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←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←rs - rt。

（3）addi rt , rs ,**immediate**

000010	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。

==> 逻辑运算指令

（4）andi 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“与”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）ori rt , rs ,**immediate**

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

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

（7）or　rd , rs , rt

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

功能：rd←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<－rt<<(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：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。

（11) 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

（12）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”。

（13）bne　rs,rt,**immediate**

110001	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：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←pc + 4 + (sign-extend)**immediate** <<2　else pc ←pc + 4。

==>跳转指令

（15）j　addr

111000	addr[27:2]

功能：pc <－{(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	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。

(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。

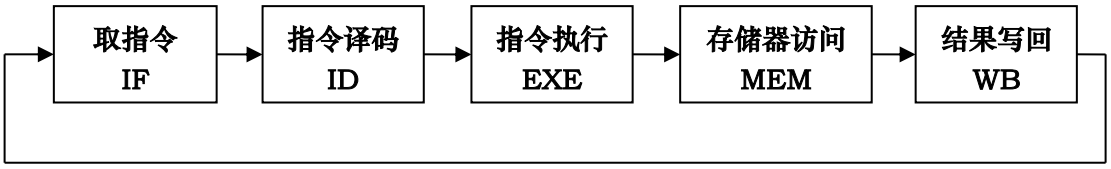

图 1 单周期 CPU 指令处理过程

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）；

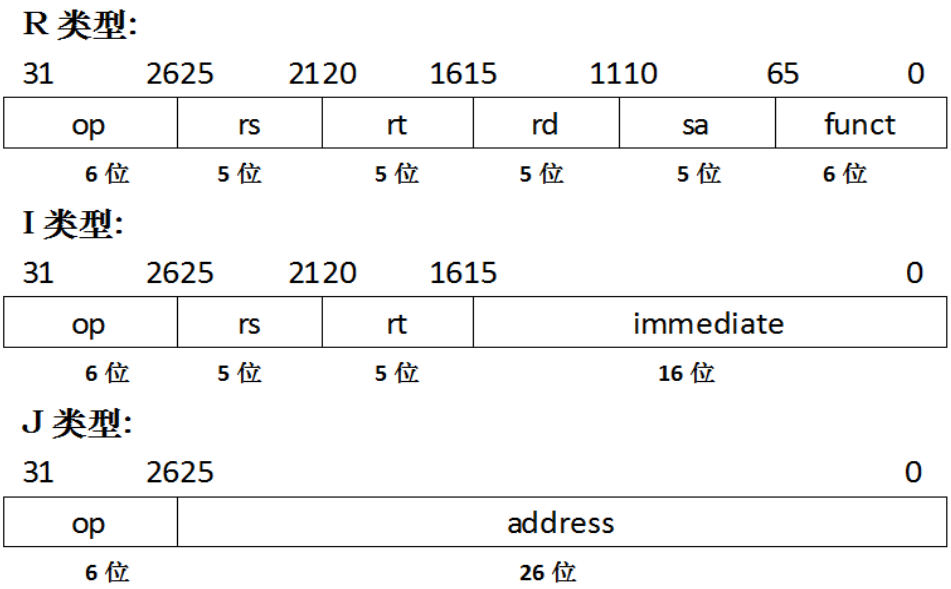

图 2 MIPS 指令的三种格式

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。

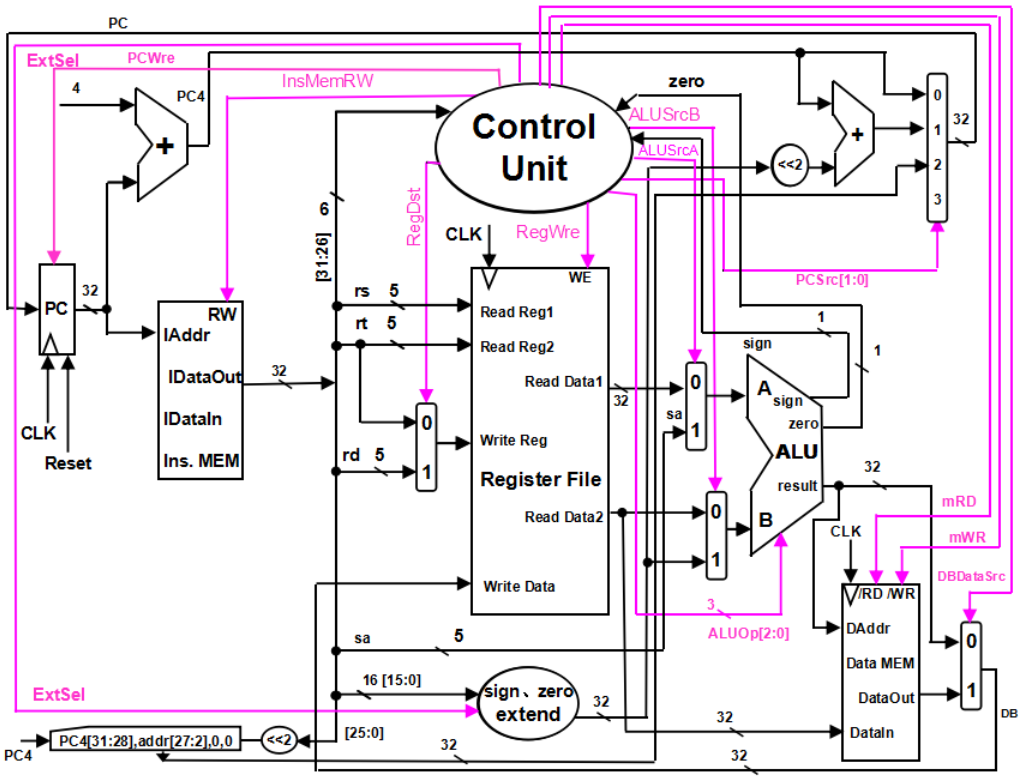

图 3 单周期 CPU 数据通路和控制线路图

图 3 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

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<－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<<2**，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	

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

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

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，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，设计时注意用模块化的思想方法设计，而关于 ALU 设计、存储器设计、寄存器组设计等等，也是需要认真考虑的问题。

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

A. CPU设计

单周期CPU在每个时钟周期完成一条指令的执行，每条指令执行过程可被分为五个阶段：IF、ID、EXE、MEM、WB。每个阶段需要实现的功能由各功能模块（module）完成，表 3列出各执行阶段与其实现模块的对照关系。除此以外，还需要一个在特定执行阶段对各个模块发出正确控制信号的控制模块（ControlUnit）和一个对立即数扩展的模块（ImmediateExtend）。

表 3 指令执行各阶段的功能模块

阶段	功能模块
IF 取指令	PC, InstructionMemory
ID 指令译码	RegisterFile
EXE 执行	ALU
MEM 访存	DataMemory
WB 结果写回	RegisterFile
实现立即数扩展	ImmediateExtend
实现对各模块控制	ControlUnit

依照图 3 单周期CPU数据通路和控制线路图和表 3 指令执行各阶段的功能模块，将CPU划分为九个功能模块(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 4所示。

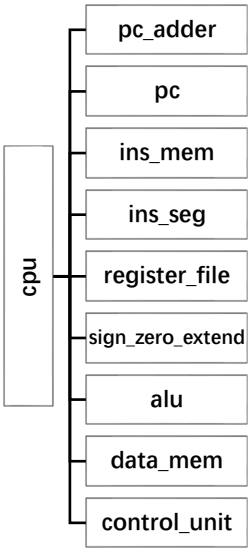

图 4 单周期 CPU 模块划分图

a) **底层模块设计**

i. **pc_adder**

模块功能：根据控制信号PCSrc，计算下一个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 <= {pc[31:28],addr,2'b00};
            2'b11: nextPC <= nextPC;
        endcase
    end
end
```

ii. **pc**

模块功能

- 根据控制信号PCWre，判断pc是否改变；
- 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，触发敏感信号时改变pc。

主要代码

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

iii. **ins_mem**

模块功能：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

- CPU初始化时，将指令读入并预存至内含128个字节的ROM中；
- 使用pc作为敏感信号，当pc发生改变时，从ROM中读取地址对应指令，并将其输出。

主要代码

```
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
```

iv. **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
```

v. **register_file**

模块功能：寄存器存储器。按控制信号RegWre对寄存器进行读或写操作。

实现思路

- CPU初始化时，初始化内含32个32位寄存器的寄存器组regFile；
- 读操作（RegWre=0）时，按寄存器编号（rs、rt）输出寄存器的值；
- 写操作（RegWre=1）时，根据控制信号RegDst选择被写入寄存器的编号（rt或rd），时钟上升沿时将数据（Write Data）写入对应寄存器。

主要代码

```
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
assign ReadData1 = regFile[ReadReg1];
assign ReadData2 = regFile[ReadReg2];

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

vi. 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。

主要代码

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

vii. alu

模块功能：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：

- 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；
- 根据表 2 ALU运算功能表完成设计。

主要代码

```
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
```

viii. 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

- CPU初始化时，初始化内含128个8位寄存器的数据存储器ram；
- 按照控制信号进行读操作或写操作；
- 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
```

ix. control_unit

模块功能：控制单元。在特定执行阶段对各个模块发出正确控制信号。

实现思路

- 根据表 1综合分析得出表 4；
- 根据操作码（op）、状态量（zero、sign）和表 4，输出对其他底层模块的控制信号。

表 4 控制信号量与控制指令对照表

指令	控制信号量											
	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	InsMemRW	mRD	mWR	RegDst	ExtSel	PCSrc	ALUOp
addi	1	0	1	0	1	1	0	0	0	1	00	000
ori	1	0	1	0	1	1	0	0	0	1	00	011
add	1	0	0	0	1	1	0	0	1	0	00	000
sub	1	0	0	0	1	1	0	0	1	1	00	000
and	1	0	0	0	1	1	0	0	1	0	00	100
or	1	0	0	0	1	1	0	0	1	0	00	011
sll	1	1	0	0	1	1	0	0	1	0	00	010
bne	1	0	0	0	0	1	0	0	x	1	01/ 00(zero)	001
slti	1	0	1	0	1	1	0	0	0	1	00	101
beq	1	0	0	0	0	1	0	0	x	1	00/ 01(zero)	001
sw	1	0	1	0	0	1	0	1	x	1	00	000
lw	1	0	1	1	1	1	1	0	0	1	00	000
bltz	1	0	0	0	0	1	0	0	0	1	00/ 01(sign)	000
j	1	x	x	0	0	1	0	0	x	0	10	000
halt	0	x	x	0	0	1	0	0	x	0	11	000

主要代码：为节约篇幅，此处仅给出addi指令相关代码（完整代码见control_unit.v），其他指令实现代码结构类似，仅各个控制信号值不同。

```
always@(op or zero or sign) begin
    case(op)
        6'b000010: begin    //addi
            PCWre <= 1;
            {ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, mRD,
mWR, RegDst, ExtSel} <= 9'b010110001;
            PCSrc[1:0] <= 2'b00;
            ALUOp[2:0] <= 3'b000;
        end
        // ... other instruction case
    endcase
end
```

b) 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 3 单周期CPU数据通路和控制线路图连接。

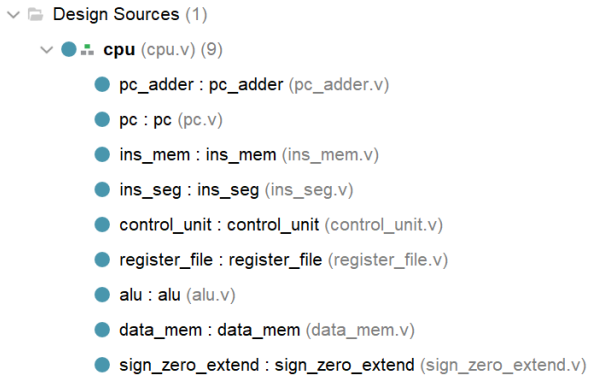

图 5 单周期 CPU 文件结构

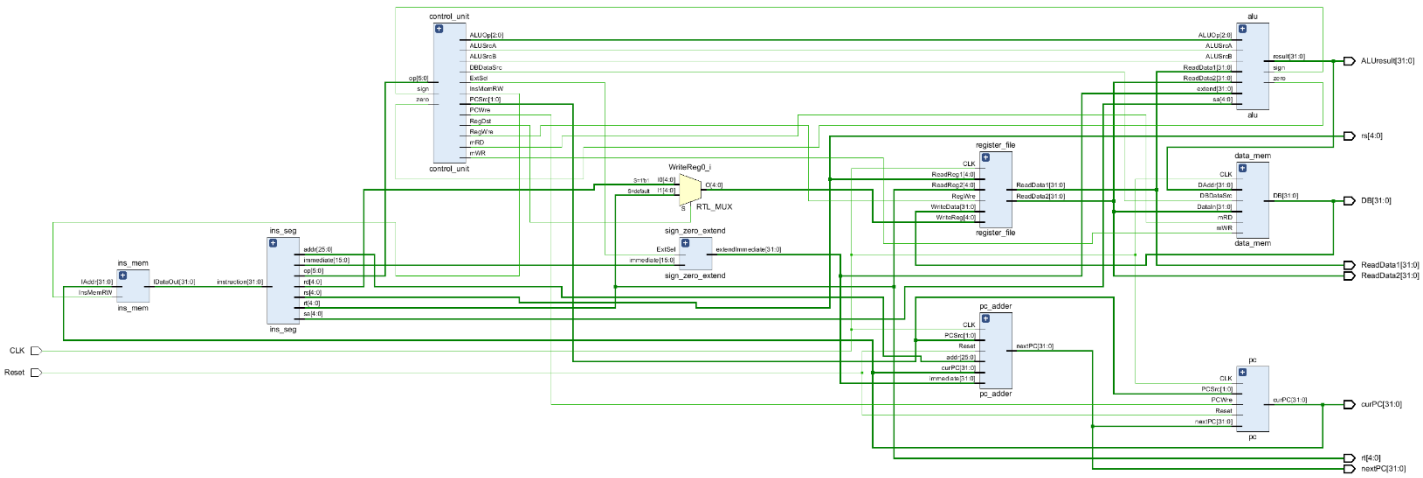

图 6 RTL Schematic

B. 仿真验证CPU正确性

a) CPU程序代码测试

根据所给文档提供的指令（表 5，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 5 单周期 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	add $3,$2,$1	000000	00010	00001	00011 00000 000000	=	00411800
0x0000000C	sub $5,$3,$2	000001	00011	00010	00101 00000 000000	=	04622800
0x00000010	and $4,$5,$2	010001	00101	00010	00100 00000 000000	=	44A22000
0x00000014	or $8,$4,$2	010011	00100	00010	01000 00000 000000	=	4C824000
0x00000018	sll $8,$8,1	011000	00000	01000	01000 00001 000000	=	60084040
0x0000001C	bne $8,$1,-2 (≠,转 18)	110001	01000	00001	1111 1111 1111 1110	≠	C501FFFE
0x00000018	sll $8,$8,1	011000	00000	01000	01000 00001 000000	=	60084040
0x0000001C	bne $8,$1,-2	110001	01000	00001	1111 1111 1111 1110	=	C501FFFE
0x00000020	slti $6,$2,4	011100	00010	00110	0000 0000 0000 0100	=	70460004
0x00000024	slti $7,$6,0	011100	00110	00111	0000 0000 0000 0000	=	70C70000
0x00000028	addi $7,$7,8	000010	00111	00111	0000 0000 0000 1000	=	08E70008
0x0000002C	beq $7,$1,-2 (=,转 28)	110000	00111	00001	1111 1111 1111 1110	=	C0E1FFFE
0x00000028	addi $7,$7,8	000010	00111	00111	0000 0000 0000 1000	=	08E70008
0x0000002C	beq $7,$1,-2	110000	00111	00001	1111 1111 1111 1110	≠	C0E1FFFE
0x00000030	sw $2,4($1)	100110	00001	00010	0000 0000 0000 1000	=	98220008
0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 1000	=	9C290008
0x00000038	addi $10,$0,-2	000010	00000	01010	1111 1111 1111 1110	=	080AFFFE
0x0000003C	addi $10,$10,1	000010	01010	01010	0000 0000 0000 0001	=	094A0001
0x00000040	bltz $10,-2 (<0,转 3C)	110010	01010	00000	1111 1111 1111 1110	<	C940FFFE
0x0000003C	addi $10,$10,1	000010	01010	01010	0000 0000 0000 0001	=	094A0001
0x00000040	bltz $10,-2	110010	01010	00000	1111 1111 1111 1110	=	C940FFFE
0x00000044	andi $11,$2,2	010000	00010	01011	0000 0000 0000 0010	=	404B0002
0x00000048	j 0x00000050	111000	00000	00000	0000 0000 0001 0100	=	E0000014
0x0000004C	or $8,$4,$2	010011	00100	00010	01000 00000 000000	↓	4C823800
0x00000050	halt	111111	00000	00000	0000 0000 0000 0000	=	FC000000

b) **使用Vivado进行仿真验证**

创建仿真文件test_single_cycle_cpu.v,在仿真文件中实例化顶层模块cpu如下:

```
// 顶层模块 cpu 实例化
cpu uut(
    // input
    .CLK(CLK),
    .Reset(Reset),
    // output
    .op(op),
    .ReadData1(ReadData1),
    .ReadData2(ReadData2),
    .curPC(curPC),
    .nextPC(nextPC),
    .ALUresult(ALUresult),
    .instruction(instruction)
);
```

仿真文件关键代码如下:

```
// 仿真 CLK 和 Reset 相关代码
initial begin
    // 初始化 CLK 信号和 Reset 信号
    CLK = 0;
    Reset = 0;
    #50;
    Reset=1;
    CLK=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:

1) **第1~4条指令测试**

表 6 第1~4条指令

地址	汇编程序	op	rs	rt	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	add $3,$2,$1	000000	00010	00001	00011 00000 000000	00411800
0x0000000C	sub $5,$3,$2	000001	00011	00010	00101 00000 000000	04622800

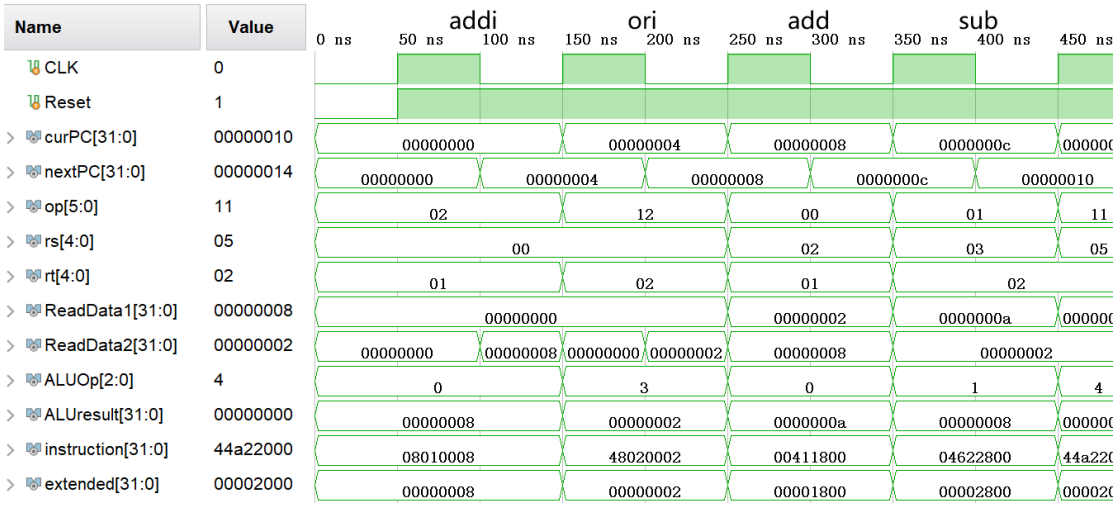

图 7 第1~4条指令波形图

i. addi $1,$0,8

如图 7所示，50~150ns：

- curPC为0x00000000，nextPC为0x00000004；
- rs对应寄存器位$0，$0=0（ReadData1）；
- rt对应寄存器位$1,$1=0(ReadData2),在时钟下降沿$1=$0+8=8；
- ALUOp为0，做加法运算，ALU运算结果为0+8=8。

ii. ori $2,$0,2

如图 7所示，150~250ns：

- curPC为0x00000004，nextPC为0x00000008；
- rs对应寄存器位$0，$0=0；
- rt对应寄存器位$1，$2=0，在时钟下降沿$2=2；
- ALUOp 为 3，做逻辑或运算，ALU 运算结果为 0|2=2。

iii. add $3,$2,$1

如图 7所示，250~350ns：

- curPC为0x00000008，nextPC为0x0000000C；
- rs对应寄存器位$2，$2=2；rt对应寄存器位$1，$1=8；
- ALUOp 为 0，做加法运算，ALU 运算结果为 2+8=10(a)。

iv. sub $5,$3,$2

如图 7所示，350~450ns：

- curPC为0x0000000C，nextPC为0x00000010；
- rs对应寄存器位$3，$3=10；rt对应寄存器位$2，$1=2；
- ALUOp 为 1，做减法运算，ALU 运算结果为 10-2=8。

2） 第5~7条指令测试

表 7 第 5~7 条指令

地址	汇编程序	op	rs	rt	rd(5)/immediate (16)	16 进制数代码
0x00000010	and $4,$5,$2	010001	00101	00010	00100 00000 000000	44A22000
0x00000014	or $8,$4,$2	010011	00100	00010	01000 00000 000000	4C824000
0x00000018	sll $8,$8,1	011000	00000	01000	01000 00001 000000	60084040

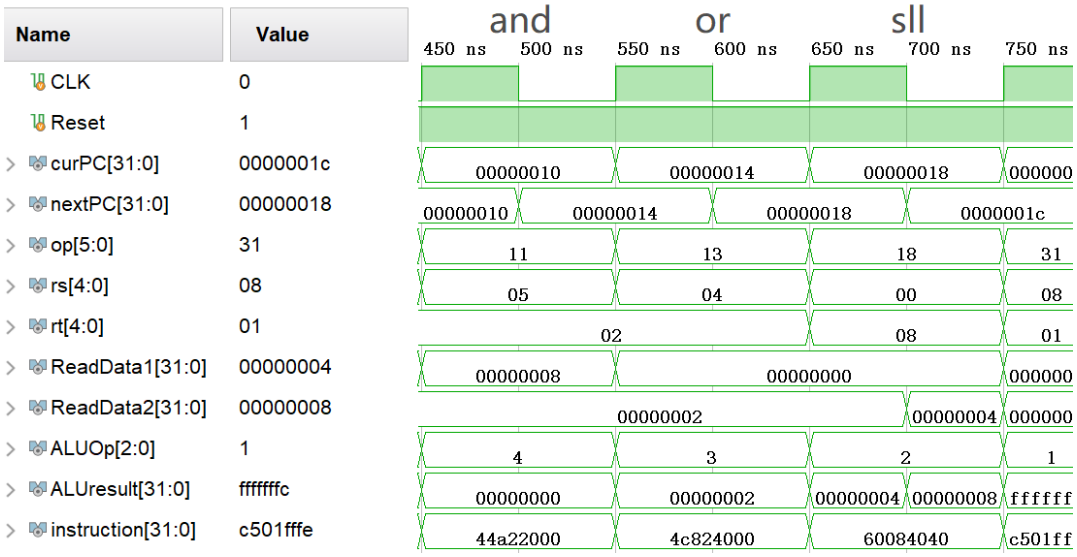

图 8 第 5~7 条指令波形图

v. and $4,$5,$2

如图 8所示，450~550ns：

- curPC为0x00000010，nextPC为0x00000014；
- rs对应寄存器位$5，$5=8；rt对应寄存器位$2，$2=2；
- ALUOp 为 4，做与运算，ALU 运算结果为 8&2=0。

vi. or $8,$4,$2

如图 8所示，550~650ns：

- curPC为0x00000014，nextPC为0x00000018；
- rs对应寄存器位$4，$4=0；rt对应寄存器位$2，$2=2；
- ALUOp 为 3，做或运算，ALU 运算结果为 0|2=2。

vii. sll $8,$8,1

如图 8所示，650~750ns：

- curPC为0x00000018，nextPC为0x0000001C；
- rs对应寄存器位$0，$0=0；
- rt对应寄存器位$8，$8=2，在时钟下降沿，$8=2<<1=4；

3） 第8~11条指令测试

地址	汇编程序	op	rs	rt	rd(5)/immediate (16)	16 进制数代码
0x0000001C	bne $8,$1,-2 (≠,转 18)	110001	01000	00001	1111 1111 1111 1110	C501FFFE
0x00000018	sll $8,$8,1	011000	00000	01000	01000 00001 000000	60084040
0x0000001C	bne $8,$1,-2	110001	01000	00001	1111 1111 1111 1110	C501FFFE
0x00000020	slti $6,$2,4	011100	00010	00110	0000 0000 0000 0100	70460004

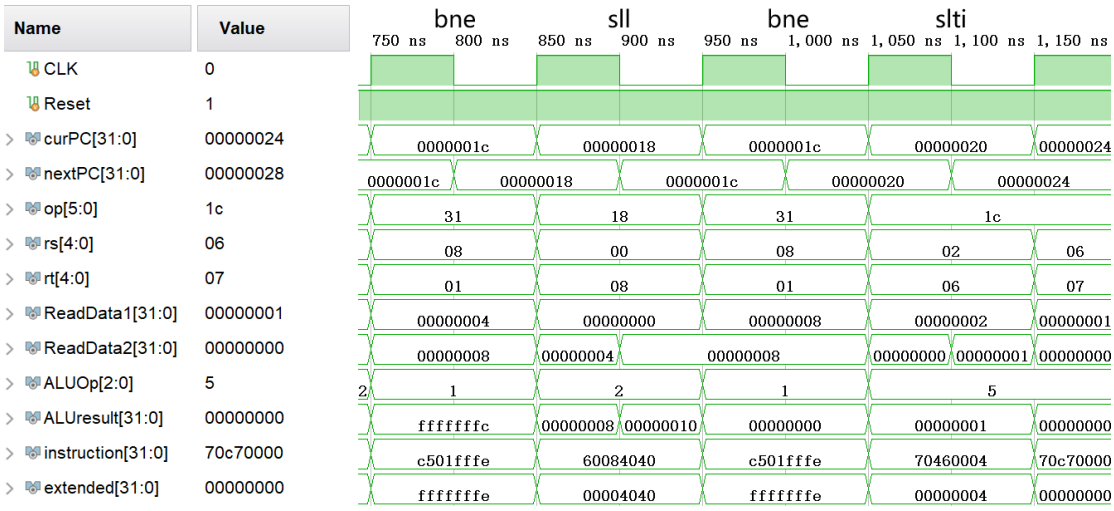

图 9 第 8~11 条指令

viii. bne $8,$1,-2 （≠,转18）

如图 9所示，750~850ns：

- curPC为0x00000018。因为$8=4，$1=8，两者不相等(zero=0)，故跳转至18，nextPC为0x00000018；
- ALUOp为1，做减法运算，ALU运算结果为4-8=-4，补码为0xfffffffc；
- rs对应寄存器位$8，$8=4；rt对应寄存器位$1，$1=8。

ix. sll $8,$8,1

如图 9所示，850~950ns：

- curPC为0x00000018，nextPC为0x0000001C；
- rs对应寄存器位$0，$0=0；
- rt对应寄存器位$1，$8=4，在时钟下降沿，$8=8；
- ALUOp 为 2，做左移运算，ALU 运算结果为 4<<1=8。

x. bne $8,$1,-2 （=，不跳转）

如图 9所示，950~1050ns：

- curPC为0x0000001C。因为$8=$1=8，两者相等（zero=1），故不发生跳转，nextPC为0x00000020；
- rs对应寄存器位$8，$8=8；rt对应寄存器位$1，$1=8。

xi. slti $6,$2,4

如图 9所示，1050~1150ns：

- curPC为0x00000020，nextPC为0x00000024；
- rs对应寄存器位$2，$2=2；
- rt对应寄存器位$6，$6=0，在时钟下降沿，$6=1；
- ALUOp 为 5,做不带符号比较运算,ALU 运算结果为(2<4?1:0)=1。

4） 第12~14条指令测试

表 8 第 12~14 条指令

地址	汇编程序	op	rs	rt	rd(5)/immediate (16)	16 进制数代码
0x00000024	slti $7,$6,0	011100	00110	00111	0000 0000 0000 0000	70C70000
0x00000028	addi $7,$7,8	000010	00111	00111	0000 0000 0000 1000	08E70008
0x0000002C	beq $7,$1,-2 (=,转 28)	110000	00111	00001	1111 1111 1111 1110	C0E1FFFE

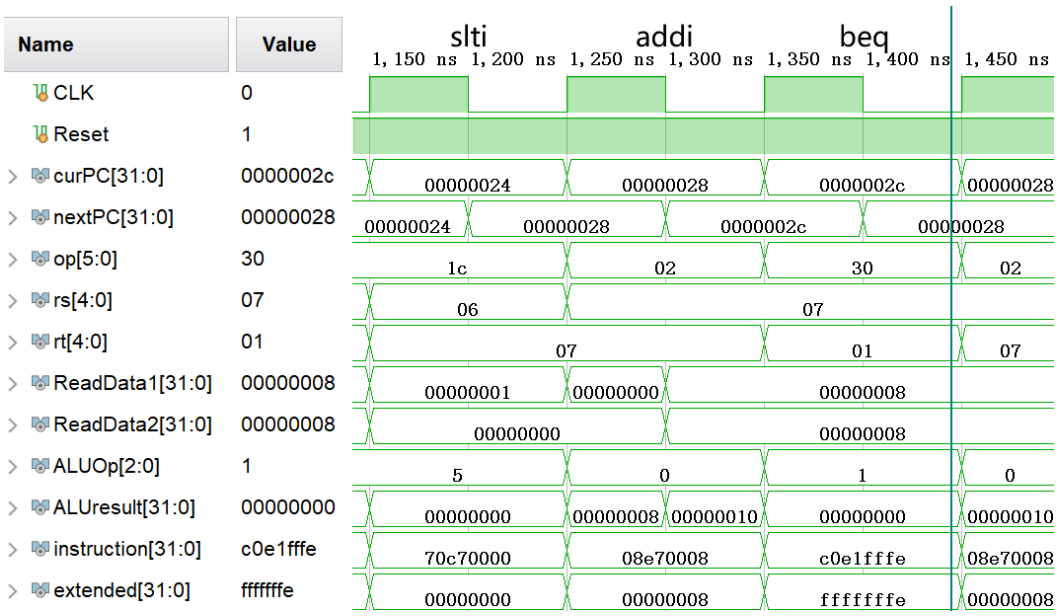

图 10 第 12~14 条指令

xii. slti $7,$6,0

如图 10所示，1150~1250ns：

- curPC为0x00000024，nextPC为0x00000028；
- ALUOp 为 5,做不带符号的比较运算,ALU 运算结果为(1<0?1:0)=0；
- rs对应寄存器位$6，$6=1；rt对应寄存器位$7，$7=0。

xiii. addi $7,$7,8

如图 10所示，1250~1350ns：

- curPC为0x00000028，nextPC为0x0000002C；
- rs和rt应寄存器位$7，$7=0；在时钟下降沿，$7=8；
- ALUOp为0，做加法运算，ALU运算结果为0+8=8。

xiv. beq $7,$1,-2 (=,转28)

如图 10所示，1350~1450ns：

- curPC为0x0000002C。因为$7=8，$1=8，两者相等（zero=1），故跳转至28，nextPC为0x00000028；
- ALUOp为1，做减法运算，ALU运算结果为8-8=0。

5) 第15~17条指令测试

地址	汇编程序	op	rs	rt	rd(5)/immediate (16)	16 进制数代码
0x00000028	addi $7,$7,8	000010	00111	00111	0000 0000 0000 1000	08E70008
0x0000002C	beq $7,$1,-2	110000	00111	00001	1111 1111 1111 1110	C0E1FFFE
0x00000030	sw $2,4($1)	100110	00001	00010	0000 0000 0000 1000	98220008

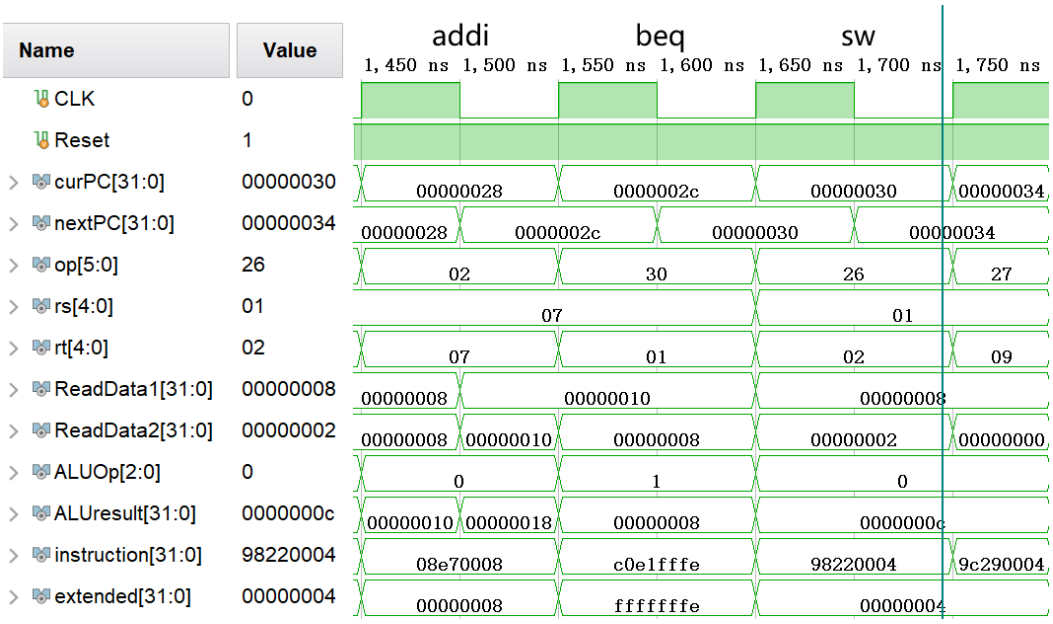

图 11 第 15~17 条指令

xv. addi $7,$7,8

如图 11所示，1450~1550ns：

- curPC为0x00000028，nextPC为0x0000002C；
- ALUOp为0，做加法运算，ALU运算结果为8+8=16。
- rs和rt应寄存器位$7，$7=8；在时钟下降沿，$7=16。

xvi. beq $7,$1,-2 (≠,不跳转)

如图 11所示，1550~1650ns：

- curPC为0x0000002C。因为$7=16,$1=8,两者不相等(zero=0)，故不跳转，nextPC为0x00000030；
- ALUOp 为 1，做减法运算，ALU 运算结果为 16-8=8。

xvii. sw $2,4($1)

如图 11所示，1650~1750ns：

- curPC为0x00000030，nextPC为0x00000034；
- rs对应寄存器位$1，$1=8； rt对应寄存器位$2，$2=2；
- ALUOp 为 0，做加法运算，此时 ALU 输入 A 为 8 (ReadData1)、输入 B 为立即数（extended）4，ALU 运算结果为 8+4=12；
- 将rt寄存器的值(2)写入数据存储器，写入地址为ALU计算结果(12)。

6) 第18~20条指令测试

表 9 第 18~20 条指令

地址	汇编程序	op	rs	rt	rd(5)/immediate (16)	16 进制数代码
0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 1000	9C290008
0x00000038	addi 10,$0,-2	000010	00000	01010	1111 1111 1111 1110	080AFFFE
0x0000003C	addi 10,$10,1	000010	01010	01010	0000 0000 0000 0001	094A0001

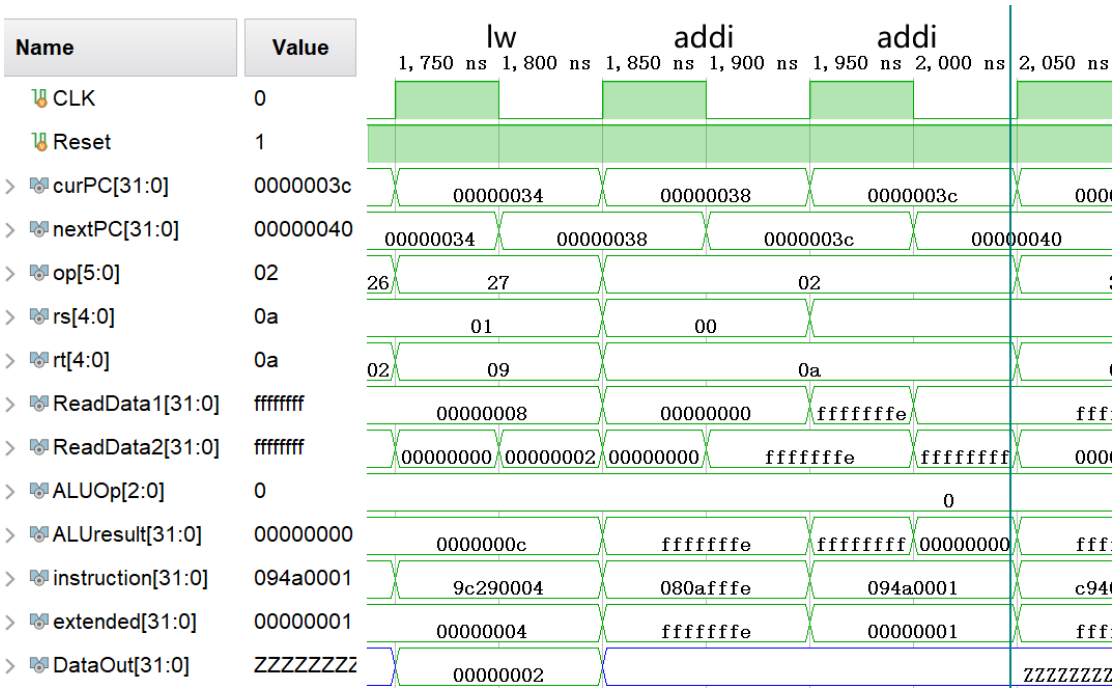

图 12 第 18~20 条指令波形图

xviii. lw $9,4($1)

如图 12所示，1750~1850ns：

- curPC为0x00000034，nextPC为0x00000038；
- rs对应寄存器位$1，$1=8；
- rt对应寄存器位$8，$9=0，在时钟下降沿，$9=2；
- 立即数为 4，ALUOp 为 0，做加法运算，ALU 运算结果为 8+4=12；
- 将数据存储器值（DataOut）读出至$9，存储器地址为ALU结果。

xix. addi $10,$0,-2

如图 12所示，1850~1950ns：

- curPC为0x00000038，nextPC为0x0000003C；
- rs对应寄存器位$0，$0=0；
- rt对应寄存器位$10，$10=0，在时钟下降沿，$10=fffffffe（-2）；
- ALUOp 为 0，做加法运算，ALU 运算结果为 0-2=-2(补码为 fffffffe)。

xx. addi $10,$10,1

如图 12所示，1950~2050ns：

- curPC为0x0000003C，nextPC为0x00000040；
- rs对应寄存器位$10，$10=fffffffe（-2）；
- rt对应寄存器位$10，$10=fffffffe，在时钟下降沿，$10=ffffffff(-1)；
- ALUOp 为 0，做加法运算，ALU 运算结果为-2+1=-1。

7） 第21~24条指令测试

表 10 第 21~24 条指令

地址	汇编程序	op	rs	rt	rd(5)/immediate (16)	16 进制数代码
0x00000040	bltz $10,-2 (<0,转 3C)	110010	01010	00000	1111 1111 1111 1110	C940FFFE
0x0000003C	addi $10,$10,1	000010	01010	01010	0000 0000 0000 0001	094A0001
0x00000044	andi $11,$2,2	010000	00010	01011	0000 0000 0000 0010	404B0002
0x00000040	bltz $10,-2	110010	01010	00000	1111 1111 1111 1110	C940FFFE

xxi. bltz $10,-2(<0,转3C)

如图 13所示，2050~2150ns：

- curPC为0x00000040，因为$10=-1<0（sign=1），故跳转至3C，nextPC为0x0000003C；
- ALUOp 为 0，做加法运算，ALU 运算结果为-1+0=-1。

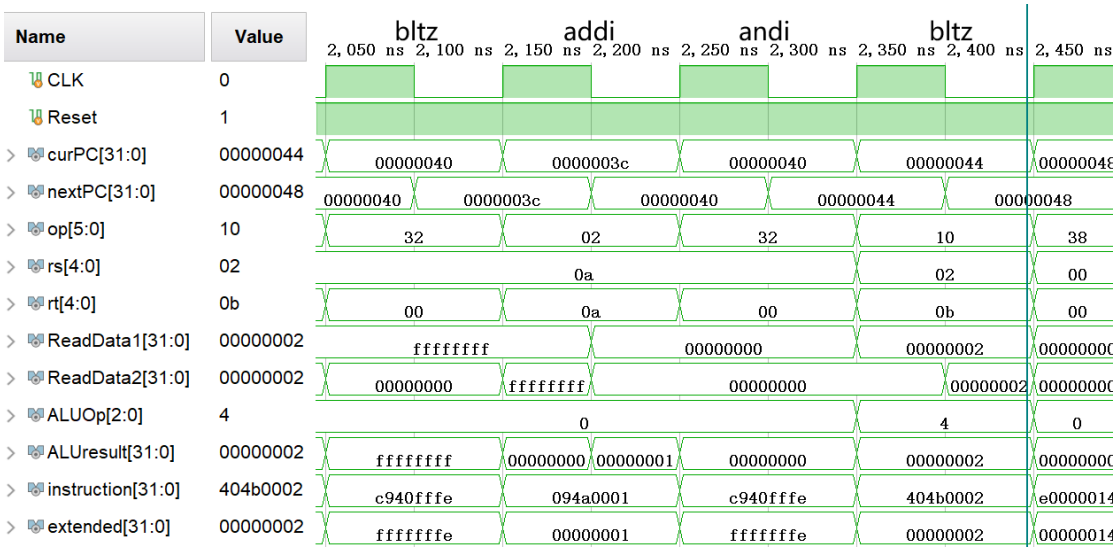

图 13 第 21~24 条指令

xxii. addi $10,$10,1

如图 13所示，2150~2250ns：

- curPC为0x0000003C，nextPC为0x00000040；
- rs和rt对应寄存器位$10,$10=ffffffff(-1)；在时钟下降沿,$10=0；
- 立即数为 1，ALUOp 为 0，做加法运算，ALU 运算结果为-1+1=0。

xxiii. bltz $10,-2(=0,不跳转)

如图 13所示，2250~2350ns：

- curPC为0x00000040。因为$10=0 (sign=0)，故不跳转，nextPC为0x00000044；
- ALUOp 为 0，做加法运算，ALU 运算结果为 0+0=0。

xxiv. andi $11,$2,2

如图 13所示，2350~2450ns：

- curPC为0x00000044，nextPC为0x00000048；
- rs对应寄存器位$2，$2=2；rt对应寄存器位$11，$11=0；
- 立即数为 2，ALUOp 为 4，做与运算，ALU 运算结果为 2&2=2。

8） 第25~27条指令测试

表 11 第 25~27 条指令

地址	汇编程序	op	rs	rt	rd(5)/immediate (16)	16 进制数代码
0x00000048	j 0x00000050	111000	00000	00000	0000 0000 0001 0100	E0000014
0x0000004C	or $8,$4,$2	010011	00100	00010	01000 00000 000000	4C823800
0x00000050	halt	111111	00000	00000	0000 0000 0000 0000	FC000000

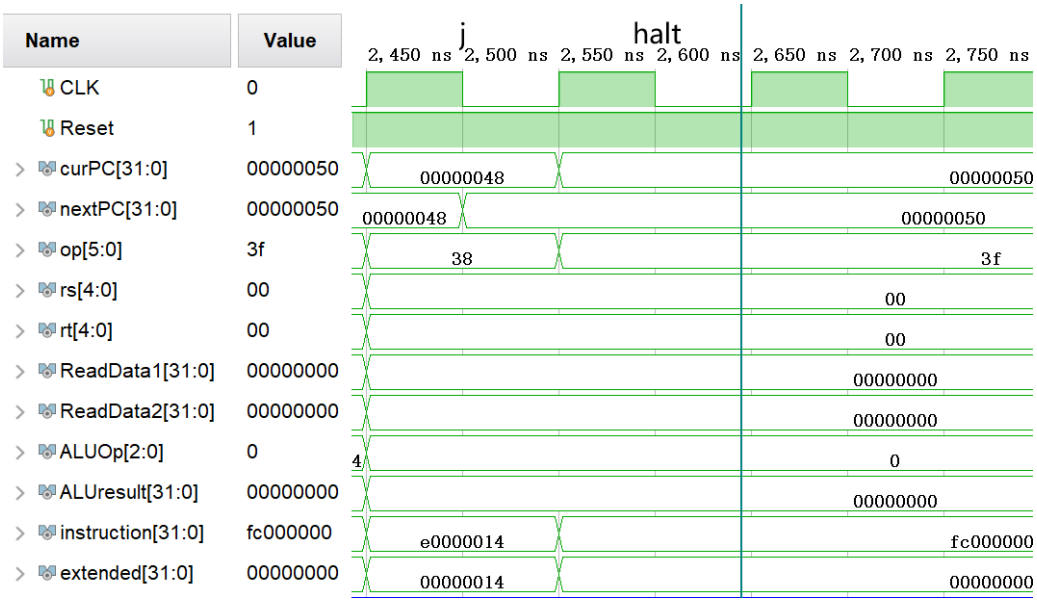

图 14 第 25~27 条指令波形图

xxv. j 0x00000050

如图 14所示，2450~2550ns：

- curPC 为 0x00000048，j 型指令根据输入地址进行跳转，nextPC 为 0x00000050；
- J型指令不涉及寄存器堆、数据存储器、ALU，与无关控制信号。

xxvi. or $8,$4,$2

j指令跳转后，该指令被跳过，故不执行。

xxvii. halt

如图 14所示，2550ns后：

- curPC 为 0x00000050，nextPC 为 0x00000050；正确。
- halt指令触发停机状态，各个模块和信号量不再发生改变。

六. 实验心得

开始着手本次实验遇到较多问题。

首先，对单周期MIPS CPU的工作流程没有充分的认识，同时对Verilog语言没有一丁点认知，导致实验开始时无从下手。后面通过学习Verilog基础语法和分析图 3 单周期CPU数据通路和控制线路图，逐渐了解了寄存器reg类型（寄存器需要初始化；initial和always中赋值对象一定是寄存器）和线wire类型（仅仅起到导线连接作用，线两端变量的值同步改变）、Verilog设计时遵循模块（module）化设计原则、阻塞赋值与非阻塞赋值、单周期CPU工作时具体的数据通路变化和各个模块在工作中起到的作用。

其次，在设计具体模块时要选择恰当的敏感信号以触发模块的具体功能，选择敏感信号时不能混用时钟信号和电平信号：组合逻辑电路使用阻塞赋值“=”；时序逻辑电路使用非阻

塞赋值“<=”。同时，还需要注意各个寄存器改变的时机，寄存器存储器和数据存储器写指令在时钟下降沿触发；在时钟下降沿计算nextPC；在时钟上升沿发生改变curPC。起初按照CPU数据通路图设计模块时，将数据选择器全部单独设计，后面发现可以将部分数据选择器整合入模块中，以减少模块设计的复杂度和代码的冗余。

最后是实现过程中的小错误，如：在control_unit中遗漏部分操作的编码（andi），造成仿真测试时所得数据与预期不相符；在ins_seg对指令的划分没有对照好字段相关规则，造成指令划分错误；没有分清各类数据的具体大小（如立即数扩展前为16位，经过扩展才为32位等）。

经过本次单周期MIPS CPU设计实验，从头到尾彻底梳理了单周期CPU工作时具体的数据通路和MIPS指令的具体执行过程，加深了对CPU执行一条指令时五个阶段各自作用的理解。同时，对Verilog这一用于设计硬件的语言有了更多的了解，学会将元件分模块化设计，由模块实现各部分功能，由各模块共同协作完成元件的目标功能。