

成绩：

实验二：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的测试方法；
5. 掌握多周期CPU的实现方法。

二. 实验内容

设计一个多周期 CPU，该 CPU 至少能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： $rd \leftarrow rs + rt$ 。

(2) sub rd, rs, rt

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

完成功能： $rd \leftarrow rs - rt$ 。

(3) addiu rt, rs, **immediate**

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

功能： $rt \leftarrow rs + (\text{sign-extend})\text{immediate}$ 。

==>逻辑运算指令

(4) and rd, rs, rt

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

功能： $rd \leftarrow rs \& rt$ ；逻辑与运算。

(5) andi rt, rs, **immediate**

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

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

(6) ori rt, rs, **immediate**

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

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

(7) xori rt, rs, **immediate**

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

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

==>移位指令

(8) sll rd, rt, sa

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

功能: $rd \leftarrow rt \ll (\text{zero-extend})sa$, 左移 sa 位, $(\text{zero-extend})sa$ 。

==>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, 具体请看表 2 ALU 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: $\text{memory}[rs + (\text{sign-extend})immediate] \leftarrow rt$ 。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: $rt \leftarrow \text{memory}[rs + (\text{sign-extend})immediate]$ 。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) $pc \leftarrow pc + 4 + ((\text{sign-extend})immediate \ll 2)$ else $pc \leftarrow pc + 4$ 。

(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) $pc \leftarrow pc + 4 + ((\text{sign-extend})immediate \ll 2)$ else $pc \leftarrow pc + 4$ 。

(15) bltz rs, immediate

110110	rs(5 位)	00000	immediate
--------	---------	-------	-----------

功能: if (rs < \$0) $pc \leftarrow pc + 4 + ((\text{sign-extend})immediate \ll 2)$ else $pc \leftarrow pc + 4$ 。

==>跳转指令

(16) j addr

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

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

(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	0000000000000000000000000000(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，这就是所谓的多周期 CPU。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，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。

实验中就按照这五个阶段进行设计，这样一条指令的执行最长需要五个(小)时钟周期才能完成，但具体情况怎样？要根据该条指令的情况而定，有些指令不需要五个时钟周期的，这就是多周期的 CPU。

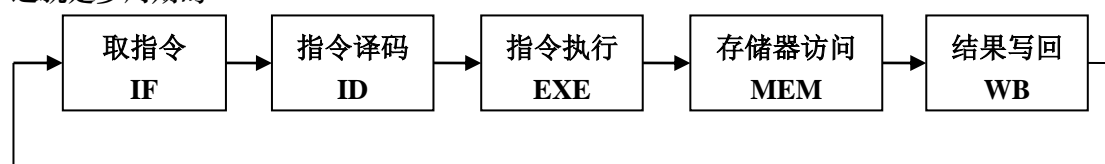
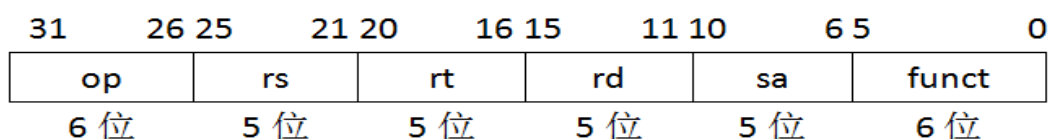


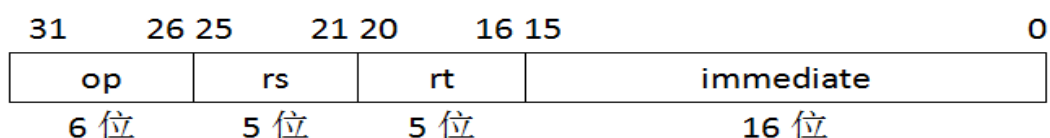
图 1 多周期 CPU 指令处理过程

MIPS 指令的三种格式:

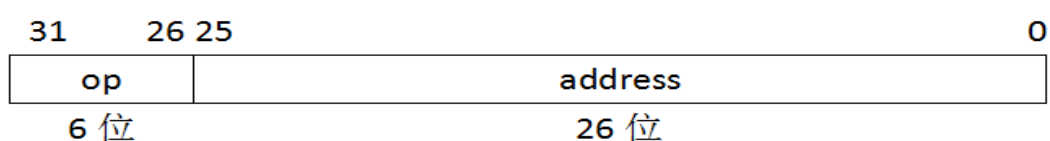
R 类型:



I 类型:



J 类型:



其中,

op: 为操作码;

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

rt: 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: 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;

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。

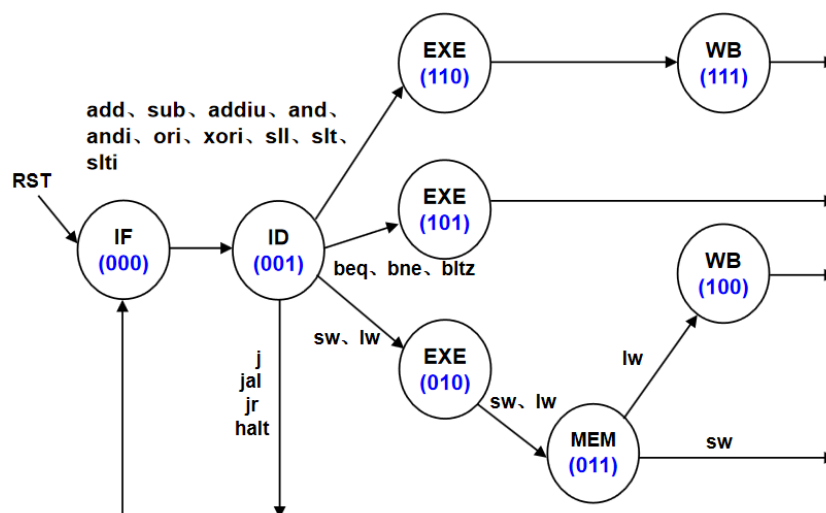


图 2 多周期 CPU 状态转移图

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

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。

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’ 状态之外, 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’ 状态时, 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: $pc \leftarrow pc+4$, 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: $pc \leftarrow pc+4+(sign-extend)immediate \times 4$, 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

表 1 控制信号作用

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

DataIn, 存储器数据输入端口

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

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	$Y = (((A < B) \&\& (A[31] == B[31])) \parallel ((A[31] == 1 \&\& B[31] == 0))) ? 1 : 0$	比较 $A < B$ 带符号
111	$Y = A \oplus B$	异或

表2 ALU运算功能表

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. 汇编器

汇编器的作用是把汇编代码转换为机器码, 只需要读取每一条指令, 并且根据指令的格式构造机器码即可。

我使用Python 3编写了汇编器, 代码见末尾的“附录B”。

B. CPU设计

● 底层模块设计

单周期CPU每个时钟周期可被划分为五个阶段, 对应地有五个最关键的模块: IF阶段对应PC, ID阶段对应InstructionMemory和RegisterFile, EXE阶段对应ALU, MEM阶段对应DataMemory, WB阶段也对应RegisterFile。除此之外, 还需要生成控制信号的控制单元 (ControlUnit), 还要有对立即数进行扩展的ImmediateExtend。最后, 还需要使用各种不同的数据选择器 (Mux) 构成完整的数据通路。上面提到的这些模块在实验内容 - 图2的数据通路图中全部都有体现。

基于以上思想, 首先开始分别设计每一个底层模块。

1. PC

PC是时序逻辑, 该模块与“单周期CPU设计与实现”实验中的对应模块完全相同, 代码如下:

```
module PC(
    input clk, input Reset, input PCWre, input [31:0] nextIAAddr,
    output reg [31:0] currentIAAddr
);
    initial currentIAAddr <= 0;
```



```

always @(posedge clk or negedge Reset) begin
    if(Reset == 0) currentIAddr <= 0;
    else begin
        if(PCWre == 1) currentIAddr <= nextIAddr;    // PC接收新地址
        else currentIAddr <= currentIAddr;    // PC不更改
    end
end
endmodule

```

2. InstructionMemory

InstructionMemory是组合逻辑。值得注意的是，在实际进行实验时，IDataIn和RW输入没有用到，因此不做实现（在代码文件中这部分内容被注释掉了）。

该模块被实现为组合逻辑，被设计为内含128个字节的ROM。ROM中存放要执行的测试程序段的机器码，使用initial语句中的\$readmemb伪指令从test_instructions.txt文件中读取出来。该外部文件的内容见附录B。

该模块输入一个32位长的IAddr，输出对应地址中的32位机器指令。该模块与“单周期CPU设计与实现”实验中的对应模块完全相同，代码如下：

```

module InstructionMemory(
    input [31:0] IAddr,
    output [31:0] IDataOut
);
    reg [7:0] ROM [0:127];
    initial begin
        $readmemb("E:/_Vivado/MIPS_CPU_Design/MultiCycleCPU/test_instructions.txt",
        ROM);
    end
    assign IDataOut[31:24] = ROM[IAddr+0];
    assign IDataOut[23:16] = ROM[IAddr+1];
    assign IDataOut[15:8] = ROM[IAddr+2];
    assign IDataOut[7:0] = ROM[IAddr+3];
endmodule

```

3. RegisterFile

读寄存器堆中是组合逻辑的功能，而我将写寄存器堆实现为时序逻辑。前者只需用assign赋值，后者则需要clk下降沿触发写入。该模块与“单周期CPU设计与实现”实验中的对应模块完全相同，代码如下：

```

module RegisterFile(
    input clk,
    input Reset,
    input WE,    // 寄存器堆写使能, 1为有效
    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] file [1:31];

```

```

integer i;
assign ReadData1 = (ReadReg1 == 0) ? 0 : file[ReadReg1];
assign ReadData2 = (ReadReg2 == 0) ? 0 : file[ReadReg2];
always @(negedge clk or negedge Reset) begin
    if(Reset == 0) begin
        for(i = 1; i <= 31; i=i+1) begin
            file[i] <= 0;
        end
    end
    else if(WE == 1 && WriteReg != 0)
        file[WriteReg] <= WriteData;
    end
endmodule

```

4. ALU

ALU是组合逻辑，该模块与“单周期CPU设计与实现”实验中的对应模块

完全相同，代码如下：

```

module ALU(
    input [2:0] ALUOp, input [31:0] A, input [31:0] B,
    output reg [31:0] result,
    output zero,    // 结果是否为0? 是为1, 否为0
    output sign     // 结果是否为负? 是为1, 否为0
);
assign zero = (result == 0) ? 1 : 0;
assign sign = result[31];
always @(ALUOp or A or B) begin
    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; // 5: 比较无符号数
        3'b110: begin                // 6: 比较带符号数
            if((A[31] == B[31]) && (A < B)) result = 1;
            else if(A[31]==1 && B[31]==0) result = 1;
            else result = 0;
        end
        3'b111: result = A ^ B;
    endcase
end
endmodule

```

5. DataMemory

与FileRegister类似地，读存储器为组合逻辑，写存储器为时序逻辑。前者只需用assign赋值，后者要在clk下降沿触发写入。该模块与“单周期CPU设计与实现”实验中的对应模块完全相同，代码如下：

```

module DataMemory(
    input clk, input [31:0] DAddr, input [31:0] DataIn,
    input RD,    // 读控制, 1有效
    input WR,    // 写控制, 1有效
    output [31:0] DataOut    // 读出32位数据
);
reg [7:0] RAM [0:63];    // 每个内存单元为8位, 即一个字节
/* 读 */
assign DataOut[7:0] = (RD==1) ? RAM[DAddr+3] : 8'bz;
assign DataOut[15:8] = (RD==1) ? RAM[DAddr+2] : 8'bz;

```

```

assign DataOut[23:16] = (RD==1) ? RAM[DAddr+1] : 8'bz;
assign DataOut[31:24] = (RD==1) ? RAM[DAddr+0] : 8'bz;
/* 写 */
always @(negedge clk) begin
    if(WR == 1) begin
        RAM[DAddr+0] <= DataIn[31:24];
        RAM[DAddr+1] <= DataIn[23:16];
        RAM[DAddr+2] <= DataIn[15:8];
        RAM[DAddr+3] <= DataIn[7:0];
    end
end
endmodule

```

6. IR

根据数据通路图容易知道IR的作用：在IRWre信号的控制下，在时钟边沿处将输入所存起来。需要注意的是，因为PC在时钟上升沿时发生改变，因此为了保证指令的稳定，IR应该在时钟下降沿处改变，与PC的改变相差半个周期。

代码如下：

```

module IR(
    input clk,
    input IRWre,
    input [31:0] insIn,
    output reg [31:0] insOut
);
always @(negedge clk) begin
    if(IRWre==1) insOut <= insIn;
    else insOut <= insOut;
end
endmodule

```

7. ADR、BDR和DBDR

这三个寄存器的实现是一样的，都是D触发器，因此设计底层模块DFF_32bits（意为作用于32位数据的D触发器）。为了保证模块的完整性，我在此模块中加入了Reset输入，当Reset为0有效时，将触发器清零。代码如下：

```

module DFF_32bits(
    input clk,
    input Reset,
    input [31:0] in,
    output reg [31:0] out
);
always @(posedge clk or negedge Reset) begin
    if(Reset==0) out <= 0;
    else out <= in;
end
endmodule

```

8. ControlUnit

与“单周期CPU设计与实现”实验不同，多周期CPU的ControlUnit是时

序逻辑，这是因为在多周期CPU中，一条指令的执行被分为了2~5个状态分步执行。状态转移图实验原理部分的图2。在多周期CPU中，控制信号不仅依赖于当前指令的操作码（opcode），还依赖于当前正处于的状态（state）。

在所有控制信号中，有四个涉及“写入”操作的信号，分别是PCWre、IRWre、mWR和RegWre，除了这四个信号之外，其余的信号不涉及写入操作，因此可以认为它们与状态无关（即在任何状态下都不会改变）。我们先来考虑除了PCWre、IRWre、mWR和RegWre以外的控制信号。很容易地可以列出如下表示指令与控制信号之间的关系的表格：

指令	opcode	ALUSrcA	ALUSrcB	DBDataSrc	WrRegDSrc	mRD	ExtSel	PCSrc[1:0]	RegDst[1:0]	ALUOp[2:0]
add	000000	0	0	0	1	0	0	00	10	000
sub	000001	0	0	0	1	0	0	00	10	001
addiu	000010	0	1	0	1	0	1	00	01	000
and	010000	0	0	0	1	0	0	00	10	100
andi	010001	0	1	0	1	0	0	00	01	100
ori	010010	0	1	0	1	0	0	00	01	011
xori	010011	0	1	0	1	0	0	00	01	111
sll	011000	1	0	0	1	0	0	00	10	010
slti	100110	0	1	0	1	0	1	00	01	110
slt	100111	0	0	0	1	0	0	00	10	110
sw	110000	0	1	0	1	0	1	00	00	000
lw	110001	0	1	1	1	1	1	00	01	000
beq	110100	0	0	0	1	0	1	01(zero)/00	00	001
bne	110101	0	0	0	1	0	1	01(!zero)/00	00	001
bltz	110110	0	0	0	1	0	1	01(sign)/00	00	000
j	111000	0	0	0	1	0	0	11	00	000
jr	111001	0	0	0	1	0	0	10	00	000
jal	111010	0	0	0	0	0	0	11	00	000
halt	111111	0	0	0	1	0	0	00	00	000

然后我们再来考虑PCWre、IRWre、mWR和RegWre这四个信号，分析过程如下：

PC的改变一定发生在sIF状态的时钟上升沿，因此应该让PCWre信号在前一个状态的时钟下降沿处直到sIF状态的时钟下降沿处保持有效电平。不过，如果当前指令是halt停机，那么PCWre一直保持无效，不用考虑状态。

经过研究发现，IRWre信号与PCWre信号的功能非常类似，都是控制指令是否发生改变的，因此我们可以让IRWre始终为有效，稍后可以验证这样做是

正确的。

mWR是存储器写使能信号，在要实现的指令中，只有sw指令的MEM状态（状态号为011）需要写存储器，这是mWR信号有效的唯一情况；其余情况mWR信号均保持无效。

RegWre是寄存器写使能信号，在R型指令和I型指令的WB状态（状态号为111）时RegWre需要有效；此外，jal指令的ID状态（状态号为001）也要写寄存器，这时RegWre也要有效；其余情况RegWre信号均保持无效。

基于以上分析，就可以写出ControlUnit的代码了。完整代码见ControlUnit.v文件，这里只给出PCWre信号的生成代码：

```
always @(negedge clk) begin
    case(state)
        3'b111, 3'b101, 3'b100: PCWre <= 1;
        3'b011: PCWre <= (opcode==6'b110000 ? 1 : 0); // sw
        3'b001: PCWre <= (opcode==6'b111000 | opcode==6'b111001 | opcode==6'b111010 ? 1 : 0);
            // j, jr, jal
        default: PCWre <= 0;
    endcase
end
```

9. ImmediateExtend

该模块为组合逻辑。该模块与“单周期CPU设计与实现”实验中的对应模块完全相同，代码如下：

```
module ImmediateExtend(
    input [15:0] original, input ExtSel, // 0: Zero-extend; 1: Sign-extend.
    output reg [31:0] extended
);
always @(*) begin
    extended[15:0] <= original; // 低16位保持不变
    if(ExtSel == 0) begin // Zero-extend 零扩展
        extended[31:16] <= 0;
    end
    else begin // Sign-extended 符号扩展
        if(original[15] == 0) extended[31:16] <= 0;
        else extended[31:16] <= 16'hFFFF;
    end
end
endmodule
```

10. 数据选择器

根据数据通路图可知，至少需要6个数据选择器，其中有1个32位四选一选择器、1个5位二选一选择器、3个32位二选一选择器。不同的数据选择器实现原理非常类似，因此下面只贴出32位四选一选择器的代码，其它数据选择器的代码见以“Mux_”开头的文件。

```
module Mux4_32bits(
```

```

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

```

● 顶层模块设计

顶层模块的作用是实例化各个底层模块,并使用导线将它们按照数据通路图(实验内容 - 图4)连接起来。限于篇幅具体代码不放于此,见工程文件中的top_CPU.v文件。

此时,单周期CPU就全部设计完成了(不包括烧板)。工程目录结构如图示:



图5 top_CPU工程结构树

C. 仿真实验验证CPU的正确性

● CPU执行测试程序段的过程

根据老师提供的“关于测试多周期CPU的简单方法”文档中的测试程序

段，可以推导得到指令的实际执行过程、寄存器的变化、指令跳转情况以及 ALU 的运算结果等信息，列为下表。注意该表已经展开了所有的分支结构，因此这个表格从上到下是以时间顺序执行的。

指令地址	汇编程序	周期数	寄存器变化 (十进制)	跳转情况	ALU 结果低 8 位 (十六进制)
0x00000000	addiu \$1,\$0,8	4	\$1 = 8		
0x00000004	ori \$2,\$0,2	4	\$2 = 2		
0x00000008	xori \$3,\$2,8	4	\$3 = 10		
0x0000000C	sub \$4,\$3,\$1	4	\$4 = 2		
0x00000010	and \$5,\$4,\$2	4	\$5 = 2		
0x00000014	sll \$5,\$5,2	4	\$5 = 8		
0x00000018	beq \$5,\$1,-2	3		=, 转 14	
0x00000014	sll \$5,\$5,2	4	\$5 = 32		
0x00000018	beq \$5,\$1,-2	3		≠, 顺序	
0x0000001C	jal 0x00000050	2	\$31 = 20	转 50	
0x00000050	sw \$2,4(\$1)	4			
0x00000054	lw \$13,4(\$1)	5	\$13 = 2		
0x00000058	jr \$31	2		转 20	
0x00000020	slt \$8,\$13,\$1	4	\$8 = 1		
0x00000024	addiu \$14,\$0,-2	4	\$14 = -2		
0x00000028	slt \$9,\$8,\$14	4	\$9 = 0		
0x0000002C	slti \$10,\$9,2	4	\$10 = 1		
0x00000030	slti \$11,\$10,0	4	\$11 = 0		
0x00000034	add \$11,\$11,\$10	4	\$11 = 1		
0x00000038	bne \$11,\$2,-2	3		≠, 转 34	
0x00000034	add \$11,\$11,\$10	4	\$11 = 2		
0x00000038	bne \$11,\$2,-2	3		=, 顺序	
0x0000003C	addiu \$12,\$0,-2	4	\$12 = -2		
0x00000040	addiu \$12,\$12,1	4	\$12 = -1		
0x00000044	bltz \$12,-2	3		<0, 转 40	
0x00000040	addiu \$12,\$12,1	4	\$12 = 0		

0x00000044	bltz \$12,-2	3		≠0, 顺序	
0x00000048	andi \$12,\$2,2	4	\$12 = 2		
0x0000004C	j 0x0000005C	2		转 5C	
0x0000005C	halt			停机	

有了这个表格，我们就可以开始仿真并检验CPU的结果了。

● 在Vivado中仿真

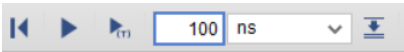
创建仿真文件CPU_sim_1.v，在其中实例化top_CPU模块，使用如下输入和输出来进行仿真：

输入	clk	CPU时钟信号
	Reset	CPU复位信号，0为复位
输出	currentIAddr	当前PC地址
	nextIAddr	下一条指令地址
	rs	rs寄存器编号
	rt	rt寄存器编号
	ReadData1	rs寄存器数据
	ReadData2	rt寄存器数据
	ALU_result	ALU运算结果
	DataOut	数据总线数据

仿真代码的关键部分如下：

```
always #50 clk = ~clk; // 仿真时钟周期为100ns
initial begin
    clk = 1;
    Reset = 0;
    #25;
    Reset = 1; // 开始仿真
    #10000; // 进行10000ns的仿真
    $stop; // 断点
end
```

设置仿真时间单位与仿真时钟周期相同，为100ns，然后通过鼠标点击单步执行（即一次仿真一个时钟周期）。下面开始验证实验内容要求的19条指令逐一验证。



说明：本实验报告中，重复的指令操作只叙述一次。在每个指令的叙述过程中，只对该指令相关的控制信号做说明，无关的控制信号将忽略。

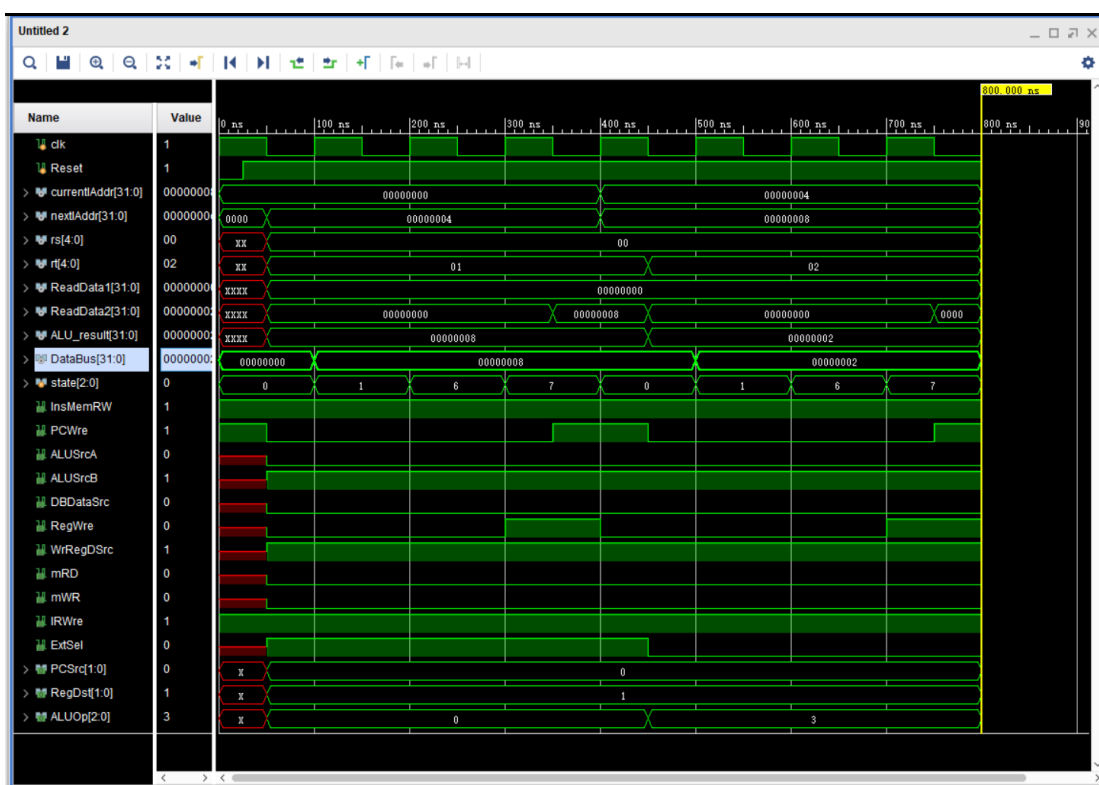


图6-1 指令1~2的波形图，0~800ns

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

如图6-1，0~400ns，共花费4个时钟周期。当前PC地址（即currentIAddr）为0x00000000。

IF (000) 状态，0~100ns。时钟下降沿到来时，IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态，100~200ns。rs为0号寄存器，ReadData1读出的值为0，被送入ADR的输入端。

EXE (110) 状态，200~300ns。ADR的输出改变为0。ALUSrcA为0，ALUSrcB为1，ALUOp为000，ExtSel为1，因此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。由于DBDataSrc为0，因此ALU的结果被送入DBDR的输入端。

WB (111) 状态，300~400ns。DBDR的输出改变为8。由于RegDst为01、WrRegDSrc为1，因此在时钟下降沿到来时，数据8被写入1号寄存器。除此之外，在时钟下降沿到来时，PCWre信号变为1，PCSrc为00，下一条指令将顺序执行。

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

如图6-1, 400~800ns, 共花费4个时钟周期。当前PC地址为0x00000004。

IF (000) 状态, 400~500ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 500~600ns。rs为0号寄存器, ReadData1读出的值为0, 被送入ADR的输入端。

EXE (110) 状态, 600~700ns。ADR的输出改变为0。ALUSrcA为0, ALUSrcB为1, ALUOp为011, ExtSel为0, 因此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 $0|2=2$ 。由于DBDataSrc为0, 因此ALU的结果被送入DBDR的输入端。

WB (111) 状态, 700~800ns。DBDR的输出改变为2。由于RegDst为01、WrRegDSrc为1, 因此在时钟下降沿到来时, 数据2被写入2号寄存器。除此之外, 在时钟下降沿到来时, PCWre信号变为1, PCSrc为00, 下一条指令将顺序执行。

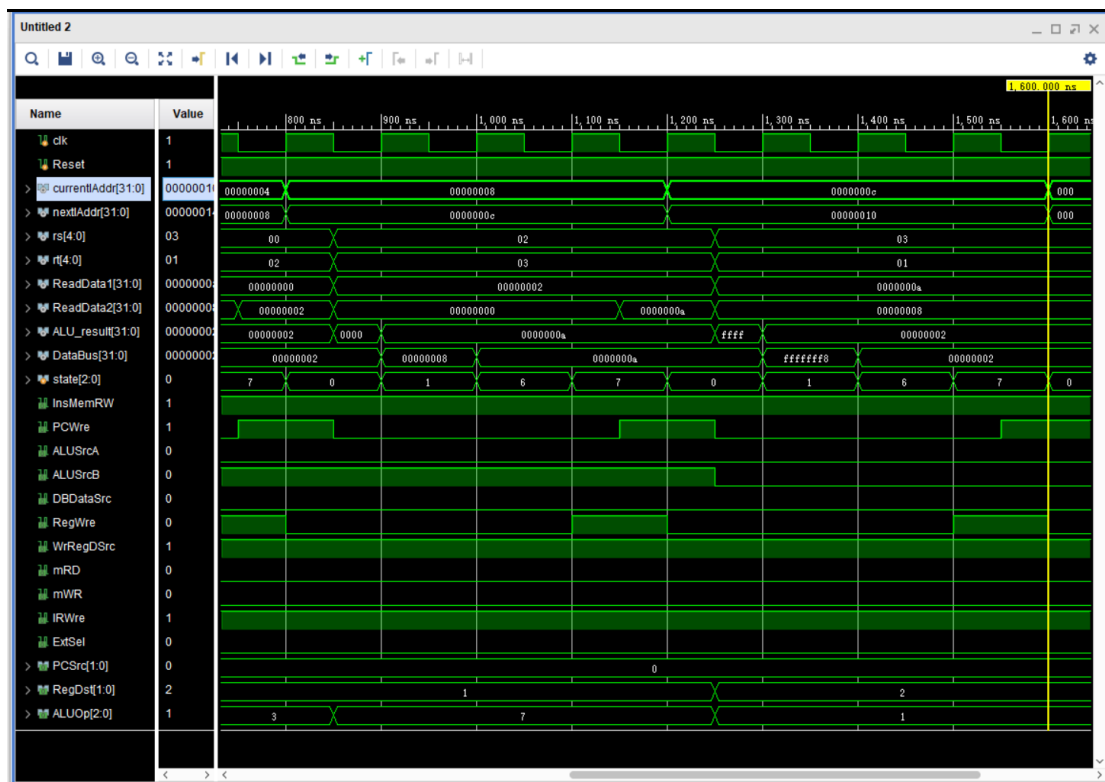


图6-2 指令3~4的波形图, 800~1600ns

3. xori \$3,\$2,8

如图6-2, 800~1200ns, 共花费4个时钟周期。当前PC地址为0x00000008。

IF (000) 状态, 800~900ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 900~1000ns。rs为2号寄存器, ReadData1读出的值为2, 被送入ADR的输入端。

EXE (110) 状态, 1000~1100ns。ADR的输出改变为2。ALUSrcA为0, ALUSrcB为1, ALUOp为111, ExtSel为0, 因此ALU将**rs寄存器的值与零扩展后的立即数做异或运算**, 得到结果为 $2^8=10$ 。由于DBDataSrc为0, 因此ALU的结果被送入DBDR的输入端。

WB (111) 状态, 1100~1200ns。DBDR的输出改变为10。由于RegDst为01、WrRegDSrc为1, 因此在时钟下降沿到来时, 数据10被写入3号寄存器。除此之外, 在时钟下降沿到来时, PCWre信号变为1, PCSrc为00, 下一条指令将顺序执行。

4. sub \$4,\$3,\$1

如图6-2, 1200~1600ns, 共花费4个时钟周期。当前PC地址为0x0000000c。

IF (000) 状态, 1200~1300ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 1300~1400ns。rs为3号寄存器, ReadData1读出的值为10, 被送入ADR的输入端; rt为1号寄存器, ReadData2读出的值为8, 被送入BDR的输入端。

EXE (110) 状态, 1400~1500ns。ADR的输出改变为10, BDR的输入变为8。ALUSrcA为0, ALUSrcB为0, ALUOp为001, 因此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 $10-8=2$ 。由于DBDataSrc为0, 因此ALU的结果被送入DBDR的输入端。

WB (111) 状态, 1500~1600ns。DBDR的输出改变为2。由于RegDst为10、WrRegDSrc为1, 因此在时钟下降沿到来时, 数据2被写入4号寄存器。除此之外, 在时钟下降沿到来时, PCWre信号变为1, PCSrc为00, 下一条指令将顺

序执行。

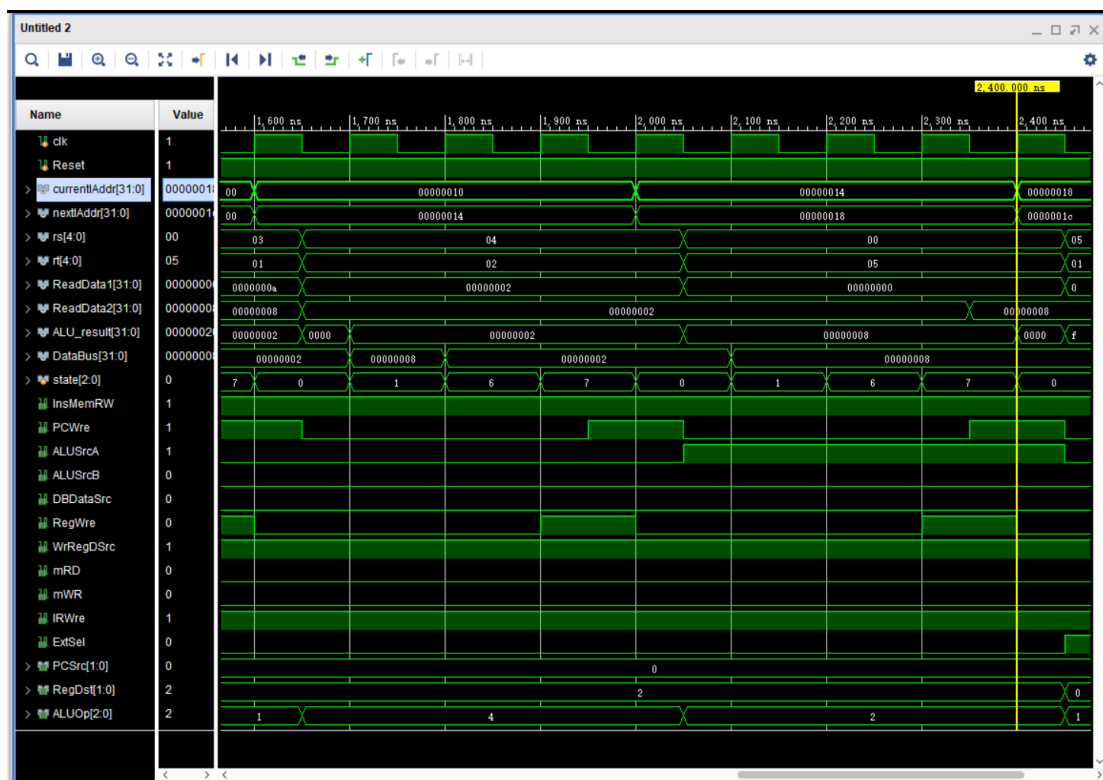


图6-3 指令5~6的波形图，1600~2400ns

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

如图6-3, 1600~2400ns, 共花费4个时钟周期。当前PC地址为0x00000010。

IF (000) 状态, 1600~1700ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 1700~1800ns。rs为4号寄存器, ReadData1读出的值为2, 被送入ADR的输入端; rt为2号寄存器, ReadData2读出的值为2, 被送入BDR的输入端。

EXE (110) 状态, 1800~1900ns。ADR的输出改变为2, BDR的输入变为2。ALUSrcA为0, ALUSrcB为0, ALUOp为010, 因此ALU将rs寄存器的值与rt寄存器的值做逻辑与运算, 得到结果为2&2=2。由于DBDataSrc为0, 因此ALU的结果被送入DBDR的输入端。

WB (111) 状态, 1900~2000ns。DBDR的输出改变为2。由于RegDst为10, WrRegDSrc为1, 因此在时钟下降沿到来时, 数据2被写入5号寄存器。除此

之外，在时钟下降沿到来时，PCWre信号变为1，PCSrc为00，下一条指令将顺序执行。

6. sll \$5,\$5,2

如图6-3,2000~2100ns,共花费4个时钟周期。当前PC地址为0x00000014。

IF (000) 状态, 1600~1700ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 2100~2200ns。rt为5号寄存器, ReadData2读出的值为2, 被送入BDR的输入端。

EXE (110) 状态, 2200~2300ns。BDR的输入变为2。ALUSrcA为1, ALUSrcB为0, ALUOp为010, 因此ALU将sa与rt寄存器的值做移位运算, 得到结果为 $2 \ll 2 = 8$ 。由于DBDataSrc为0, 因此ALU的结果被送入DBDR的输入端。

WB (111) 状态, 2300~2400ns。DBDR的输出改变为8。由于RegDst为10、WrRegDSrc为1, 因此在时钟下降沿到来时, 数据8被写入5号寄存器。除此之外, 在时钟下降沿到来时, PCWre信号变为1, PCSrc为00, 下一条指令将顺序执行。

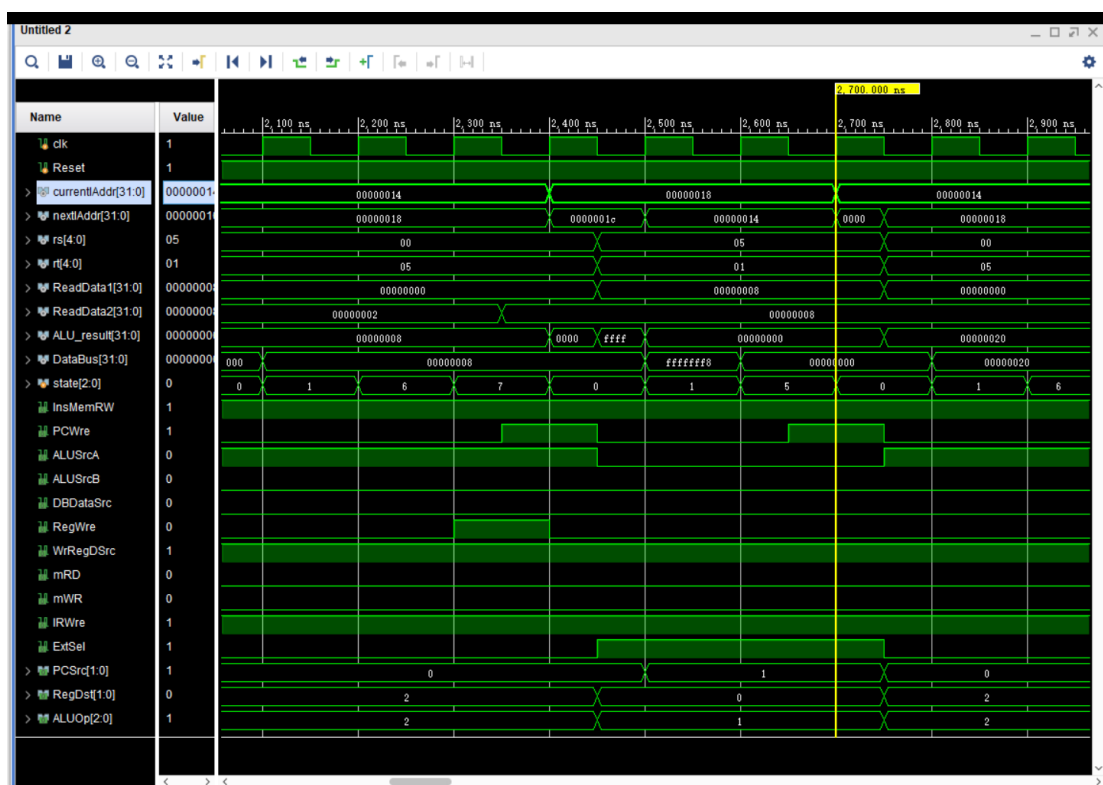


图6-4 指令7的波形图, 1600~2400ns

7. beq \$5,\$1,-2

如图6-4, 2400~2700ns, 共花费3个时钟周期。当前PC地址为0x00000018。

IF (000) 状态, 2400~2500ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 2500~2600ns。rs为5号寄存器, ReadData1读出的值为8, 被送入ADR的输入端; rt为1号寄存器, ReadData2读出的值为8, 被送入BDR的输入端。

EXE (101) 状态, 2600~2700ns。ADR的输入变为8, BDR的输入变为8。ALUSrcA为0, ALUSrcB为0, ALUOp为001, 因此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 $8-8=0$, 因此zero输出为1, 即满足跳转条件。PCSrc变为01, ExtSel的值为1, 因此立即数被符号扩展并左移两位再与当前PC相加, 作为下一PC地址。除此之外, 在时钟下降沿到来时, PCWre信号变为1, 下一条指令将跳转到0x00000014。

从波形图便可看出, 执行完beq指令以后, 跳转到0x00000014的sll指令执行。

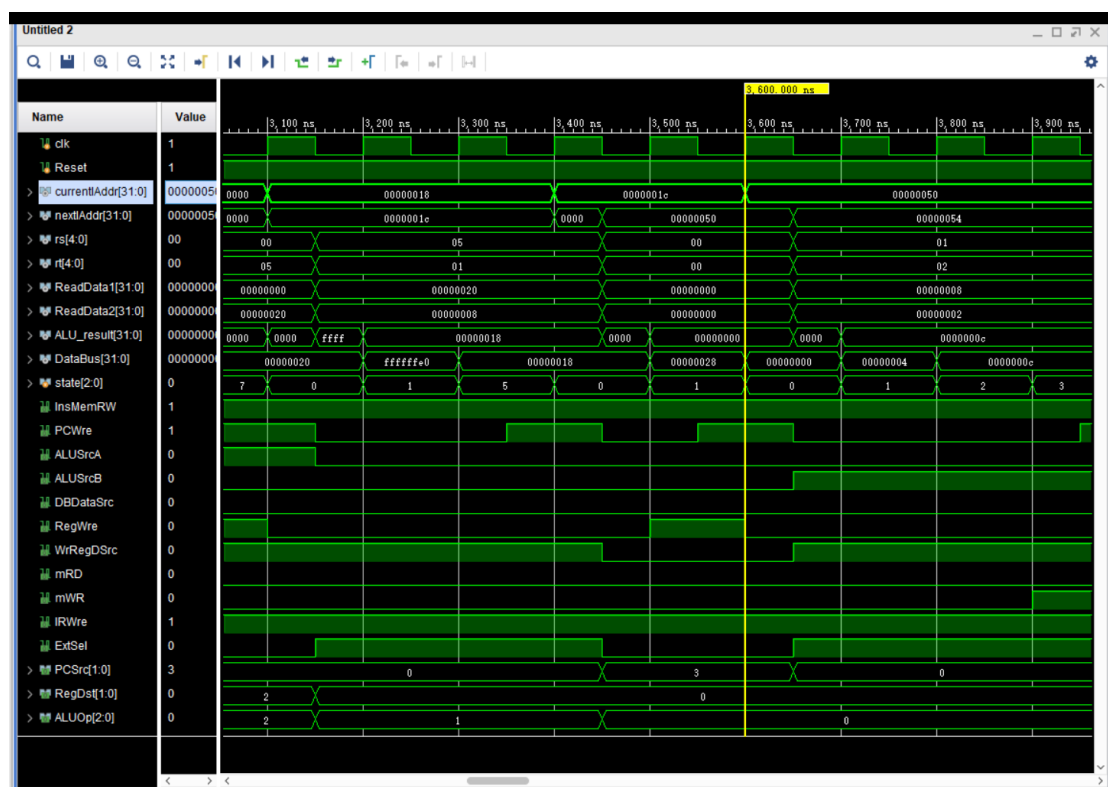


图6-5 指令8的波形图, 3400~3600ns

8. jal 0x00000050

如图6-5, 3400~3600ns, 共花费2个时钟周期。当前PC地址为0x0000001c。

IF (000) 状态, 3400~3500ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 3500~3600ns。WrRegDSrc变为0, RegWre信号变为1, RegDst为00, 因此在时钟下降沿到来时, PC+4的值被写入31号寄存器中。又因为PCSrc为11, 因此下一PC地址为{PC4[31:28], addr, 00}。在时钟下降沿到来时, PCWre信号变为1, 下一条指令将跳转到0x00000050。

从波形图便可看出, 执行完jal指令以后, 跳转到0x00000050执行。

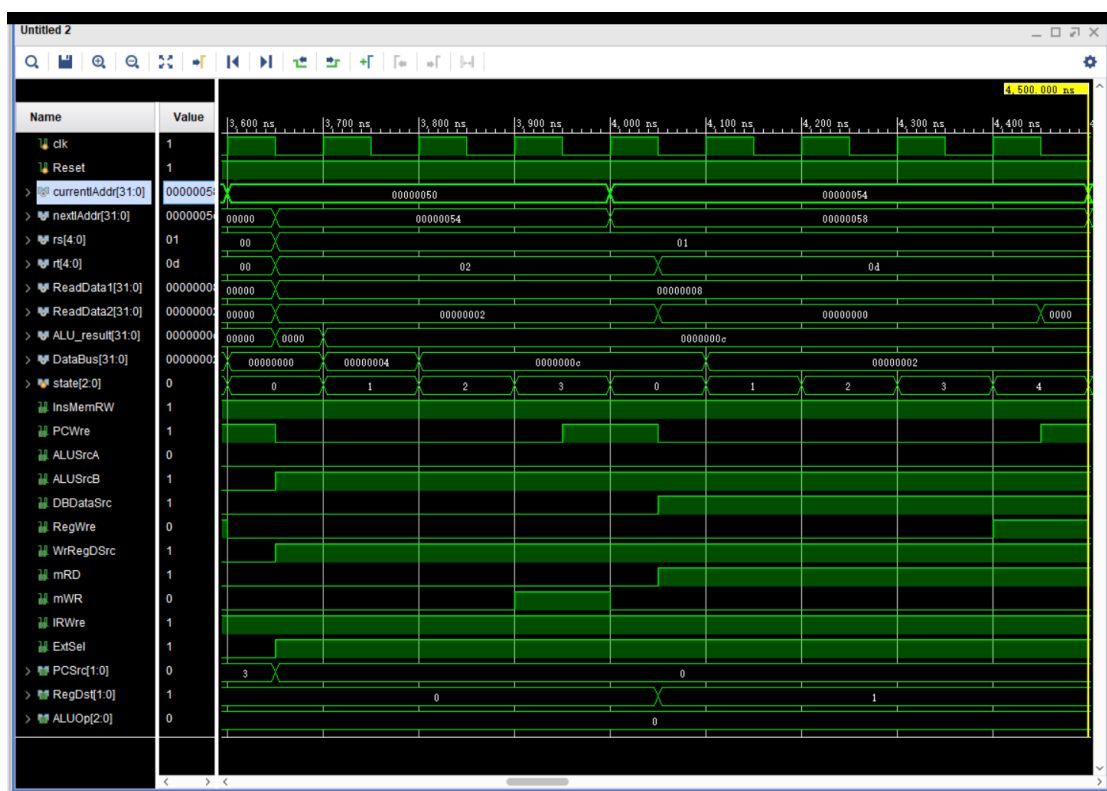


图6-6 指令9~10的波形图, 3600~4500ns

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

如图6-6, 3600~4000ns, 共花费4个时钟周期, 当前PC为0x00000050。

IF (000) 状态, 3600~3700ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 3700~3800ns。rs为1号寄存器, ReadData1读出的值为8, 被送入ADR的输入端; rt为2号寄存器, ReadData2读出的值为2, 被送入BDR

的输入端。

EXE (010) 状态, 3800~3900ns。ADR的输出改变为8, BDR的输出改变为2。ALUSrcA为0, ALUSrcB为1, ALUOp为000, ExtSel为1, 因此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 $8+4=12$, ALU的这个结果被送入ALUoutDR的输入端。

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端。mWR信号为1, 即存储器写使能有效, 因此BDR的输出2被写入存储器中地址为12的位置。除此之外, 在时钟下降沿到来时, PCWre信号变为1, PCSrc为00, 下一条指令将顺序执行。

10. lw \$t3,4(\$t1)

如图6-6, 4000~4500ns, 共花费5个时钟周期, 当前PC为0x00000054。

IF (000) 状态, 4000~4100ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 4100~4200ns。rs为1号寄存器, ReadData1读出的值为8, 被送入ADR的输入端。

EXE (010) 状态, 4200~4300ns。ADR的输出改变为8。ALUSrcA为0, ALUSrcB为1, ALUOp为000, ExtSel为1, 因此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 $8+4=12$, ALU的这个结果被送入ALUoutDR的输入端。

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端。mRD信号为1, 即存储器读使能有效, 因此数据存储器中地址为12的内容(数据2)被输出到DataOut端口。又因为DBDataSrc为1, 因此DataOut被送入DBDR的输入端。

WB (100) 状态, 4400~4500ns。DBDR的输出改变为2。RegDst为01, WrRegDSrc为1, RegWre为1, 因此在时钟下降沿到来时, 数据总线上的数据被写入13号寄存器。除此之外, 在时钟下降沿到来时, PCWre信号变为1, PCSrc为00, 下一条指令将顺序执行。

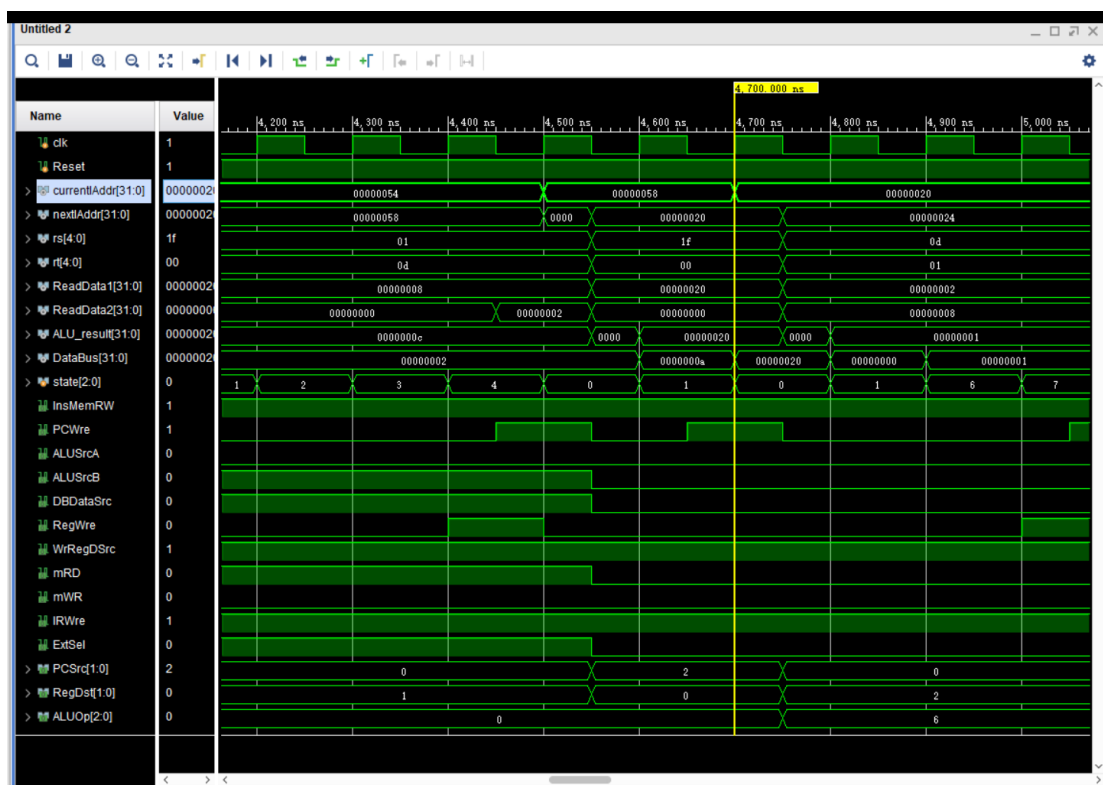


图6-7 指令11的波形图，4500~4700ns

11. jr \$31

如图6-7, 4500~4700ns, 共花费2个时钟周期。当前PC地址为0x00000058。

IF (000) 状态, 4500~4600ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 4600~4700ns。rs为31号寄存器, ReadData1的值为0x20 (这是当时执行jal时写入进去的), 该值直接送入由PCSrc操控的哪个数据选择器的2号输入端。PCSrc的值为10, 因此下一PC地址为0x00000020。PCWre信号变为1后, 下一条执行的指令就是0x00000020处的指令了。

从波形图便可看出, 执行完jr指令以后, 确实跳转到0x00000020执行。

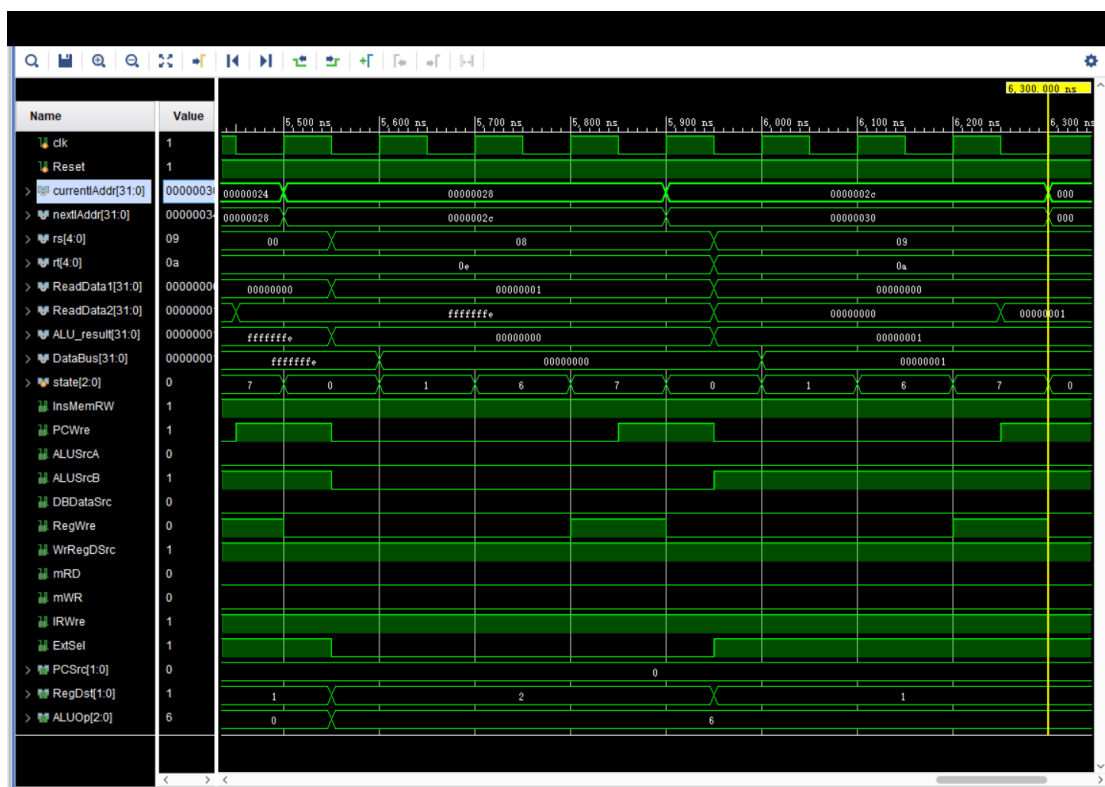


图6-8 指令12~13的波形图, 5500~6300ns

12. slt \$9,\$8,\$14

如图6-8, 5500~5900ns, 共花费4个时钟周期。当前PC地址为0x00000028。

IF (000) 状态, 5500~5600ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 5600~5700ns。rs为8号寄存器, ReadData1读出的值为1, 被送入ADR的输入端; rt为14号寄存器, ReadData2读出的值为-2, 被送入BDR的输入端。

EXE (110) 状态, 5700~5800ns。ADR的输出改变为1, BDR的输入变为-2。ALUSrcA为0, ALUSrcB为0, ALUOp为110, 因此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 $1 < -2 = 0$ 。由于DBDataSrc为0, 因此ALU的结果被送入DBDR的输入端。

WB (111) 状态, 5800~5900ns。DBDR的输出改变为0。由于RegDst为10、WrRegDSrc为1, 因此在时钟下降沿到来时, 数据0被写入9号寄存器。除此之外, 在时钟下降沿到来时, PCWre信号变为1, PCSrc为00, 下一条指令将顺序执行。

13. slti \$10,\$9,2

如图6-8, 5900~6300ns, 共花费4个时钟周期。当前PC地址为0x0000002c。

IF (000) 状态, 5900~6000ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 6000~6100ns。rs为9号寄存器, ReadData1读出的值为0, 被送入ADR的输入端。

EXE (110) 状态, 6100~6200ns。ADR的输出改变为0。ALUSrcA为0, ALUSrcB为1, ALUOp为110, ExtSel为1, 因此ALU将rs寄存器的值与符号扩展后的立即数做带符号比较运算, 得到结果为 $0 < 2 = 1$ 。由于DBDataSrc为0, 因此ALU的结果被送入DBDR的输入端。

WB (111) 状态, 6200~6300ns。DBDR的输出改变为1。由于RegDst为01、WrRegDSrc为1, 因此在时钟下降沿到来时, 数据1被写入10号寄存器。除此之外, 在时钟下降沿到来时, PCWre信号变为1, PCSrc为00, 下一条指令将顺序执行。

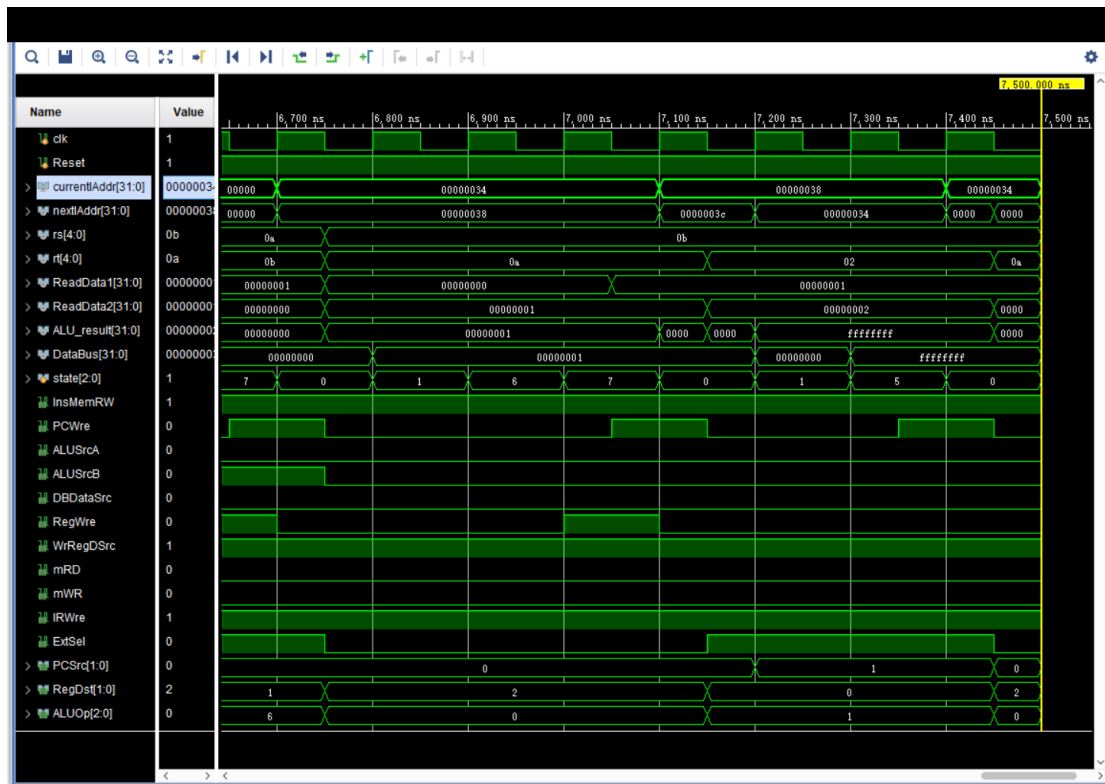


图6-9 指令14~15的波形图, 6700~7400ns

14. add \$11,\$11,\$10

如图6-9,6700~7100ns,共花费4个时钟周期。当前PC地址为0x00000034。

IF (000) 状态, 6700~6800ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 6800~6900ns。rs为11号寄存器, ReadData1读出的值为0, 被送入ADR的输入端; rt为10号寄存器, ReadData2读出的值为1, 被送入BDR的输入端。

EXE (110) 状态, 6900~7000ns。ADR的输出改变为0, BDR的输入变为1。ALUSrcA为0, ALUSrcB为0, ALUOp为000, 因此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 $0+1=1$ 。由于DBDataSrc为0, 因此ALU的结果被送入DBDR的输入端。

WB (111) 状态, 7000~7100ns。DBDR的输出改变为1。由于RegDst为10、WrRegDSrc为1, 因此在时钟下降沿到来时, 数据1被写入11号寄存器。除此之外, 在时钟下降沿到来时, PCWre信号变为1, PCSrc为00, 下一条指令将顺序执行。

15. bne \$11,\$2,-2

如图6-9,7100~7400ns,共花费3个时钟周期。当前PC地址为0x00000038。

IF (000) 状态, 7100~7200ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 7200~7300ns。rs为11号寄存器, ReadData1读出的值为1, 被送入ADR的输入端; rt为2号寄存器, ReadData2读出的值为2, 被送入BDR的输入端。

EXE (101) 状态, 7300~7400ns。ADR的输入变为1, BDR的输入变为2。ALUSrcA为0, ALUSrcB为0, ALUOp为001, 因此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 $1-2=-1$, 因此zero输出为0, 即满足跳转条件。PCSrc变为01, ExtSel的值为1, 因此立即数被符号扩展并左移两位再与当前PC相加, 作为下一PC地址。除此之外, 在时钟下降沿到来时, PCWre信号变为1, 下一条指令将跳转到0x00000034。

从波形图便可看出, 执行完bne指令以后, 跳转到0x00000034的add指令执

行。

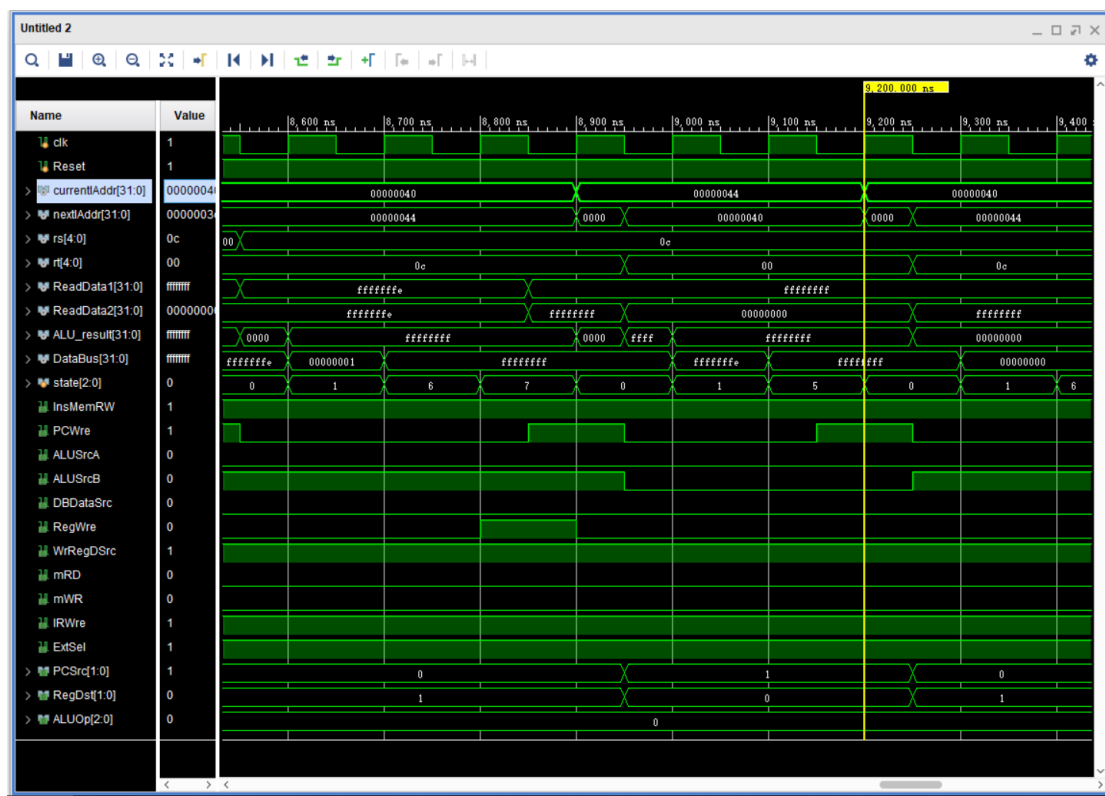


图6-10 指令16的波形图, 8900~9200ns

16. bltz \$12, -2

如图6-10, 8900~9200ns, 共花费3个时钟周期。当前PC地址为0x00000044。

IF (000) 状态, 8900~9000ns。时钟下降沿到来时, IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态, 9000~9100ns。rs为12号寄存器, ReadData1读出的值为-1, 被送入ADR的输入端。虽然这里看似没用到rt, 但根据该指令的行为, 可以认为rt为0号寄存器, ReadData2读出的值为0, 被送入BDR的输入端。

EXE (101) 状态, 9100~9200ns。ADR的输出变为-1, BDR的输出变为0。ALUSrcA为0, ALUSrcB为0, ALUOp为000, 因此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-1+0=-1, 因此sign输出为1, 即满足跳转条件。PCSrc变为01, ExtSel的值为1, 因此立即数被符号扩展并左移两位再与当前PC相加, 作为下一PC地址。除此之外, 在时钟下降沿到来时, PCWre信号变为1, 下一条指令将跳转到0x00000040。

从波形图便可看出，执行完bne指令以后，跳转到0x00000040的addiu指令执行。

17. andi \$t2,\$t2,2

这是一条非常普通的I型指令，指令行为分析没有必要再赘述了。

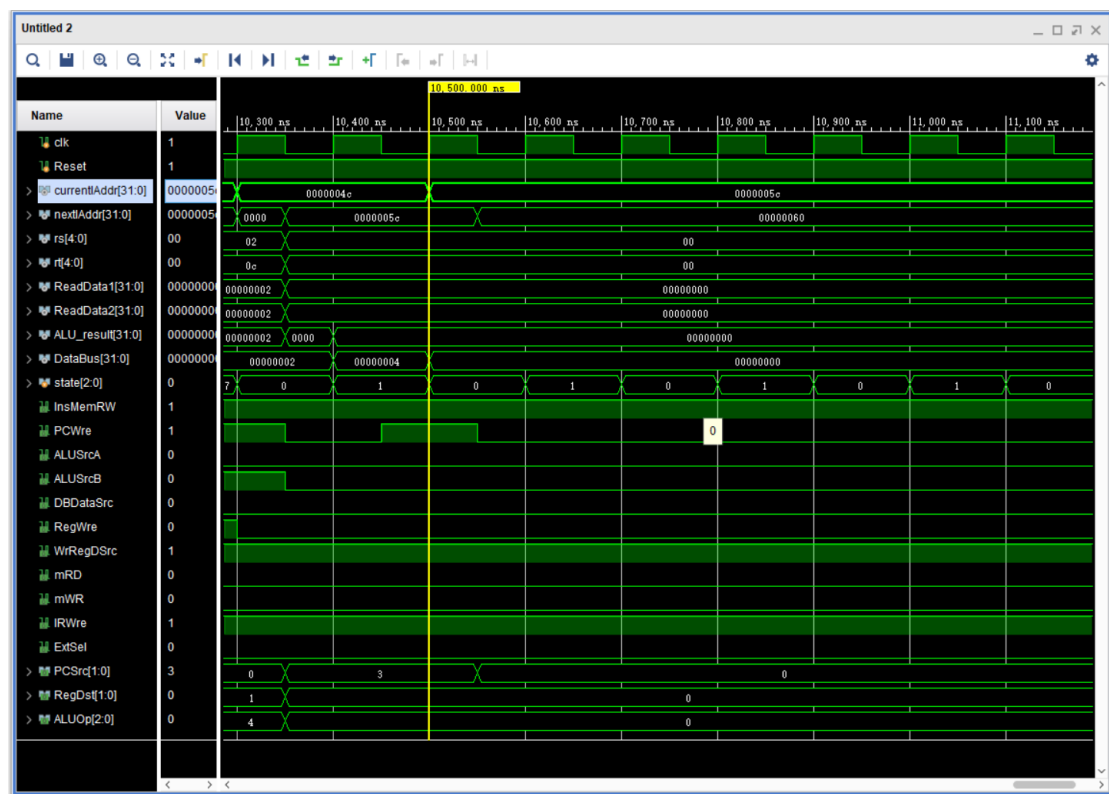


图6-11 指令18~19的波形图，10300ns以后

18. j 0x0000005C

如图6-11，10300~10500ns，共花费2个时钟周期。当前PC地址为0x0000004c。

IF (000) 状态，10300~10400ns。时钟下降沿到来时，IR寄存器的输出变为当前执行的指令机器码。

ID (001) 状态，10400~10500ns。PCSrc为11，因此下一PC地址为{PC4[31:28], addr, 00}。在时钟下降沿到来时，PCWre信号变为1，下一条指令将跳转到0x0000005c。

从波形图便可看出，执行完jal指令以后，跳转到0x0000005c执行。

19. halt

如图6-11, 10500ns以后为halt指令, 当前PC为0x0000005c。此时PCWre信号永远保持0, 因此PC不能递增, 此后所有数值、控制信号都保持不变, 系统停机。

D. 在Basys3实验板上实现

● 思想方法

成功仿真duo 周期CPU后, 在Basys3实验板上实现就不困难了。

根据实验要求, 要在板上的四个七段数码管上显示当前PC、下一PC、rs地址、rs数据、rt地址、rt数据、ALU结果以及DB总线的数据。而要显示的数据由CPU提供, 因此需要先写出显示七段数码管的模块。

最后, 将封装好的CPU顶层模块和新写好的显示模块再次封装, 就可以综合、实现、烧板了。

● 操作方法

1. 四位计数器

需要一个四位计数器来扫描四个七段数码管。代码很简单, 已略去, 见文件Counter4.v。

2. 时钟分频器

Basys3板载时钟频率为100MHz, 需要将其分频为1000Hz的较为合适:

```
module clk_div(
    input clk,
    output reg clk_sys = 0
);

    reg [25:0] div_counter = 0;
    always @(posedge clk) begin
        if(div_counter >= 50000) begin
            clk_sys <= ~clk_sys;
            div_counter <= 0;
        end
        else div_counter <= div_counter + 1;
    end
endmodule
```

3. 扫描显示四个数字需要用到四位计数器:

```
module Counter4(
```

```

input clk,
output reg [1:0] count
);
always @(posedge clk) begin
    if(count == 2'b11) count <= 0;
    else count <= count + 1;
end
endmodule

```

4. 十六进制数到七段数码管的译码器

将模块命名为Hex_To_7Seg，设计为组合逻辑电路，引脚说明：

hex，输入一个4位数字

dispcode，输出七段数码管显示信号

使用case语句选择hex即可，代码冗长，已省略。

5. 4位四选一数据选择器

由于Basys3板一次只能显示一个数字，所以有必要配合数据选择器来实现扫描显示四个数字。数据选择器的代码关键部分如下：

```

always @(choice or in0 or in1 or in2 or in3) begin
    case(choice)
        2'b00: out = in0;
        2'b01: out = in1;
        2'b10: out = in2;
        2'b11: out = in3;
        default: out = 0;
    endcase
end

```

6. 扫描显示四个数字的模块

将显示模块命名为Four_LED，引脚说明：

clk，扫描显示用的时钟

reset，复位

hex0、hex1、hex2、hex3，分别对应从左到右的四个数码管上显示的数字

enable，扫描使能信号

dispcode，七段数码管显示信号

这事实上是前面几个底层模块的顶层模块，按照如下示意图连接即可：

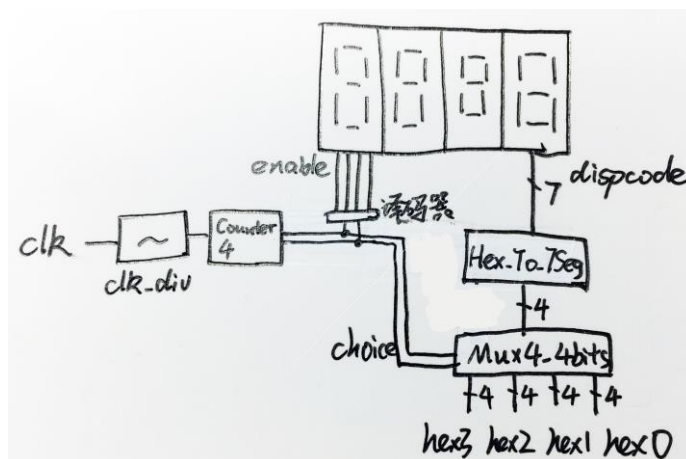


图7

7. 按键消抖

由于硬件原因，按板上的按钮时提供的电平变化可能不是单一的、稳定的上升沿或下降沿，因此需要对按钮进行消抖，其原理是按下按钮后延迟20ms才接受下一次按下。

代码参考了互联网上的，如下：

```
module Button_Debounce(
    input clk,
    input btn_in,
    output btn_out
);
    reg [2:0] btn=0;
    wire clk_20ms;
    ClockDivisor #(1000000) t_20ms(clk,clk_20ms);
    always @(posedge clk_20ms) begin
        btn[0]<=btn_in;
        btn[1]<=btn[0];
        btn[2]<=btn[1];
    end
    assign btn_out=(btn[2]&btn[1]&btn[0])|(~btn[2]&btn[1]&btn[0]);
endmodule
```

8. 烧板顶层模块 (Basys3_CPU)

该模块封装top_CPU（就是仿真用的那个顶层模块）和Four_LED（用于显示四个数字的模块）。

注意到“CPU烧板时，Basys3板的使用说明”文档中的要求如下：

开关SW_in (SW15、SW14) 状态情况如下。显示格式： 左边两位数码管BB：右边两位数码管BB。 以下是数码管的显示内容。

SW_in = 00 : 显示当前PC值：下条指令PC值

SW_in = 01 : 显示RS寄存器地址:RS寄存器数据

SW_in = 10 : 显示RT寄存器地址:RT寄存器数据

SW_in = 11 : 显示ALU结果输出:DB总线数据。

复位信号 (reset) 接开关SW0 , 按键 (单脉冲) 接按键BTNR。

于是很容易写出Basys3_CPU的代码, 其关键部分如下, 重点部分已标出:

```
Four_LED Four_LED(
    .clock(basys3_clock),
    .reset(reset_sw),
    .hex3(DisplayData[15:12]),
    .hex2(DisplayData[11:8]),
    .hex1(DisplayData[7:4]),
    .hex0(DisplayData[3:0]),
    .enable(enable),
    .dispcode(dispcode)
);
Mux4_16bits Mux4_16bits(
    .choice(SW_in),
    .in0({currentIAddr[7:0], nextIAddr[7:0]}),
    .in1({3'b000, rs, ReadData1[7:0]}),
    .in2({3'b000, rt, ReadData2[7:0]}),
    .in3({ALU_result[7:0], DataBus[7:0]}),
    .out(DisplayData)
);
```

最后的最后, 整个工程的结构如下图所示:

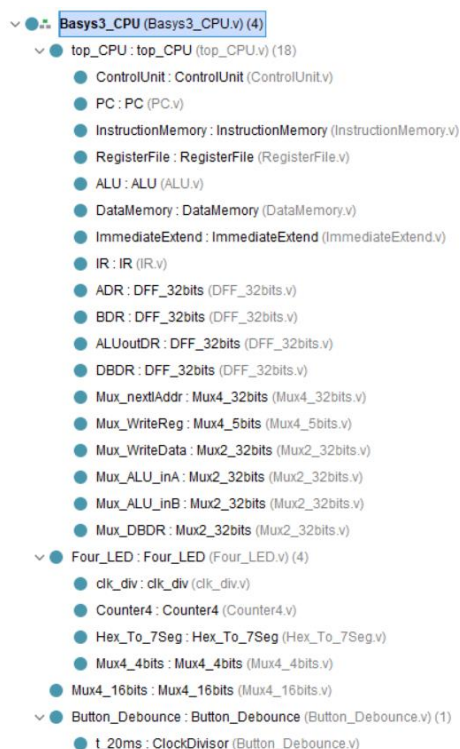
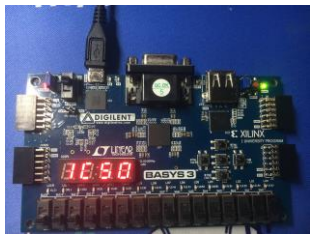
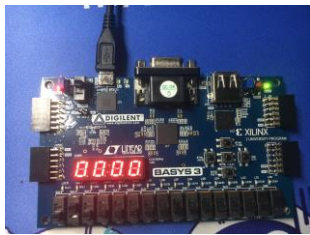
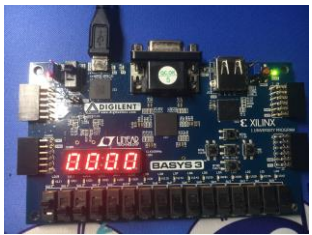
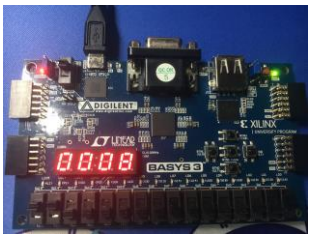


图8 完整工程结构树

● 检验结果

考虑到指令过多, 且每条指令都包含多个状态, 每个状态对应到Basys3板上

的显示都有4个，如果附上全部照片会造成篇幅过长。这里只选取一条有代表性的跳转指令作为示例。下面显示的是执行0x0000001C处的“jal 0x00000050”（J型指令，没有用到ALU）：

当前PC:下一PC	rs地址:rs数据	rt地址:rt数据	ALU结果:DB总线
			

更多指令已略去。

六. 实验心得

1. 多周期CPU设计与实现与单周期CPU设计与实现这两个实验有非常多的相似之处——InstructionMemory、RegisterFile、DataMemory、ALU等底层模块是完全相同的。因此这些模块可以直接使用在实验二中已经实现完成的代码，大大简化了实验步骤。

2. 从数据通路图可以看出，ControlUnit模块需要重写，此外还增加了多周期CPU特有的IR、ADR、BDR、DBDR和ALUoutDR这几个寄存器。IR、ADR、BDR、DBDR和ALUoutDR都可以看做是D触发器，其实现代码比较容易实现，而且它们几乎是相同的，因此可以写一份代码，然后实例化为多个模块即可。

3. 多周期CPU的ControlUnit新增加了时钟作为输入，这是因为ControlUnit内部要实现状态转移的功能。容易想到，ControlUnit主要完成两件事——一是状态转移，根据当前指令opcode以及当前状态推断出下一个状态；二是输出函数，也就是根据当前指令opcode和当前所处状态生成各种控制信号来控制CPU中的其他部件。根据“实验原理 - 图3”来实现这两部分功能即可。总的来说，ControlUnit的思路还是很清晰的，只不过写起代码来稍微麻烦了一些。

4. 在Vivado中进行仿真时，我发现了一个之前没有在意的细节。在仿真波形窗口的左侧有Sources和Object窗口，内部可以显示光标处各个变量的值。需要注意的是，这个地方显示的值是光标“之后”的值而不是之前的。比如光标恰好处于时钟上升沿处，那么这里显示的clk的值就是1而不是0。其他变量同样如此，在变量发生改变的地方显示的是变量改变

后的值而不是原来的值。

5. 关于bltz指令的小技巧。bltz指令格式为“bltz rs, immediate”，它只有一个寄存器操作数，也就是rs。考虑到指令中的无关位均为0，我的想法是把它当作普通的I型指令，也就是说rt就是0号寄存器。再将ALU设置为加法，所以ALU的运算结果就是(rs)+0=(rs)，因此ALU的sign输出就是rs寄存器中值的符号位！若符号为1，则代表它为负数，bltz的结果为真；反之为假。这样能够处理bltz这一特殊格式的指令。

6. 关于IR寄存器的IRWre信号。我认为在本次实验中，IRWre这一信号并没有实际的作用（事实上在我的实现中，我令该信号恒为高电平1），这是因为IR寄存器的输入来自InstructionMemory的输出，而InstructionMemory间接地受到PCWre信号的控制（注意它是组合逻辑），当PCWre无效时，InstructionMemory的输出值便不会改变，因此即使IRWre信号为1，IR的输出也不会改变。

7. 编写了适用于本次实验的汇编器，实现将MIPS汇编语言翻译为指令机器码。因为我使用Python编写，因此也达到了巩固Python编程的目的。

8. 关于Verilog的Debug经验。首次写完程序运行的时候，出现bug是非常正常的现象，而Verilog语言与其他高级语言（如C/C++等）不太一样，后者的程序是以顺序执行为框架的，而Verilog语言具有显著的“模块化”特征，因此其Debug方法也略有不同，不过主要思想还是一致的。根据我的Debug经历，我认为把握住硬件电路本身是非常重要的。在此次实验中，就是要熟悉数据通路图。当仿真时发现某处数据不对，那么就要在数据通路图中找到该数据引脚，先在直接相关的模块中排查错误，此时该模块中必定有至少一个错误的数数据（可能是输入，也可能是局部变量），如果是一个输入信号错了，那么再顺着该输入信号去研究相关的另一模块……主要就是一种“顺藤摸瓜”的思想。

9. Vivado对于Verilog的宽容度很高，有许多错误并不会报Error，这时候就需要程序员异常小心了。最常见的错误应该是变量的位宽不对应，如RegDst信号。RegDst在“单周期CPU设计与实现”实验中是1位宽的，而在“多周期CPU设计与实现”实验中是2位宽的。最开始我直接复制了单周期CPU中的部分代码，就导致了这种位宽不对应的错误。不过，采取上面第8条“顺藤摸瓜”的Debug思想，我还是较为顺利地找出了错误并予以改正。

10. 与单周期类似地，首次烧板时，Implementation会报错，显示关于按钮的一个错误（错误代码：30-574）。解决办法是在约束文件（Basys3_CPU.xdc）中添加如下代码：

```
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets next_button]
```

七. 期末课程总结

1. 第一个实验是MIPS汇编实验。在该实验中，我将计算机组成原理理论课上学到的MIPS指令在PCSpim跑了起来，写出了第一个MIPS汇编程序（我当时写的是冒泡排序）。此外，在此次实验中还深刻理解了大端序与小端序的区别，并学会了如何在设计程序时考虑端序问题。除此之外，我还了解到了MIPS指令和伪指令的区别，以及它们对于编写汇编程序的作用。

2. 在实验二和实验三这两次CPU设计实验中，我学习并使用了Verilog HDL硬件描述语言，知道了Verilog是一种“模块化”的语言，还了解了该语言的基本语法以及一些操作细节（如阻塞赋值和非阻塞赋值等）。从完全不会这门语言，到能够用该语言实现单周期和多周期CPU。

3. 通过一学期的实验，我更加深入地理解了单周期和多周期CPU的工作原理，理解了ALU等底层模块的功能及其实现、数据通路的构建、各种寄存器的作用以及数据选择器在数据通路中的重要作用，还通过实例更清晰地认识了状态机的功能及其实现。总的来说，我对计算机内部组成（主要是CPU的组成）有了更加全面和深入的认识。

4. “单周期CPU设计与实现”和“多周期CPU设计与实现”这两个实验十分复杂，内容繁多，要求严格，涉及的知识也很多，完成起来有难度。通过这两次的实验，我学会了稳定心态，冷静地写代码、找错误，也知道了完成这种复杂任务一定要有耐心，不能乱了阵脚。此外还要有信心，既然老师布置给我们这些实验，它们就是一定可以被完成的。只要愿意投入就一定会有回报。

5. Verilog语言的调试与以往学的C/C++、Python高级程序设计语言的调试方法不同。在其他语言中，调试方法往往是观察特定的变量在函数中的变化情况，以及分支、循环条件的判断等。而在Verilog语言中有所不同。根据我自己的经验，我认为调试Verilog要遵从该语言的“模块化”这一特性：当发现仿真结果的某个信号不正确的时候，首先应该检查的是该信号直接出现的底层模块——是不是模块逻辑写错了？是不是有特殊情况没有考虑到？其次考虑底层模块更高一层的模块——是不是底层模块实例化时不小心遗漏了某个引脚？是不是搞混了相似的变量名称？如果还是没有找出错误，继续按照这样的步骤检查更高层的模块，直到顶层模块。根据这一学期的实验，我遇到的所有错误几乎全部都可以用这种方法来找到并改正。

6. 这学期的计算机组成原理理论课和实验课让我对计算机（主要是CPU）的认识深入

了很多，计算机硬件不再那么神秘。相信在以后的课程学习中，甚至是更未来的科研实习和工作中都会用到我在计组实验课上得到的知识或方法。

附录A: 测试程序段

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	addiu \$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 000 0000 0000	=	04612000
0x00000010	and \$5,\$4,\$2	010000	00100	00010	00101 000 0000 0000	=	40822800
0x00000014	sll \$5,\$5,2	011000	00000	00101	00101 000 1000 0000	=	60052880
0x00000018	beq \$5,\$1,-2(=,转 14)	110100	00101	00001	1111 1111 1111 1110	=	D0A1FFFE
0x0000001C	jal 0x00000050	111010	00000	00000	0000 0000 0001 0100	=	E8000014
0x00000020	slt \$8,\$13,\$1	100111	01101	00001	01000 000 0000 0000	=	9DA14000
0x00000024	addiu \$14,\$0,-2	000010	00000	01110	1111 1111 1111 1110	=	080EFFFF
0x00000028	slt \$9,\$8,\$14	100111	01000	01110	01001 000 0000 0000	=	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 000 0000 0000	=	016A5800
0x00000038	bne \$11,\$2,-2 (≠,转 34)	110101	01011	00010	1111 1111 1111 1110	=	D562FFFE
0x0000003C	addiu \$12,\$0,-2	000010	00000	01100	1111 1111 1111 1110	=	080CFFFF
0x00000040	addiu \$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
0x00000048	andi \$12,\$2,2	010001	00010	01100	0000 0000 0000 0010	=	444C0002
0x0000004C	j 0x0000005C	111000	00000	00000	0000 0000 0001 0111	=	E0000017
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
0x0000005C	halt	111111	00000	00000	0000 0000 0000 0000	=	FC000000

附录B: 汇编器代码 (Python 3)

```

import re
def reg_to_num(reg):
    regid = int(reg.lstrip('$'))
    result = bin(regid)[2:].rjust(5, '0')
    return result
def type_r(opcode, operands):
    rs = reg_to_num(operands[1])
    rt = reg_to_num(operands[2])
    rd = reg_to_num(operands[0])
    return opcode + rs + rt + rd + 11 * '0'
def type_i(opcode, operands):
    rs = reg_to_num(operands[1])
    rt = reg_to_num(operands[0])
    immediate = int(operands[2])
    if immediate < 0:
        immediate += 2**16
        immediate = bin(immediate)[2:].rjust(16, '1')
    else:
        immediate = bin(immediate)[2:].rjust(16, '0')
    return opcode + rs + rt + immediate

```

```

def type_j(opcode, operands):
    addr = bin(int(operands[0], 16))
    addr = str(addr[2:-2]).rjust(26, '0')
    return opcode + addr
def asm_compile(instruction, operands):
    if instruction == 'add':
        result = type_r('000000', operands)
    elif instruction == 'sub':
        result = type_r('000001', operands)
    elif instruction == 'addiu':
        result = type_i('000010', operands)
    elif instruction == 'and':
        result = type_r('010000', operands)
    elif instruction == 'andi':
        result = type_i('010001', operands)
    elif instruction == 'ori':
        result = type_i('010010', operands)
    elif instruction == 'xori':
        result = type_i('010011', operands)
    elif instruction == 'sll':
        opcode = '011000'
        rt = reg_to_num(operands[1])
        rd = reg_to_num(operands[0])
        sa = bin(int(operands[2]))[2:].rjust(5, '0')
        result = opcode + 5*'0' + rt + rd + sa + 6*'0'
    elif instruction == 'slti':
        result = type_i('100110', operands)
    elif instruction == 'slt':
        result = type_r('100111', operands)
    elif instruction == 'sw':
        opcode = '110000'
        tempRegex = re.compile('([0-9])\(((.+))\')')
        rs = reg_to_num(tempRegex.search(operands[1]).group(2))
        rt = reg_to_num(operands[0])
        immediate = int(tempRegex.search(operands[1]).group(1))
        if immediate < 0:
            immediate += 2 ** 16
            immediate = bin(immediate)[2:].rjust(16, '1')
        else:
            immediate = bin(immediate)[2:].rjust(16, '0')
        result = opcode + rs + rt + immediate
    elif instruction == 'lw':
        opcode = '110001'
        tempRegex = re.compile('([0-9])\(((.+))\')')
        rs = reg_to_num(tempRegex.search(operands[1]).group(2))
        rt = reg_to_num(operands[0])
        immediate = int(tempRegex.search(operands[1]).group(1))
        if immediate < 0:
            immediate += 2 ** 16
            immediate = bin(immediate)[2:].rjust(16, '1')
        else:
            immediate = bin(immediate)[2:].rjust(16, '0')
        result = opcode + rs + rt + immediate
    elif instruction == 'beq':
        opcode = '110100'
        rs = reg_to_num(operands[0])
        rt = reg_to_num(operands[1])
        immediate = int(operands[2])
        if immediate < 0:
            immediate += 2 ** 16
            immediate = bin(immediate)[2:].rjust(16, '1')
        else:
            immediate = bin(immediate)[2:].rjust(16, '0')
        result = opcode + rs + rt + immediate
    elif instruction == 'bne':
        opcode = '110101'
        rs = reg_to_num(operands[0])
        rt = reg_to_num(operands[1])
        immediate = int(operands[2])

```



```

        if immediate < 0:
            immediate += 2 ** 16
            immediate = bin(immediate)[2:].rjust(16, '1')
        else:
            immediate = bin(immediate)[2:].rjust(16, '0')
            result = opcode + rs + rt + immediate
    elif instruction == 'bltz':
        opcode = '110110'
        rs = reg_to_num(operands[0])
        immediate = int(operands[1])
        if immediate < 0:
            immediate += 2 ** 16
            immediate = bin(immediate)[2:].rjust(16, '1')
        else:
            immediate = bin(immediate)[2:].rjust(16, '0')
            result = opcode + rs + '00000' + immediate
    elif instruction == 'j':
        result = type_j('111000', operands)
    elif instruction == 'jr':
        opcode = '111001'
        rs = reg_to_num(operands[0])
        result = opcode + rs + '0'*21
    elif instruction == 'jal':
        result = type_j('111010', operands)
    elif instruction == 'halt':
        opcode = '111111'
        return opcode + '0'*26
    return result

if __name__ == '__main__':
    asmFile = open('E:\\SYSU_Lessons\\计算机组成原理\\计组实验\\5.实验三:多周期CPU\\test_code.asm',
mode='r', encoding='UTF-8')
    regex = re.compile('([a-zA-Z]+) +(.+)')
    count = 1
    for asmCode in asmFile:
        asmCode = asmCode.strip()
        match = regex.search(asmCode)
        if match is not None:
            instruction = match.group(1)
            operands = match.group(2).split(',')
        else:
            instruction = asmCode
            operands = []
        machineCode = asm_compile(instruction, operands)
        #print(str(count).rjust(2), ':', end='')
        print(machineCode)
        count += 1

```