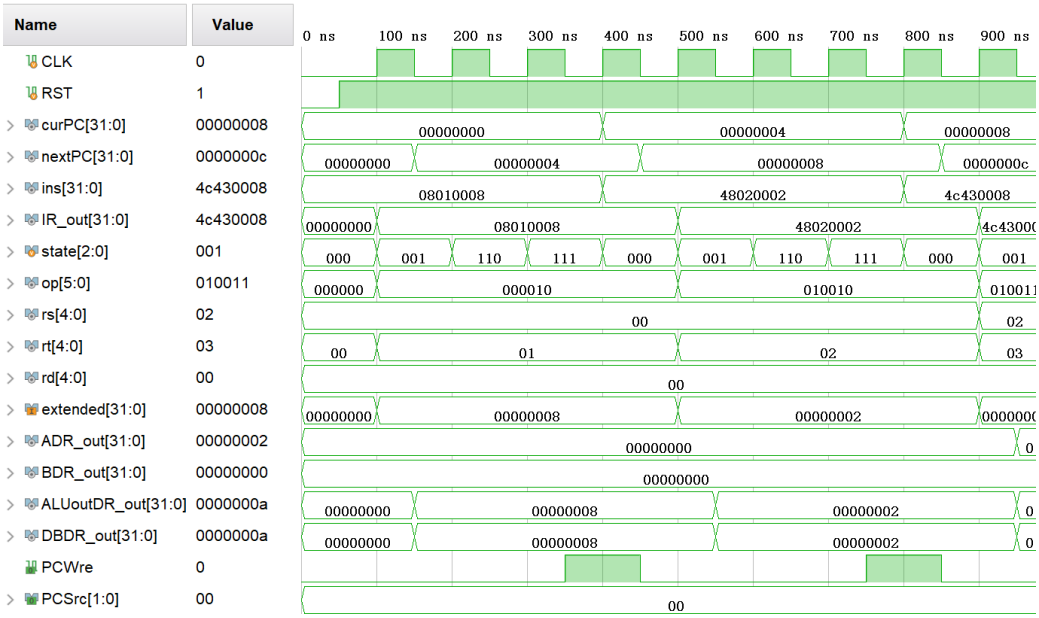

图 8 指令1~2波形图

(1) addi $1, $0, 8

1) 如图 8所示，0~400ns，共4个时钟周期，PC地址为0x00。
2) IF（000）状态，0~100ns。IR输出为当前指令机器码。
3) ID（001）状态，100~200ns。rs为0号寄存器，rt为1号寄存器，经过符号扩展的立即数为8。
4) EXE（110）状态，200~300ns。ALU将rs寄存器的值与符号扩展后的立即数做加法运算，得到结果为0+8=8。
5) WB（111）状态，300~400ns。DBDR输出为8，时钟下降沿PCWre控制信号有效，在时钟上升沿（400ns），数据8被写入1号寄存器。PCSrc为00，下一条指令将顺序执行。

(2) ori $2, $0, 2

1) 如图 8 所示，400~800ns 共 4 个时钟周期， PC 地址为 0x04。
2) IF（000）状态，400~500ns。取指令，IR 输出为当前指令机器码。
3) ID（001）状态，500~600ns。rs 为 0 号寄存器，rt 为 2 号寄存器，经过符号扩展的立即数为 2。
4) EXE（110）状态，600~700ns。ALU 将 rs 寄存器的值与零扩展后的立即数做逻辑或运算，得到结果为 0|2=2，ALU 结果被送入 DBDR 输入端。
5) WB(111)状态,700~800ns。DBDR 输出为 2,时钟下降沿 PCWre 控制信号有效，时钟上升沿（800ns），数据 2 被写入 2 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

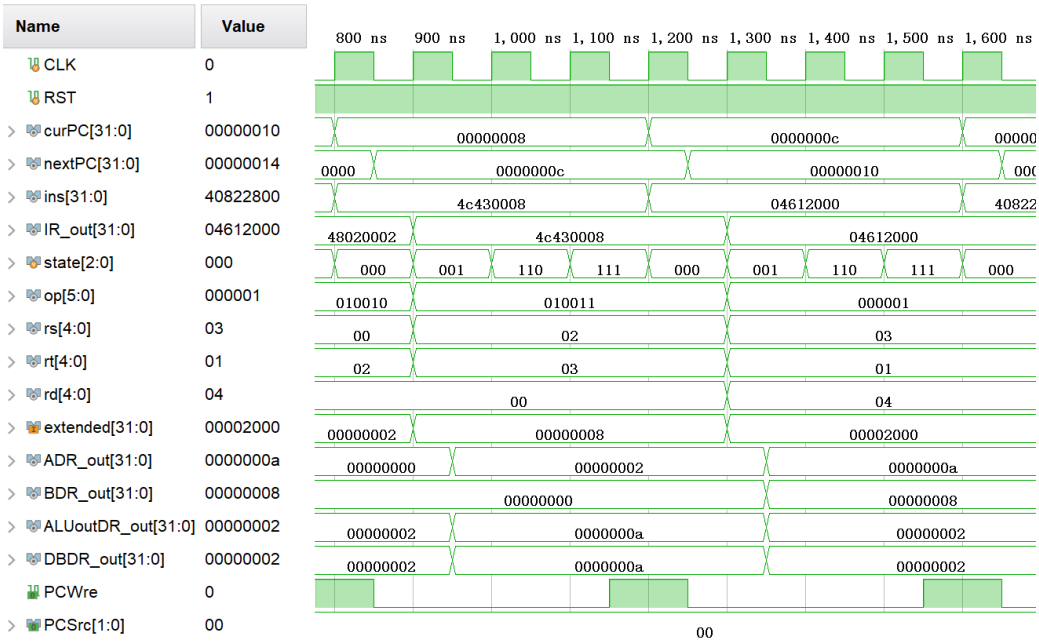

图 9 指令 3~4 波形图

(3) xori $3, $2, 8

1) 如图 9 所示，800~1200ns 共 4 个时钟周期，PC 地址为 0x08。
2) IF（000）状态，800~900ns。取指令，IR 输出为当前指令机器码。
3) ID（001）状态，900~1000ns。rs 为 2 号寄存器，rt 为 3 号寄存器，经过符号扩展的立即数为 8。
4) EXE（110）状态，1000~1100ns。ALU 将 rs 寄存器的值与零扩展后的立即数做逻辑异或运算，得到结果为 2^8=10，ALU 结果被送入 DBDR 输入端。
5) WB（111）状态，1100~1200ns。DBDR 输出为 2，时钟下降沿 PCWre 控制信号有效，在时钟上升沿（1200ns），数据 2 被写入 3 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

(4) sub $4, $3, $1

1) 如图 9 所示，1200~1600ns 共 4 个时钟周期，PC 地址为 0x0c。
2) IF（000）状态，1200~1300ns。IR 输出为当前指令机器码。
3) ID（001）状态，1300~1400ns。rs 为 3 号寄存器，rt 为 1 号寄存器，rd 为 4 号寄存器。
4) EXE（110）状态，1400~1500ns。ALU 将 rs 寄存器的值与 rt 寄存器的值做减法运算，得到结果为 10-8=2，ALU 结果被送入 DBDR 输入端。
5) WB（111）状态，1500~1600ns。DBDR 输出为 2，时钟下降沿 PCWre 控制信号有效，在时钟上升沿（1600ns），数据 2 被写入 4 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

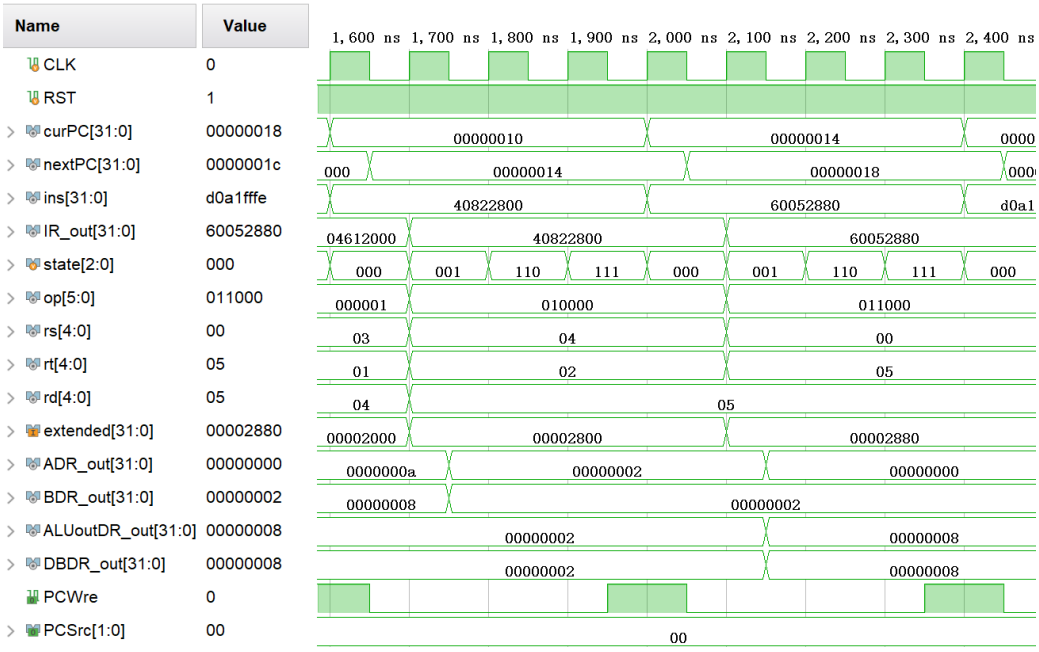

图 10 指令 5~6 波形图

(5) and $5, $4, $2

1) 如图 10 所示，1600~2000ns 共 4 个时钟周期，PC 地址为 0x10。
2) IF（000）状态，1600~1700ns。IR 输出为当前指令机器码。
3) ID（001）状态，1700~1800ns。rs 为 4 号寄存器，rt 为 2 号寄存器，rd 为 5 号寄存器。
4) EXE（110）状态，1800~1900ns。ALU 将 rs 寄存器的值与 rt 寄存器值做逻辑与运算，得到结果为 2&2=2，ALU 结果被送入 DBDR 输入端。
5) WB（111）状态，1900~2000ns。DBDR 输出为 2，时钟下降沿 PCWre 控制信号有效，在时钟上升沿（2000ns），数据 2 被写入 5 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

(6) sll $5, $5, $2

1) 如图 10 所示，2000~2400ns 共 4 个时钟周期，PC 地址为 0x14。
2) IF（000）状态，2000~2100ns。IR 输出为当前指令机器码。
3) ID（001）状态，2100~2200ns。rs 为 5 号寄存器，rt 为 5 号寄存器。
4) EXE（110）状态，2200~2300ns。ALU 将 rs 寄存器的值与 sa 寄存器值做移位运算，得到结果为 2<<2=8，ALU 结果被送入 DBDR 输入端。
5) WB（111）状态，2300~2400ns。DBDR 输出为 8，时钟下降沿 PCWre 控制信号有效，在时钟上升沿（2400ns），数据 2 被写入 5 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

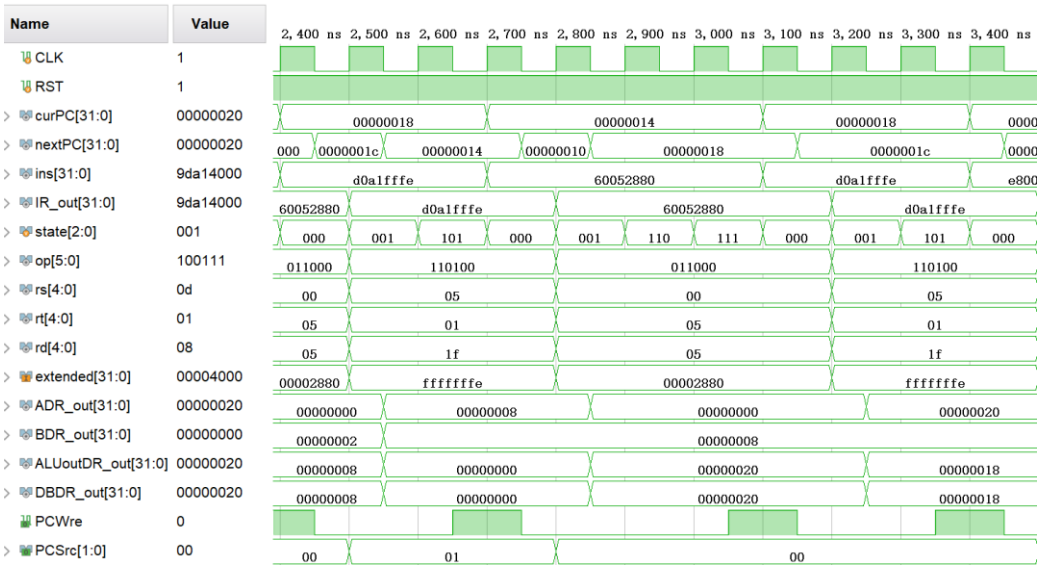

图 11 指令 7 波形图

(7) beq $5, $1, -2

1) 如图 11 所示，2400~2700ns 共 3 个时钟周期，PC 地址为 0x18。
2) IF（000）状态，2400~2500ns。IR 输出为当前指令机器码。
3) ID（001）状态，2500~2600ns。rs 为 5 号寄存器，rt 为 1 号寄存器，经符号扩展的立即数为-2。
4) EXE（110）状态，2600~2700ns。ALU 将 rs 寄存器的值与 rt 寄存器值做减法运算，得到结果为 8-8=0，zero 输出为 1，即满足跳转条件，PCSrc 为 01，PC<－PC4+(sign-extend)immediate ×4，下一条指令跳转至 0x14。
5) 跳转至 0x014 执行 sll 指令后，当再次执行跳转至 0x18 执行 beq 指令时，5 号寄存器的值与 1 号寄存器的值不再相等，PCSrc 为 00，随后下一条指令按顺序执行。

(8) jal 0x00000050

1) 如图 12 所示，3400~3600ns 共 2 个时钟周期，PC 地址为 0x1c。
2) IF（000）状态，3400~3500ns。。IR 输出为当前指令机器码。
3) ID（001）状态，3500~3600ns。Write Reg 输入 31 号寄存器，Write Data 输入 PC+4，在时钟上升沿将 PC+4 写入 31 号寄存器中。PCSrc 为 11，因此下一 PC 地址为{PC4[31:28], addr, 00}，下一条指令跳转到 0x50。

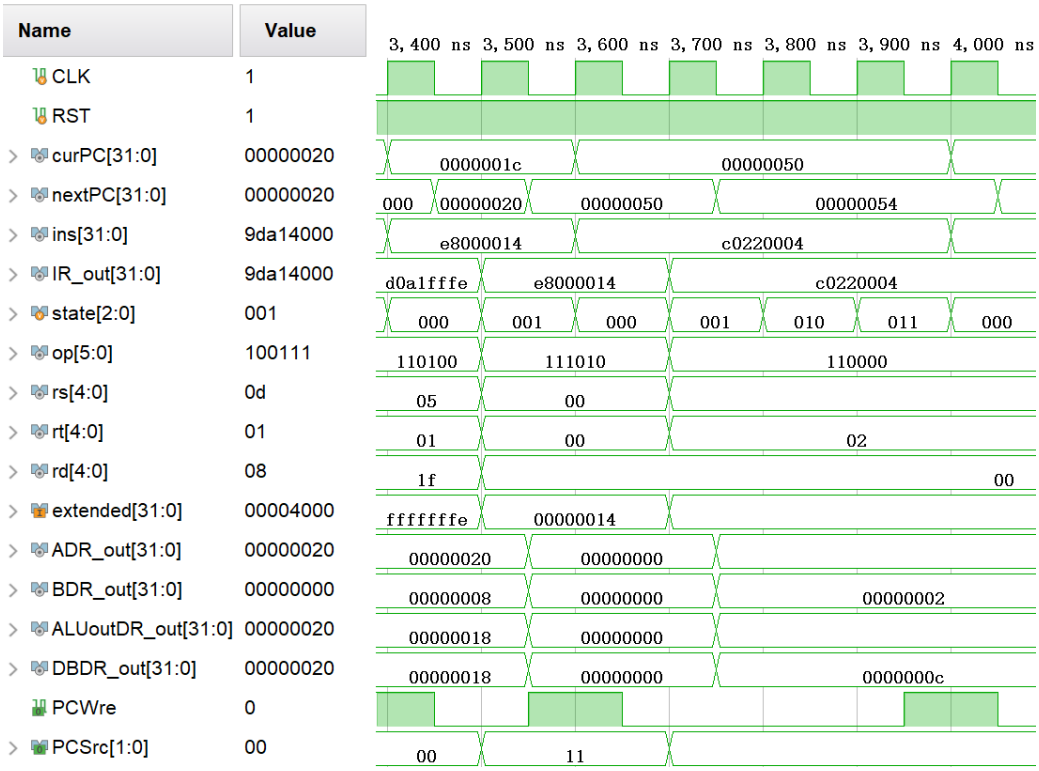

图 12 指令 8~9 波形图

(9) sw $2, 4($1)

1) 如图 12 所示,3600~4000ns 共 4 个时钟周期，PC 地址为 0x50。
2) IF（000）状态，3600~3700ns。。IR 输出为当前指令机器码。
3) ID（001）状态，3700~3800ns。rs 为 1 号寄存器，rt 为 2 号寄存器，经符号扩展的立即数为 4。
4) EXE（010）状态，3800~3900ns。ALU 将 rs 寄存器的值与经过符号扩展的立即数 4 做加法运算，得到结果为 8+4=12，ALU 结果被送入 DAddr 输入端。
5) MEM (011) 状态，3900~4000ns。ALUoutDR 的输出改变为 12，并作为地址提供给数据存储器 DAddr 输入端口。mWR 信号有效，在时钟下降沿将 2 号寄存器的值写入存储器中地址为 12 的位置。PCSrc 为 00，下一条指令将顺序执行。

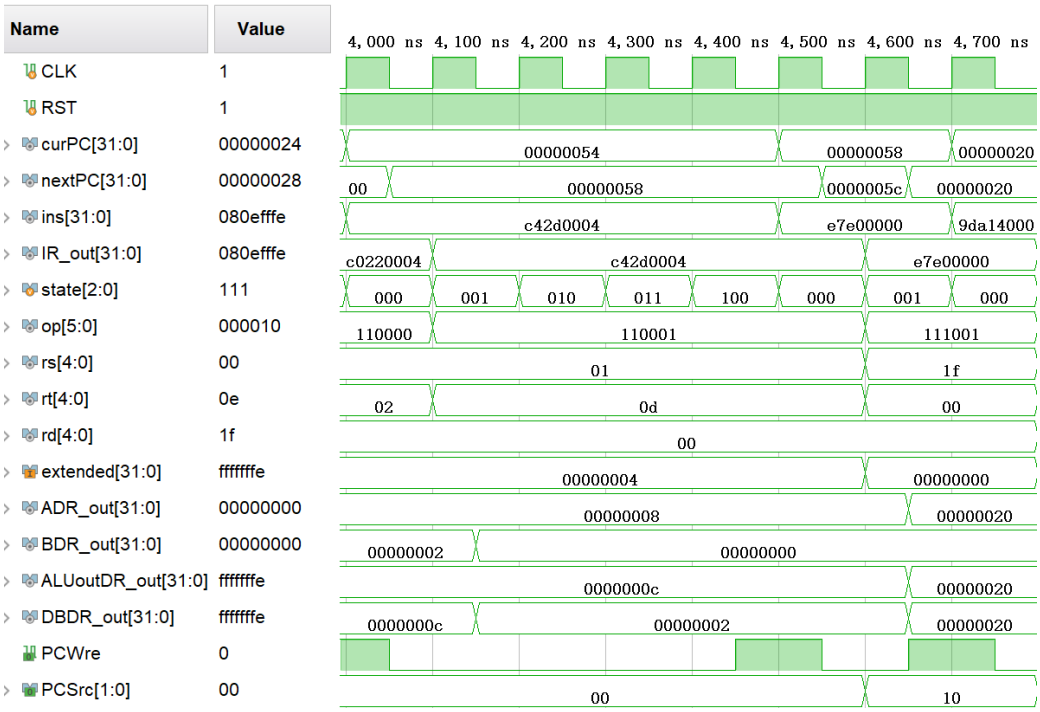

图 13 指令 10~11 波形图

(10) lw $13, 4($1)

1) 如图 13 所示，4000~4500ns 共 5 个时钟周期，PC 地址为 054。
2) IF（000）状态，4000~4100ns。IR 输出为当前指令机器码。
3) ID（001）状态，4100~4200ns。rs 为 1 号寄存器，rt 为 13 号寄存器，经符号扩展的立即数为 4。
4) EXE（010）状态，4200~4300ns。ALU 将 rs 寄存器的值与经过符号扩展的立即数 4 做加法运算，得到结果为 8+4=12，ALU 结果被送入 DAddr 输入端。
5) MEM(011)状态，4300~4400ns。ALUoutDR 的输出改变为 12，并作为地址提供给数据存储器 DAddr 输入端口。mRD 信号有效，在时钟下降沿将存储器中地址为 12 的位置的数据读出至输出端 DataOut，DataOut 数据被送入 DBDR 输入端。
6) WB(100)，4400~4500ns。DBDR 输出为 2，在时钟上升沿 DBDR 输出被写入 13 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

(11) jr $31

1) 如图 13 所示，4500~4700ns 共 2 个时钟周期，PC 地址为 0x58。
2) IF（000）状态，4500~4600ns。IR 输出为当前指令机器码。
3) ID（001）状态，4600~4700ns。rs 为 31 号寄存器，31 号寄存器保存的地址为 0x20。PCSrc 为 10，因此下一指令地址为 PC<－rs，即下一条指令跳转到 0x20。

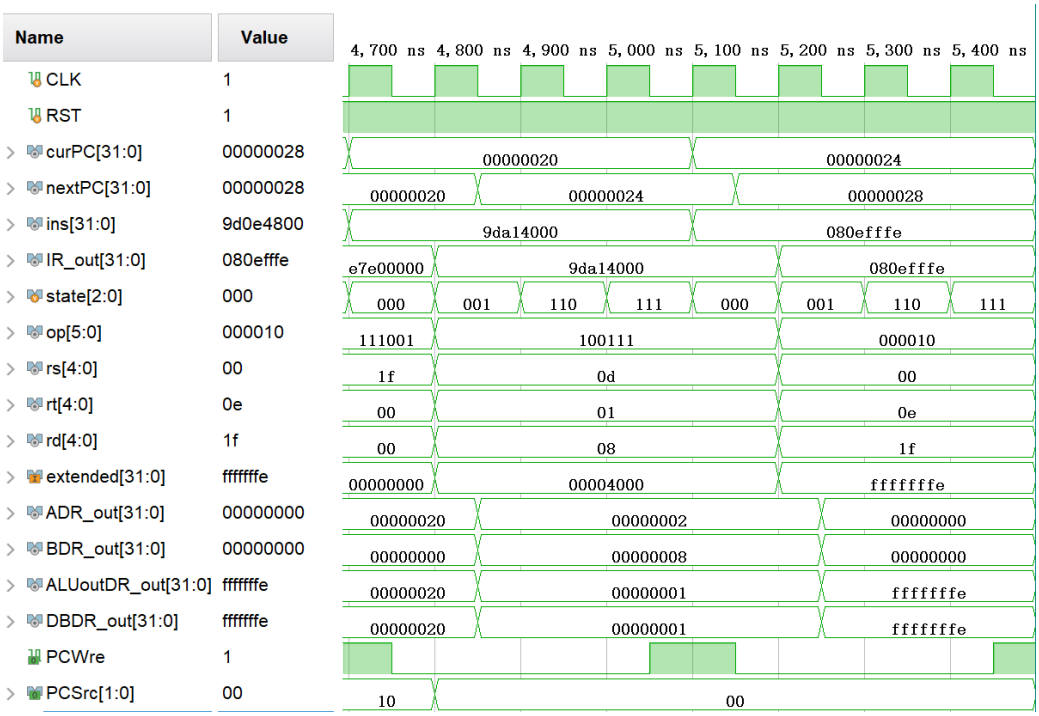

图 14 指令 12~13 波形图

(12) slt $8, $13, $1

1) 如图 14 所示，4700~5100ns 共 4 个时钟周期，PC 地址为 0x20。
2) IF（000）状态，4700~4800ns。IR 输出为当前指令机器码。
3) ID（001）状态，4800~4900ns。rs 为 13 号寄存器，rt 为 1 号寄存器，rd 为 8 号寄存器。
4) EXE（110）状态，4900~5000ns。ALU 将 rs 寄存器的值与 rt 寄存器的值做带符号比较运算，得到结果为 1<-2=0，ALU 结果被送入 DBDR 输入端。
5) WB(111)，5000~5100ns。DBDR 输出为 0，在时钟上升沿 DBDR 输出被写入 8 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

(13) addi $14, $0, -2

1) 如图 14 所示，5100~5500ns 共 4 个时钟周期，PC 地址为 0x24。
2) IF（000）状态，5100~5200ns。IR 输出为当前指令机器码。
3) ID（001）状态，5200~5300ns。rs 为 0 号寄存器，rt 为 14 号寄存器，经过符号扩展的立即数为-2。
4) EXE（110）状态，5300~5400ns。ALU 将 rs 寄存器的值与符号扩展的立即数做加法运算，得到结果为 0-2=-2，ALU 结果被送入 DBDR 输入端。
5) WB(111)，5400~5500ns。DBDR 输出为-2，在时钟上升沿 DBDR 输出被写入 14 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

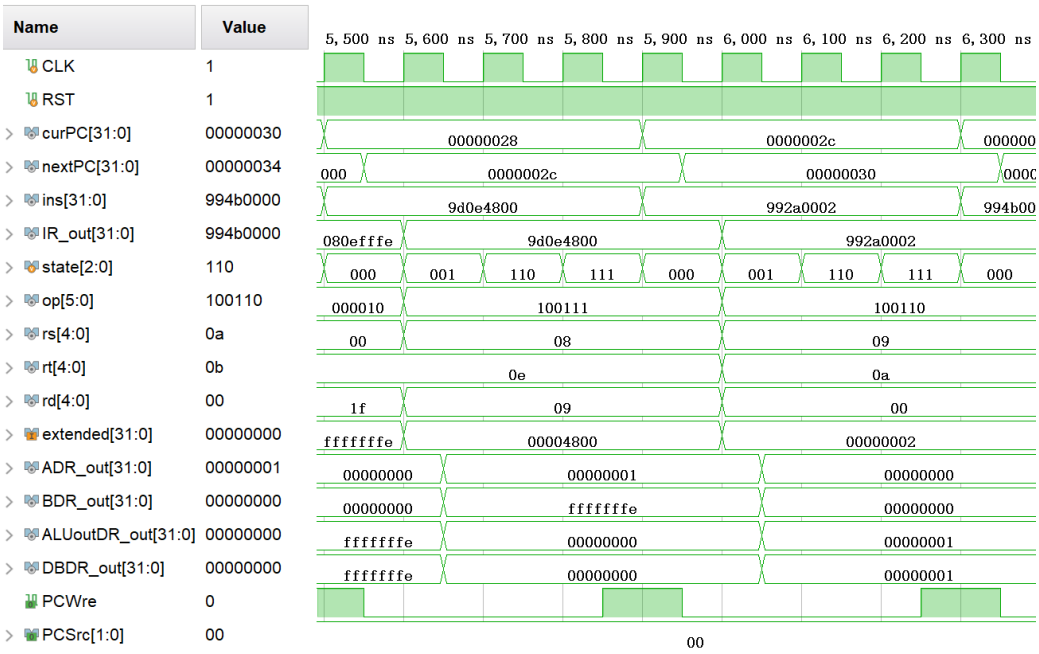

图 15 指令 14~15 波形图

(14) slt $9, $8, $14

1) 如图 15 所示，5500~5900ns 共 4 个时钟周期，PC 地址为 0x28。
2) IF（000）状态，5500~5600ns。IR 输出为当前指令机器码。
3) ID（001）状态，5600~5700ns。rs 为 8 号寄存器，rt 为 14 号寄存器，rd 为 9 号寄存器。
4) EXE（110）状态，5700~5800ns。ALU 将 rs 寄存器的值与 rt 寄存器的值做带符号比较运算，得到结果为 1<-2=0，ALU 结果被送入 DBDR 输入端。
5) WB(111)，5800~5900ns。DBDR 输出为 0，在时钟上升沿 DBDR 输出被写入 9 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

(15) slti $10, $9, 2

1) 如图 15 所示，5900~6300ns 共 4 个时钟周期，PC 地址为 0x2c。
2) IF（000）状态，5900~6000ns。IR 输出为当前指令机器码。
3) ID（001）状态，6000~6100ns。rs 为 9 号寄存器，rt 为 10 号寄存器，经符号扩展的立即数为 2。
4) EXE（110）状态，6100~6200ns。ALU 将 rs 寄存器的值与经符号扩展的立即数做带符号比较运算，得到结果为 0<2=1，ALU 结果被送入 DBDR 输入端。
5) WB(111)，6200~6300ns。DBDR 输出为 1，在时钟上升沿 DBDR 输出被写入 10 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

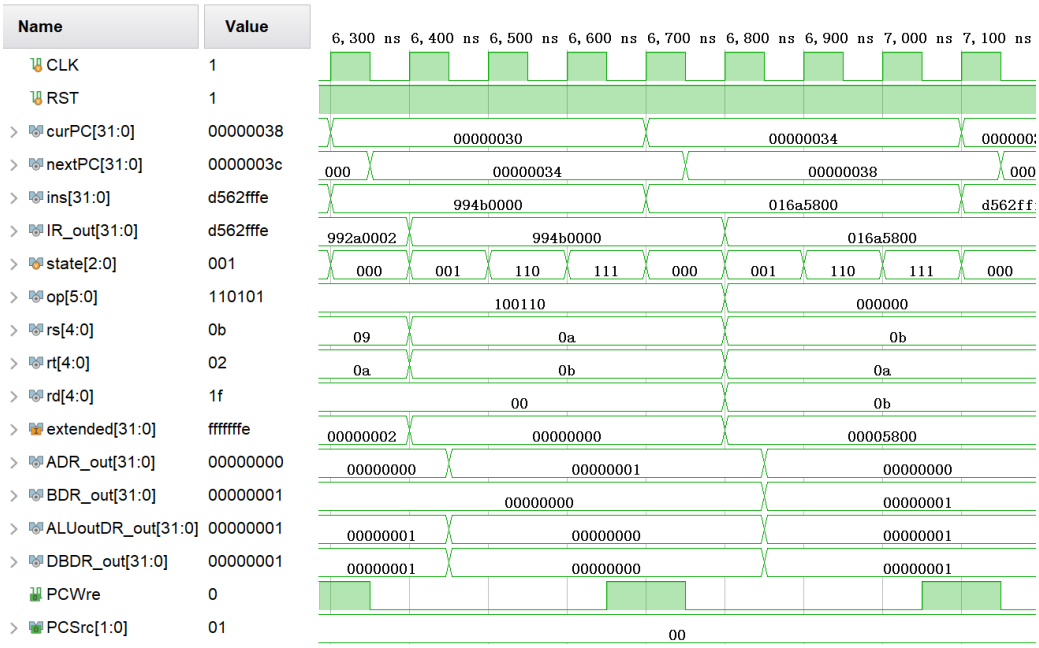

图 16 指令 16~17 波形图

(16) slti $11, $10, 0

1) 如图 16 所示，6300~6700ns 共 4 个时钟周期，PC 地址为 0x30。
2) IF（000）状态，6300~6400ns。IR 输出为当前指令机器码。
3) ID（001）状态，6400~6500ns。rs 为 11 号寄存器，rt 为 10 号寄存器，经符号扩展的立即数为 0。
4) EXE（110）状态，6500~6600ns。ALU 将 rs 寄存器的值与经符号扩展的立即数做带符号比较运算，得到结果为 1<0=0，ALU 结果被送入 DBDR 输入端。
5) WB（111），6600~6700ns。DBDR 输出为 0，在时钟上升沿 DBDR 输出被写入 11 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

(17) add $11, $11, $10

1) 如图 16 所示，6700~7100ns 共 4 个时钟周期，当 PC 地址为 0x34。
2) IF（000）状态，6700~6800ns。IR 输出为当前指令机器码。
3) ID（001）状态，6800~6900ns。rs 为 11 号寄存器，rt 为 10 号寄存器，rd 寄存器为 11 号寄存器。
4) EXE（110）状态，6900~7000ns。ALU 将 rs 寄存器的值与 rt 寄存器的值做加法运算，得到结果为 0+1=0，ALU 结果被送入 DBDR 输入端。
5) WB（111），7000~7100ns。DBDR 输出为 1，在时钟上升沿 DBDR 输出被写入 11 号寄存器。PCSrc 为 00，下一条指令将顺序执行

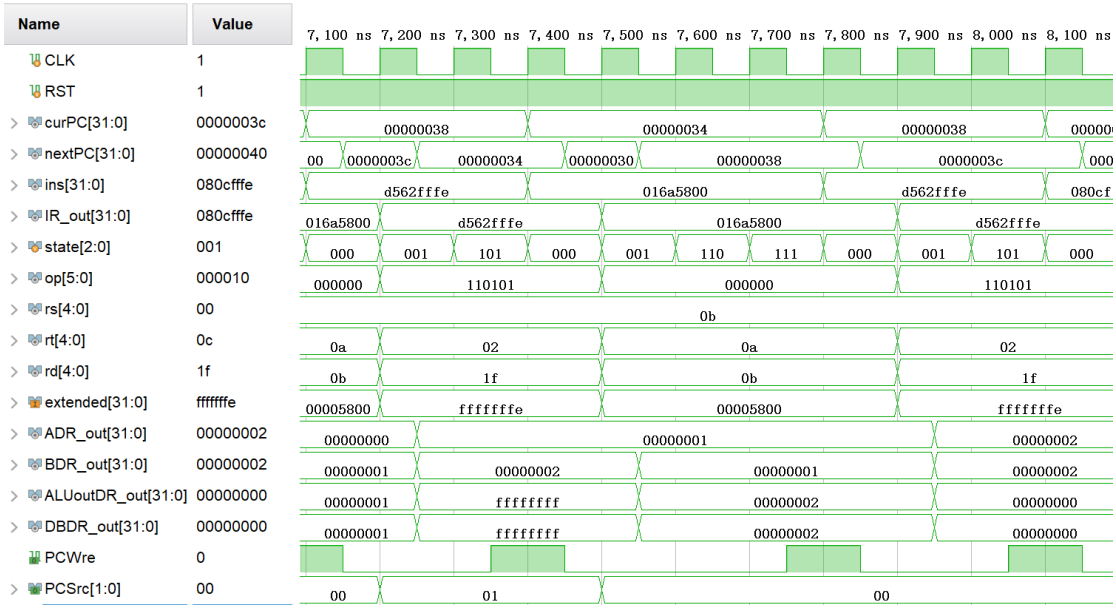

图 17 指令 18 波形图

(18) bne $11, $2, -2

1) 如图 17 所示，7100~7400ns 共 3 个时钟周期，PC 地址为 0x38。
2) IF（000）状态，7100~7200ns。取指令，IR 寄存器输出为当前执行的指令机器码。
3) ID（001）状态，7200~7300ns。rs 为 11 号寄存器，rt 为 2 号寄存器，经符号扩展的立即数为-2。
4) EXE（110）状态，7300~7400ns。ALU 将 rs 寄存器的值与 rt 寄存器值做减法运算，得到结果为 2-1=1，zero 输出为 0，即满足跳转条件，PCSrc 为 01，PC<－PC4+(sign-extend)immediate ×4，下一条指令跳转至 0x34。
5) 跳转至 0x34 执行 add 指令后，当再次执行跳转至 0x38 执行 bne 指令时，2 号寄存器的值与 11 号寄存器的值相等，PCSrc 为 00，随后下一条指令按顺序执行。

(19) addi $12, $0, -2

1) 如图 18 所示，8100~8500ns 共 4 个时钟周期，PC 地址为 0x3c。
2) IF（000）状态，8100~8200ns。IR 输出为当前指令机器码。
3) ID（001）状态，8200~8300ns。rs 为 0 号寄存器，rt 为 12 号寄存器，经符号扩展的立即数为-2。
4) EXE（110）状态，8300~8400ns。ALU 将 rs 寄存器的值与经符号扩展的立即数做求和运算，得到结果为 0-2=-2。
5) WB(111)，8400~8500ns。DBDR 输出为-2，在时钟上升沿 DBDR 输出被写入 12 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

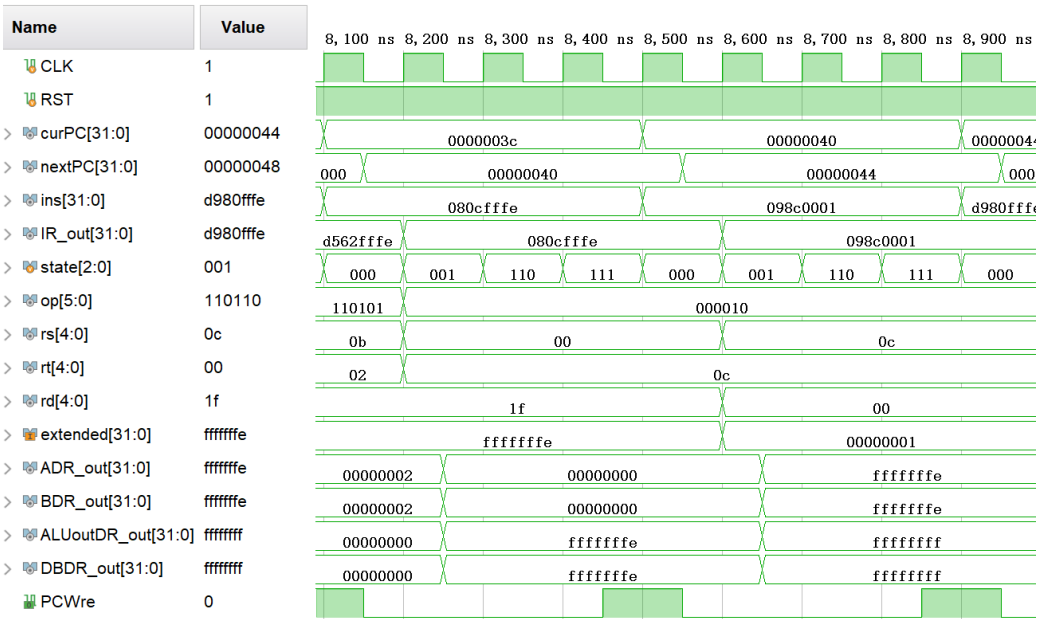

图 18 指令 19~20 波形图

(20) addi $12, $0, -2

1) 如图 18 所示，8100~8500ns 共 4 个时钟周期，PC 地址为 0x3c。
2) IF（000）状态，8100~8200ns。IR 输出为当前指令机器码。
3) ID（001）状态，8200~8300ns。rs 为 0 号寄存器，rt 为 12 号寄存器，经符号扩展的立即数为-2。
4) EXE（110）状态，8300~8400ns。ALU 将 rs 寄存器的值与经符号扩展的立即数做求和运算，得到结果为 0-2=-2，ALU 结果被送入 DBDR 输入端。
5) WB(111)，8400~8500ns。DBDR 输出为-2，在时钟上升沿 DBDR 输出被写入 12 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

(21) addi $12, $12, 1

1) 如图 18 所示，8500~8900ns 共 4 个时钟周期，PC 地址为 0x40。
2) IF（000）状态，8500~8600ns。IR 输出为当前指令机器码。
3) ID（001）状态，8600~8700ns。rs 为 12 号寄存器，rt 为 12 号寄存器，经符号扩展的立即数为 1。
4) EXE（110）状态，8700~8800ns。ALU 将 rs 寄存器的值与经符号扩展的立即数做求和运算，得到结果为-2+1=-1，ALU 结果被送入 DBDR 输入端。
5) WB(111)，8800~8900ns。DBDR 输出为-1，在时钟上升沿 DBDR 输出被写入 12 号寄存器。PCSrc 为 00，下一条指令将顺序执行。

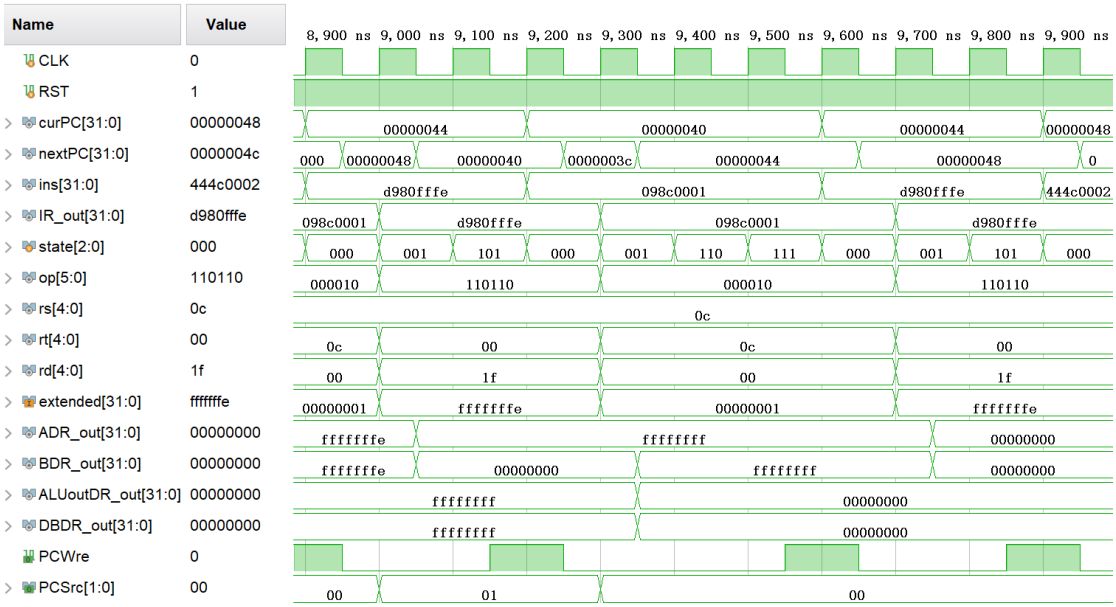

图 19 指令 21 波形图

(22) bltz $12, -2

1) 如图 19 所示，8900~9200ns 共 3 个时钟周期，PC 地址为 0x44。
2) IF（000）状态，8900~9000ns。IR 输出为当前指令机器码。
3) ID（001）状态，9000~9100ns。rs 为 12 号寄存器，rt 可以看作 0 号寄存器，经符号扩展的立即数为-2。
4) EXE（110）状态，9100~9200ns。ALU 将 rs 寄存器的值与 rt 寄存器值做加法运算，得到结果为-1+0=-1，sign 输出为 1，即满足跳转条件，PCSrc 为 01，PC<－PC4+(sign-extend)immediate ×4，下一条指令跳转至 0x40。
5) 跳转至 0x40 执行 addi 指令后，当再次执行跳转至 0x40 执行 bltz 指令时，12 号寄存器的值等于 0，PCSrc 为 00，随后下一条指令按顺序执行。

(23) andi $12, $2, 2

1) 如图 20 所示，9900~10300ns 共 4 个时钟周期，PC 地址为 0x48。
2) IF（000）状态，9900~10000ns。IR 输出为当前指令机器码。
3) ID（001）状态，10000~10100ns。rs 为 2 号寄存器，rt 为 12 号寄存器，经符号扩展的立即数为 2。
4) EXE（110）状态，10100~10200ns。ALU 将 rs 寄存器的值与经符号扩展的立即数做与运算，得到结果为 2^2=2，ALU 结果被送入 DBDR 输入端。
5) WB（111），10200~10300ns。DBDR 输出为 2，在时钟上升沿 DBDR 输出被写入 12 号寄存器。PCSrc 为 00，下一指令顺序执行。

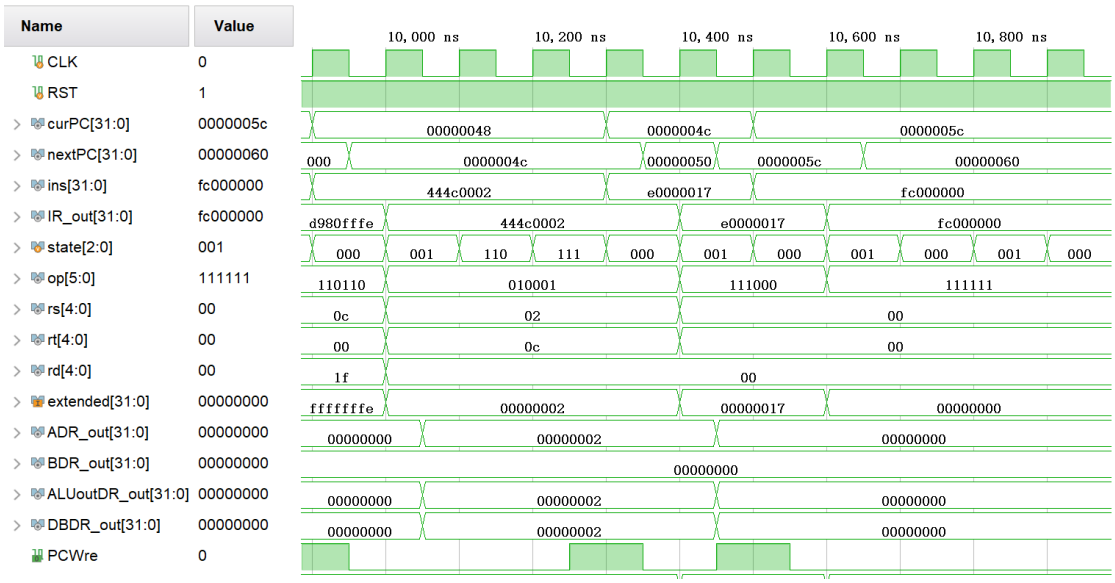

图 20 指令 22~24 波形图

(24) j 0x0000005c

1) 如图 20 所示，10300~10500ns 共 2 个时钟周期，PC 地址为 0x4c。
2) IF（000）状态，10300~10400ns。IR 输出为当前指令机器码。
3) ID（001）状态，10400~10500ns。PCSrc 为 11，因此下一条指令地址为{PC4[31:28], addr, 00}，即下一条指令跳转到 0x5c。

(25) halt

1) 如图 20 所示，10500ns 以后为 halt 指令。当前 PC 地址为 0x5c。PCWre 控制信号永远保持无效，因此 PC 不再自增，系统保持停机状态。

六. 实验心得

多周期MIPS CPU是在单周期MIPS CPU的基础上发展而来的，因此他们在实现原理上与有非常多的相似之处，如：instructionMemory、registerFile和ALU等底层模块设计完全相同，区别在于数据通路上新增的寄存器和controlUnit的逻辑由组合逻辑变为时序逻辑。新增的寄存器IR、ADR、BDR、DBDR和ALUoutDR的原理可以看作是D触发器。多周期CPU实验主要的难点和重点在于理解一条指令分多个周期、多个步骤完成，在不同时钟周期需要清楚的知道各个底层模块的具体状态。

在实验中，主要遇到以下两个比较大的bug:

1) controlUnit中新增时钟信号用于内部的指令根据指令状态图（图 3）的状态进行转移。状态转移需要根据当前指令的opCode和当前cpu所处的状态生成各底层模块的控制信号，并推断出该指令的下一状态。在编写状态转移规则时，需要注意不同指令对应的状态码，避免因为细节问题导致cpu运行状态错乱。
2) 寄存器、存储器读出或写入操作需要注意时机（时钟上升沿或时钟下降沿），在实

验一开始时写入和读出均在时钟下降沿发生，造成了数据混乱。

另外，本次实验还编写了简单的MIPS代码编码器，因为编码器涉及文件的读入写出和字符串相关的处理，因此使用python进行编写。

经过本次多周期MIPS CPU设计实验，既复习了理论课上学习的相关知识，也完成了能运作CPU的实现，巩固了在完成单周期CPU实验时对其原理、数据通路和指令执行五阶段具体实现的理解。与此同时，再次提醒自己要注重细节，注意细节的具体实现，否则容易导致许多莫名其妙的bug发生。

附录：编码器代码（compiler.py）

```
import re
import sys

def regToNum(reg):
    regID = int(reg.lstrip(' $'))
    result = bin(regID)[2:].rjust(5, '0')
    return result

def signExtend(immediate):
    if immediate >= 0:
        immediate = bin(immediate)[2:].rjust(16, '0')
    else:
        immediate = bin(immediate + 2 ** 16)[2:].rjust(16, '1')
    return immediate

def typeR(op, operand):
    rs = regToNum(operand[1])
    rt = regToNum(operand[2])
    rd = regToNum(operand[0])
    return op + rs + rt + rd + 11 * '0'

def typeI(op, operand):
    rs = regToNum(operand[1])
    rt = regToNum(operand[0])
    immediate = signExtend(int(operand[2]))
    return op + rs + rt + immediate

def typeJ(op, operand):
    addr = bin(int(operand[0], base=16))[2:-2].rjust(26, '0')
    return op + addr
```

```
def asmCompile(ins, operand):
    if ins == 'add':
        result = typeR('000000', operand)
    elif ins == 'sub':
        result = typeR('000001', operand)
    elif ins == 'addi':
        result = typeI('000010', operand)
    elif ins == 'and':
        result = typeR('010000', operand)
    elif ins == 'andi':
        result = typeI('010001', operand)
    elif ins == 'ori':
        result = typeI('010010', operand)
    elif ins == 'xori':
        result = typeI('010011', operand)
    elif ins == 'sll':
        sa = bin(int(operand[2]))[2:].rjust(5, '0')
        op = '011000'
        result = (
            op+'00000'+regToNum(operand[1]) + regToNum(operand[0]) + sa).ljust(32, '0')
    elif ins == 'slti':
        result = typeI('100110', operand)
    elif ins == 'slt':
        result = typeR('100111', operand)
    elif ins == 'sw' or ins == 'lw':
        tempRegex = re.compile('([0-9])+\((.+)\)')
        rt = regToNum(operand[0])
        rs = regToNum(tempRegex.search(operand[1]).group(2))
        immediate = signExtend(
            (int(bin(int(tempRegex.search(operand[1]).group(1)))[2:], base=2)))
        if ins == 'sw':
            op = '110000'
        else:
            op = '110001'
        result = op + rs + rt + immediate
    elif ins == 'beq':
        operand[0], operand[1] = operand[1], operand[0]
        result = typeI('110100', operand)
    elif ins == 'bne':
        operand[0], operand[1] = operand[1], operand[0]
        result = typeI('110101', operand)
    elif ins == 'bltz':
        operand.insert(1, '00000')
```

```
        operand[0], operand[1] = operand[1], operand[0]
        result = typeI('110110', operand)
    elif ins == 'j':
        result = typeJ('111000', operand)
    elif ins == 'jr':
        result = '111001' + (bin(int((operand[0]).lstrip(' $')))
             [2:].rjust(5, '0')).ljust(26, '0')
    elif ins == 'jal':
        result = typeJ('111010', operand)
    elif ins == 'halt':
        result = '111111' + '0' * 26
    else:
        result = ''
    return result

if __name__ == '__main__':
    if len(sys.argv) != 3:
        print('[Failed]command format: python MIPS_compiler.py <input> <output>')
        exit()
    try:
        asmFile = open(sys.argv[1], 'r', encoding='UTF-8')
    except FileNotFoundError:
        print('the given file "{0}" can\'t be found'.format(sys.argv[1]))
    else:
        outputFile = open(sys.argv[2], 'w')
    regex = re.compile('([a-zA-Z]+) +(.+)')
    output = []
    for asmCode in asmFile:
        asmCode = asmCode.strip()
        match = regex.search(asmCode)
        if match is not None:
            instruction = match.group(1)
            operand = match.group(2).split(',')
        else:
            instruction = asmCode
            operand = []
        machineCode = asmCompile(instruction, operand)
        output.append(machineCode)
    outputFile.write('\n'.join(output))
    print('Complete the MIPS code assembly.')
```