

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) addiu rt, rs, immediate

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

功能：rt ← rs + (sign-extend)immediate; immediate 无符号扩展再参加“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: $rd \leftarrow rt \ll (\text{zero-extend})sa$, 左移 sa 位, (zero-extend)sa。**==>比较指令**(9) slti rt, rs, **immediate** 带符号数

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

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

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

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

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

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

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

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

特别说明: **immediate** 是从 PC+4 地址开始和转移到的指令之间指令条数。**immediate** 符号扩展之后左移 2 位再相加。为什么要左移 2 位? 由于跳转到的指令地址肯定是 4 的倍数 (每条指令占 4 个字节), 最低两位是“00”, 因此将 **immediate** 放进指令码中的时候, 是右移了 2 位的, 也就是以上说的“指令之间指令条数”。

(13) bne rs, rt, **immediate**

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

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

特别说明: 与 beq 不同点是, 不等时转移, 相等时顺序执行。

(14) bltz rs, **immediate**

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

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

(15) j addr

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

功能：pc $\leftarrow \{(pc+4)[31:28], addr[27:2], 2'b00\}$ ，无条件跳转。

说明：由于 MIPS32 的指令代码长度占 4 个字节，所以指令地址二进制数最低 2 位均为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位可用于存放地址，事实上，可存放 28 位地址，剩下最高 4 位由 pc+4 最高 4 位拼接上。

==> 停机指令

(16) halt

111111	00000000000000000000000000000000(26 位)
--------	--

功能：停机；不改变 PC 的值，PC 保持不变。

三. 实验原理

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期（如果晶振的输出没有经过分频就直接作为 CPU 的工作时钟，则时钟周期就等于振荡周期。若振荡周期经二分频后形成时钟脉冲信号作为 CPU 的工作时钟，这样，时钟周期就是振荡周期的两倍。）

CPU 在处理指令时，一般需要经过以下几个步骤：

(1) 取指令(IF)：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，同时，PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 PC，当然得到的“地址”需要做些变换才送入 PC。

(2) 指令译码(ID)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

(3) 指令执行(EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(4) 存储器访问(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) 结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

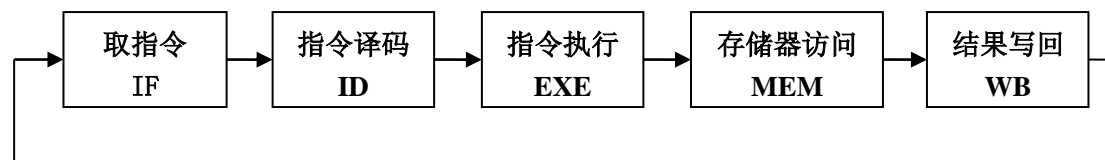
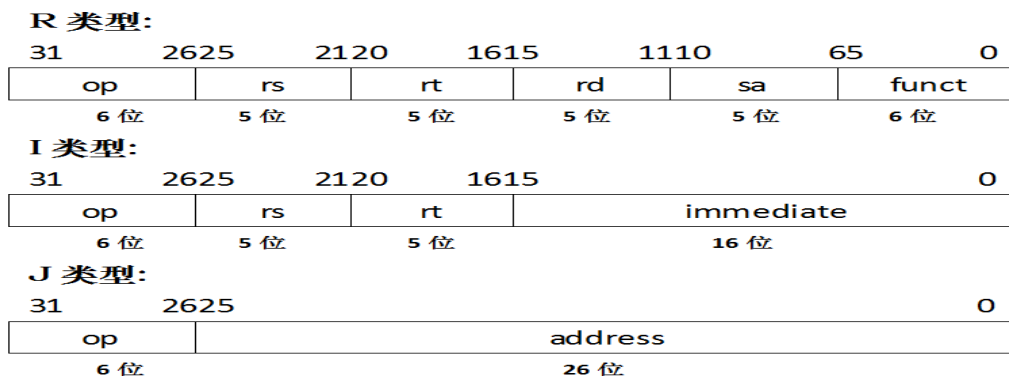


图 1 单周期 CPU 指令处理过程

MIPS 指令的三种格式：



其中,

op: 为操作码;

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

rt: 可读可写。为第 2 个源操作数寄存器, 或目的操作数寄存器, 寄存器地址 (同上);

rd: 只写。为目的操作数寄存器, 寄存器地址 (同上);

sa: 为位移量 (shift amt), 移位指令用于指定移多少位;

funct: 为功能码, 在寄存器类型指令中 (R 类型) 用来指定指令的功能与操作码配合使用;

immediate: 为 16 位立即数, 用作无符号的逻辑操作数、有符号的算术操作数、数据加载 (Load) / 数据保存 (Store) 指令的数据地址字节偏移量和分支指令中相对程序计数器 (PC) 的有符号偏移量;

address: 为地址。

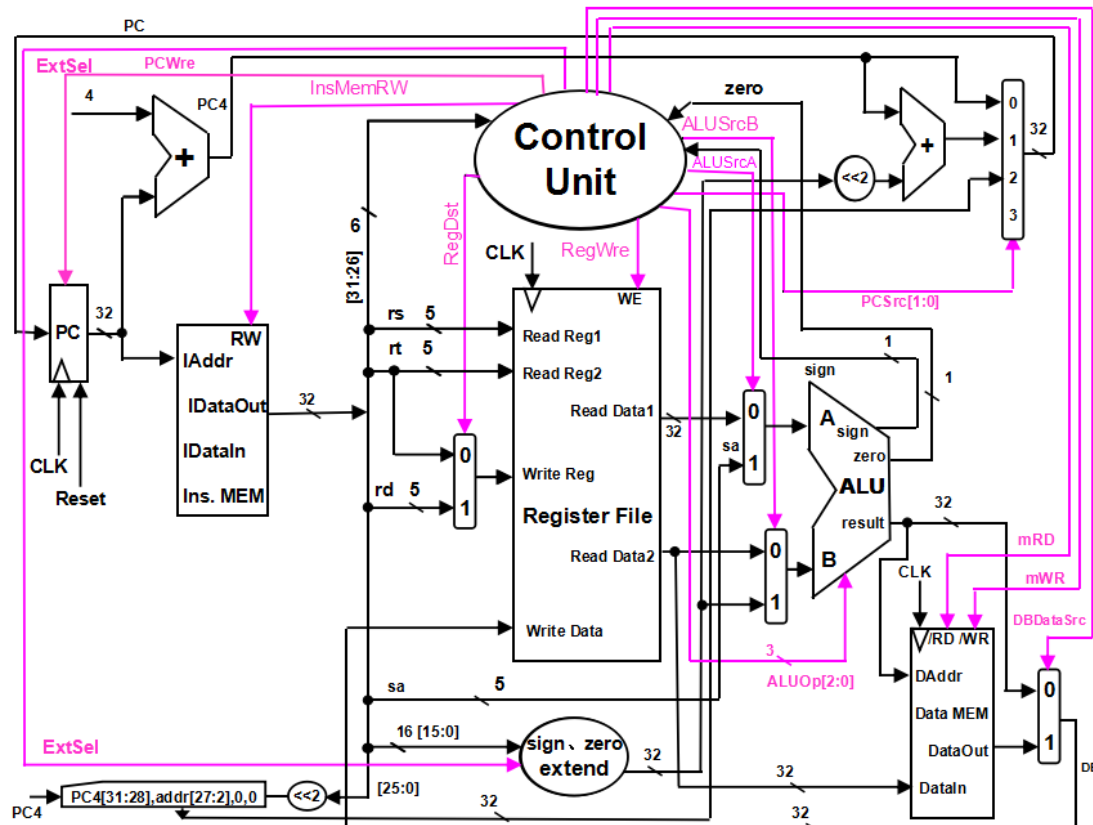


图 2 单周期 CPU 数据通路和控制线路图

图 2 是一个简单的基本上能够在单周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，先给出寄存器地址，读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。图中控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

相关部件及引脚说明：

控制信号名	状态 “0”	状态 “1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addiu、or、and、andi、ori、slti、beq、bne、bltz、sw、lw	来自移位数 sa，同时，进行 (zero-extend)sa，即 $\{27\{1'b0\},sa\}$ ，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、beq、bne、bltz	来自 sign 或 zero 扩展的立即数，相关指令：addi、andi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll	来自数据存储器 (Data MEM) 的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、bltz、sw、halt	寄存器组写使能，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器 (Ins. Data)
mRD	输出高阻态	读数据存储器，相关指令：lw
mWR	无操作	写数据存储器，相关指令：sw
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addiu、andi、ori、slti、lw	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll
ExtSel	(zero-extend)immediate(0 扩展)，相关指令：addiu、andi、ori	(sign-extend)immediate (符号扩展)，相关指令：slti、sw、lw、bne、bne、bltz
PCSrc[1..0]	00: $pc \leftarrow pc+4$ ，相关指令：add、addiu、sub、or、ori、and、andi、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0)； 01: $pc \leftarrow pc+4+(\text{sign-extend})\text{immediate}$ ，相关指令：beq(zero=1)、bne(zero=0)、bltz(sign=1)； 10: $pc \leftarrow \{(pc+4)[31:28],\text{addr}[27:2],2'b00\}$ ，相关指令：j； 11: $pc \leftarrow pc$ ，仅用于 halt 指令	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

表 1 控制信号的作用

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) 【没有用到】~~

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 【没有用到】~~

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

ReadReg1, rs 寄存器地址输入端口

ReadReg2, rt 寄存器地址输入端口

WriteReg, 将数据写入的寄存器端口, 其地址来源 rt 或 rd 字段

WriteData, 写入寄存器的数据输入端口

ReadData1, rs 寄存器数据输出端口

ReadData2, 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

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 与 B 不带符号
110	$Y = (((\text{rega} < \text{regb}) \ \&\& \ (\text{rega}[31] == \text{regb}[31]) \)) \ \ (\ (\text{rega}[31] == 1 \ \&\& \ \text{regb}[31] == 0) \)) \) ? 1 : 0$	比较 A 与 B 带符号
111	$Y = A \oplus B$	异或

表 2 ALU 运算功能表

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，值得注意的问题，设计时，用模块化的思想方法设计，关于 ALU 设计、存储器设计、寄存器组设计等等，也是必须认真考虑的问题。

四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五. 实验过程与结果

A. 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。引脚说明：

clk，输入时钟信号

Reset，复位，低电平有效，初始化PC为0

PCWre，控制信号，PC是否可以更改，0为不更改，1为更改

nextIAddr，下一条指定的地址

currentIAddr，当前正在执行的指令地址

```
module PC(
    input clk, input Reset, input PCWre, input [31:0] nextIAddr,
    output reg [31:0] currentIAddr
);
    initial currentIAddr <= 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，该模块的引脚说明已经在“实验内容”中叙述，这里不再重复。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（在我的代码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96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IAAddr，输出对应地址中的32位机器指令。代码如下：

```
module InstructionMemory(
    input [31:0] IAAddr,
    output [31:0] IDataOut
);
    reg [7:0] ROM [0:95];
    initial begin
        $readmemb("E:/_Vivado/MIPS_CPU_Design/SingleCycleCPU/test_instructions.txt", ROM);
    end
    assign IDataOut[31:24] = ROM[IAAddr+0];
    assign IDataOut[23:16] = ROM[IAAddr+1];
    assign IDataOut[15:8] = ROM[IAAddr+2];
    assign IDataOut[7:0] = ROM[IAAddr+3];
endmodule
```

3. RegisterFile

该模块的引脚说明已经在“实验内容”中叙述，这里不再赘述。

读寄存器堆中是组合逻辑的功能，而写寄存器堆则是时序逻辑。前者只需用assign赋值，后者则需要clk下降沿触发写入。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，该模块的引脚说明已经在“实验内容”中叙述，这里不再赘述。

```
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。该模块的引脚说明已经在“实验内容”中叙述，这里不再赘述。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. ControlUnit

ControlUnit是组合逻辑，引脚说明：

opcode，输入的6位操作码

zero，输入ALU的运算结果是否为0的指示位

sign，输入ALU的运算结果是否为负的指示位

PCWre、ALUSrcA、ALUSrcB、DBDataSrc、RegWre、InsMemRW、mRD、mWR、RegDst、ExtSel、PCSrc、ALUOp，共12个控制信号，共15位。

控制单元的作用是将机器码中的操作码（opcode）转换为各个控制信号，从而控制不同的指令在不同的数据通路中传输。各控制信号的作用见实验内容 - 表1，需要注意的是，表1中的Reset并不是控制单元发出的控制信号，因此实现ControlUnit的时候不考虑它。

根据表1，可以得到各个指令与控制信号之间的关系，如下表：

指令	opcode	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	InsMemRW	mRD	mWR	RegDst	ExtSel	PCSrc[1:0]	ALUOp[2:0]
add	000000	1	0	0	0	1	1	0	0	1	0	00	000
sub	000001	1	0	0	0	1	1	0	0	1	0	00	001
addiu	000010	1	0	1	0	1	1	0	0	0	1	00	000
andi	010000	1	0	1	0	1	1	0	0	0	0	00	100
and	010001	1	0	0	0	1	1	0	0	1	0	00	100
ori	010010	1	0	1	0	1	1	0	0	0	0	00	011
or	010011	1	0	0	0	1	1	0	0	1	0	00	011
sll	011000	1	1	0	0	1	1	0	0	1	0	00	010
slti	011100	1	0	1	0	1	1	0	0	0	1	00	110
sw	100110	1	0	1	0	0	1	0	1	0	1	00	000
lw	100111	1	0	1	1	1	1	1	0	0	1	00	000
beq	110000	1	0	0	0	0	1	0	0	0	1	01(zero)/00	001
bne	110001	1	0	0	0	0	1	0	0	0	1	01(!zero)/00	001
bltz	110010	1	0	0	0	0	1	0	0	0	1	01(sign)/00	000
j	111000	1	0	0	0	0	1	0	0	0	0	10	000
halt	111111	0	0	0	0	0	1	0	0	0	0	11	000

表3 指令与控制信号关系表

根据上表，结合case语句即可生成每条指令的控制信号。为了节约篇幅，这里只给出add、sub这两条指令相关的代码，其它指令都有类似的结构。（完整的代码见ControlUnit.v文件）

```

module ControlUnit(
    input [5:0] opcode,
    input zero,
    input sign,
    output ... 【此处省略】
);
always @(opcode or zero or sign) begin
    case(opcode)
        6'b000000: begin // add
            PCWre <= 1;
            {ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, mRD, mWR, RegDst,
ExtSel} <= 9'b000110010;
            {PCSrc[1:0], ALUOp[2:0]} <= 5'b00_000;
        end
        6'b000001: begin // sub
            PCWre <= 1;
            {ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, mRD, mWR, RegDst,
ExtSel} <= 9'b000110010;
            {PCSrc[1:0], ALUOp[2:0]} <= 5'b00_001;
        end
        ... 【此处省略】
    endcase
end
endmodule

```

7. ImmediateExtend

该模块为组合逻辑，引脚说明如下：

original，扩展前的16位立即数

ExtSel，控制信号，决定是零扩展还是符号扩展

extended，扩展后的32位立即数

该模块的实现方法体现在如下代码中：

```

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

```

8. 数据选择器

根据数据通路图可知，至少需要5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2）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）。工程目录结构如图所示：

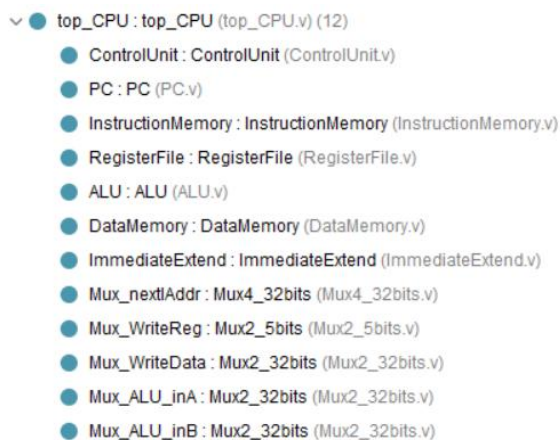


图3 top_CPU工程结构树

B. 仿真验证CPU的正确性

● CPU执行测试程序段的过程

根据老师提供的《关于测试单周期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	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	\$1 = 8		08
0x00000004	ori \$2,\$0,2	\$2 = 2		02
0x00000008	add \$3,\$2,\$1	\$3 = 10		0A
0x0000000C	sub \$5,\$3,\$2	\$5 = 8		08
0x00000010	and \$4,\$5,\$2	\$4 = 0		00
0x00000014	or \$8,\$4,\$2	\$8 = 2		02
0x00000018	sll \$8,\$8,1	\$8 = 4		04
0x0000001C	bne \$8,\$1,-2		≠, 转 18	FC
0x00000018	sll \$8,\$8,1	\$8 = 8		08
0x0000001C	bne \$8,\$1,-2		=, 继续	00
0x00000020	slti \$6,\$2,4	\$6 = 1		01
0x00000024	slti \$7,\$6,0	\$7 = 0		00
0x00000028	addiu \$7,\$7,8	\$7 = 8		08
0x0000002C	beq \$7,\$1,-2		=, 转 28	00
0x00000028	addiu \$7,\$7,8	\$7 = 16		16
0x0000002C	beq \$7,\$1,-2		≠, 继续	08
0x00000030	sw \$2,4(\$1)			0C
0x00000034	lw \$9,4(\$1)	\$9 = 2		0C
0x00000038	addiu \$10,\$0,-2	\$10 = -2		FE
0x0000003C	addiu \$10,\$10,1	\$10 = -1		FF
0x00000040	bltz \$10,-2		<0, 转 3C	FF
0x0000003C	addiu \$10,\$10,1	\$10 = 0		00
0x00000040	bltz \$10,-2		<0, 继续	00
0x00000044	andi \$11,\$2,2	\$11 = 2		02
0x00000048	j 0x00000050		转 50	00
0x0000004C	or \$8,\$4,\$2			00
0x00000050	halt			00

表4 指令实际执行过程

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：

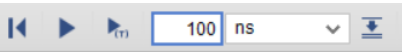
输入	clk	CPU时钟信号
	Reset	CPU复位信号，0为复位
输出	currentIAddr	当前PC地址
	nextIAddr	下一条指令地址
	rs	rs寄存器编号
	rt	rt寄存器编号
	ReadData1	rs寄存器数据
	ReadData2	rt寄存器数据
	ALU_result	ALU运算结果
	DataOut	数据总线数据

表6 top_CPU顶层模块定义的输入输出

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 = 0;
    Reset = 0;
    #25;
    Reset = 1;    // 开始仿真
    #3000;        // 进行3000ns的仿真
    #100; $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16条指令逐一验证。



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

如图4-1所示，0~100ns。当前PC地址（即currentIAddr）为0x00000000。PCSrc信号为00，顺序执行；

rs和rt分别为0号和1号，执行指令后值分别为0和8（注意寄存器堆的写入发生在时钟下降沿）；

ALUOp为000，做加法运算，运算结果为8。ALUSrcA信号为0，ALU的输入A为rs寄存器的值；ALUSrcB信号为1，ALU的输入B为符号扩展（ExtSel信号为1）

后的立即数；

RegDst信号为0，因此目的寄存器是rt；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

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

如图4-1所示，100~200ns。当前PC为0x00000004。PCSrc为00，顺序执行；rs和rt分别为0号和2号，执行后值分别为0和2；

ALUOp为003，做逻辑或运算，结果为2。ALUSrcA信号为0，输入rs寄存器的值；ALUSrcB信号为1，输入零扩展（ExtSel信号为0）后的立即数；

RegDst信号为0，因此目的寄存器是rt；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

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

如图4-1所示，200~300ns。当前PC为0x00000008。PCSrc为00，顺序执行；rs和rt分别是2号和1号，执行前后值不变，分别为2和8；

ALUOp为000，做加法运算，结果为8+2=10（十六进制的0a）。ALUSrcA和ALUSrcB信号均为0，因此ALU的输入来源都是寄存器堆；

RegDst信号为1，因此目的寄存器由rd指定而不是rt；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

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

如图4-1所示，300~400ns。当前PC为0x0000000c。PCSrc为00，顺序执行；rs和rt分别是3号和2号，值分别为10和2；

ALUOp为001，做减法运算，结果为10-2=8。ALUSrcA和ALUSrcB信号均为0，因此ALU的输入来源都是寄存器堆；

RegDst信号为1，因此目的寄存器由rd指定而不是rt；DBDataSrc信号为0，写

回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

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

如图4-1所示，400~500ns。当前PC为0x00000010，PCSrc为00，顺序执行；rs和rt分别是5号和2号，值分别为8和2；

ALUOp为100，做逻辑与运算，结果为 $8 \& 2 = 0$ 。ALUSrcA和ALUSrcB信号均为0，因此ALU的输入来源都是寄存器堆；

RegDst信号为1，因此目的寄存器由rd指定而不是rt；DBDataSrc为0，写回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

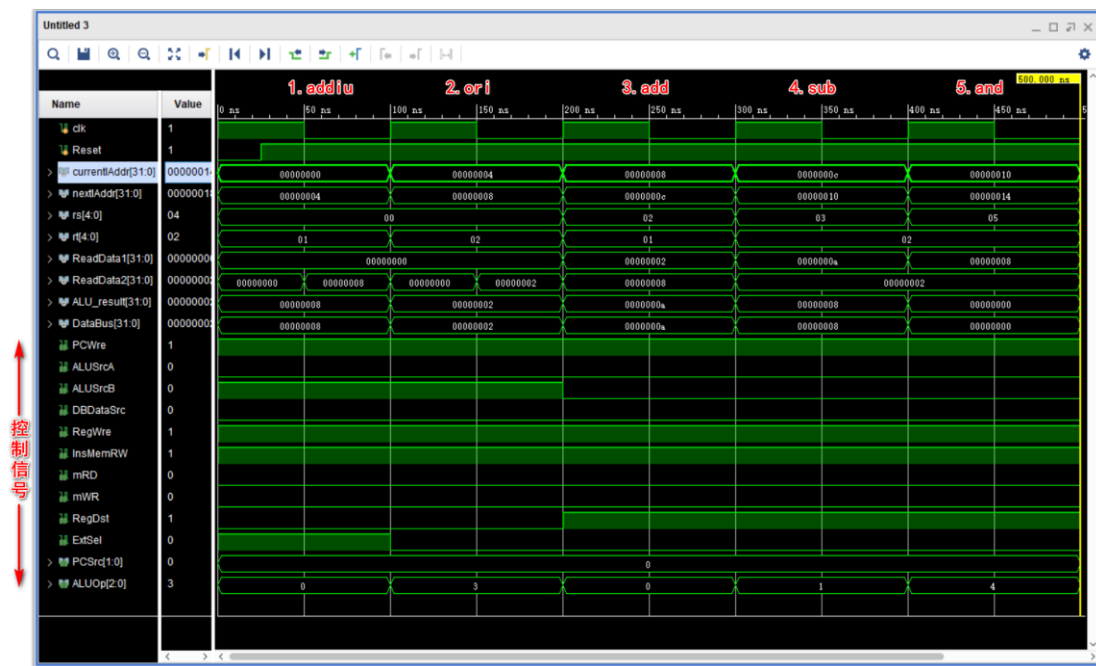


图4-1 波形图，第1~5条指令

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

如图4-2所示，500~600ns。当前PC为0x00000014。PCSrc为00，顺序执行；rs和rt分别是4号和2号，值分别为0和2；

ALUOp为011，做逻辑或运算，结果为 $0 | 2 = 2$ ；ALUSrcA和ALUSrcB信号均为0，因此ALU的输入来源都是寄存器堆；

RegDst信号为1，因此目的寄存器由rd指定而不是rt；DBDataSrc信号为0，写

回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

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

注意，该条指令与其它I型指令有所不同，它的寄存器操作数是rd和rt，sa为立即数，该指令中rs寄存器没有用到。

如图4-2所示，600~700ns。当前PC为0x00000018，PCSrc为0，顺序执行；

rt寄存器为8号，其值为2<<1=4；

ALUOp为010，做逻辑左移操作，结果为4；ALUSrcA信号为1，输入为经零扩展（ExtSel为0）的sa立即数，ALUSrcB信号为0，输入rt寄存器的值；

RegDst信号为1，因此目的寄存器由rd指定；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

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

如图4-2所示，700~800ns。当前PC为0x0000001c。这是跳转指令，PCSrc待定，一会再分析。

rs和rt分别是8号和1号，值分别为4和8；

ALUOp为001，做减法运算，结果为4-8=-4，补码为0xffffffc。此结果不是0，故ALU的zero输出为0，这表示比较结果不相等，bne条件为真。该结果（zero=0）再被送入控制单元，控制单元根据规则发出控制信号PCSrc为01，表示nextIAddr为currentIAddr+4-2×4，下一步将跳转到分支去。由波形图可以看出，当前指令地址为0x0000001c，下一条指令的地址为00000018，这证明跳转确实发生了。该指令不涉及写回寄存器堆，也不涉及数据存储器，因此RegWre、mRD、mWR信号均为0。

另外，900~1000ns也为0x0000001c处的bne指令，只不过这时ALU的zero输出为1，表示结果相等，bne条件为假。PCSrc信号00，不再分支，而是顺序执行到0x00000020。波形图的结果与预期符合得很好。

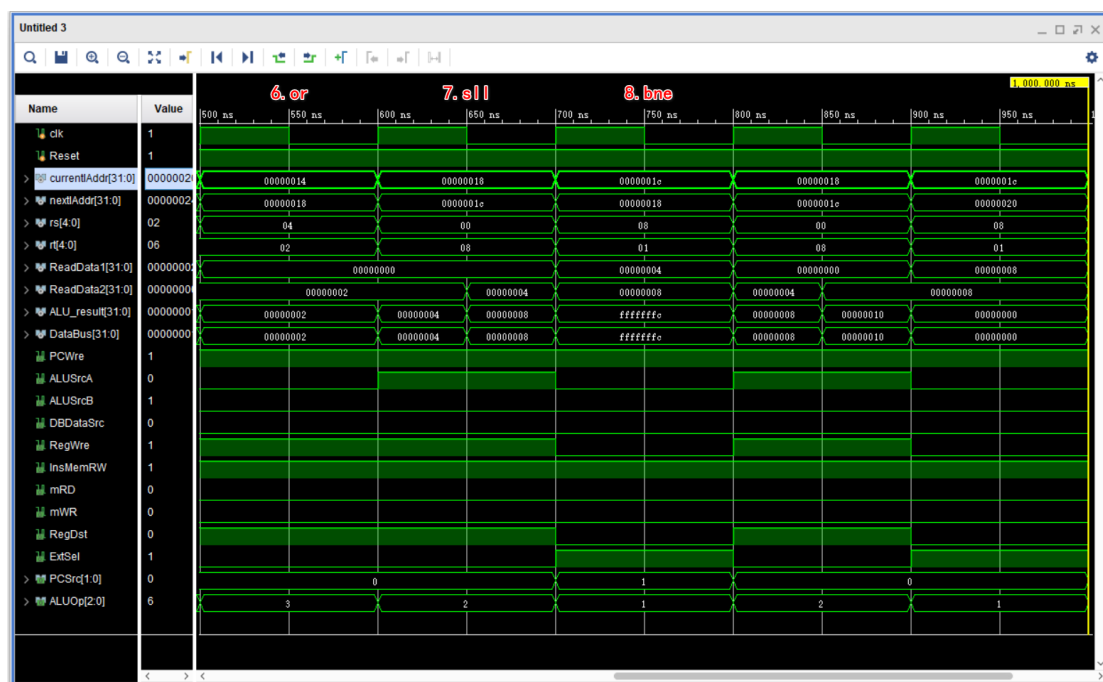


图4-2 波形图，第6~5条指令

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

如图4-3所示，1000~1100ns。当前PC为0x00000020。PCSrc为00，顺序执行；

rs和rt分别为2号和6号，执行后值分别为2和1；

ALUOp为110，做带符号比较操作，结果为 $2 < 1 = 1$ 。ALUSrcA信号为0，输入rs寄存器的值；ALUSrcB信号为1，输入符号扩展（ExtSel信号为1）后的立即数；RegDst信号为1，因此目的寄存器是rt；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

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

如图4-3所示，1300~1400ns。当前PC为0x0000002c，这是跳转指令。

rs和rt分别为7号和1号，值分别为8和8；

ALUOp为001，做减法运算，结果为 $8 - 8 = 0$ ，ALU的zero输出为1，表示运算数相等，beq条件为真。因此PCSrc为10，表示nextIAddr为currentIAddr+4-2×4，下一步将跳转到分支去。由波形图可以看出，当前指令地址为0x0000002c，下一

条指令地址为0x00000028，因此确实发生了跳转。

该指令不涉及写回寄存器堆，也不涉及数据存储器，因此RegWre、mRD、mWR信号均为0。

另外，在1500ns~1600ns也是bne指令，这里的rs和rt的值不再相等，ALU的zero输出为0，beq条件为假。PCSrc信号变为00，顺序执行。观察波形，符合结果。

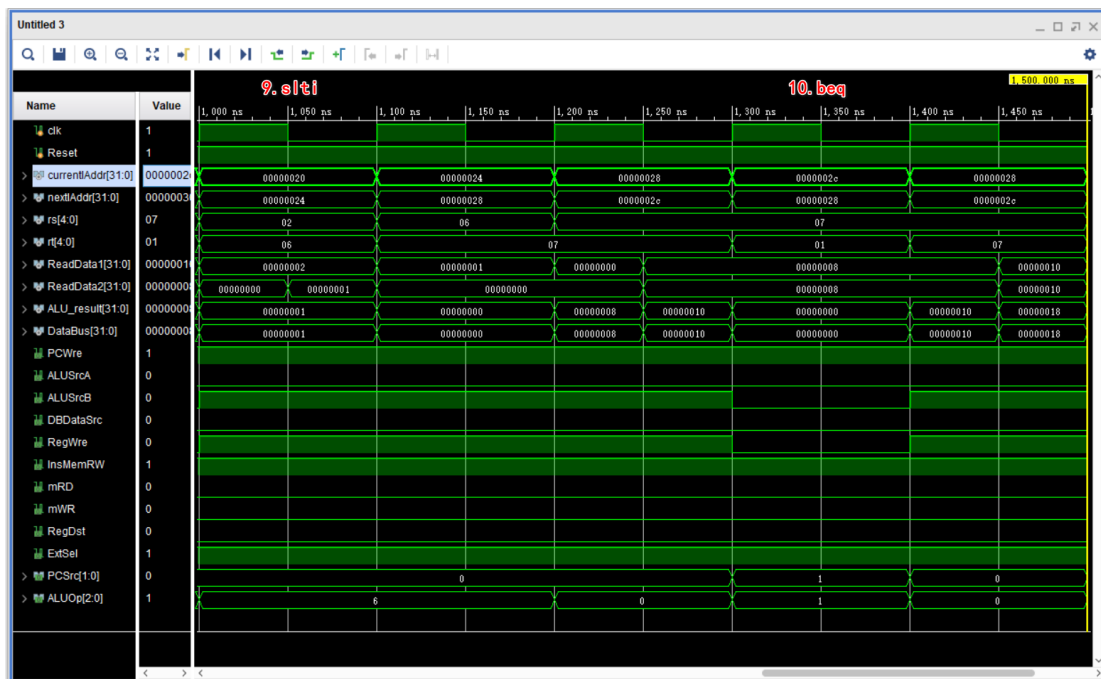


图4-3 波形图，第9~10条指令

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

如图4-4所示，1600ns~1700ns。当前PC为0x00000030，PCSrc为00，顺序执行；

rs和rt分别为1号和2号，值分别为8和2；

ALUOp为000，做加法运算；ALUSrcA为0，运算数A为寄存器堆的rs中的值，ALUSrcB为1，表示运算数B为符号扩展（ExtSel为1）后的立即数；最后ALU运算出来的结果为8+4=12，表示数据存储器地址。

此时mWR信号变为1，数据存储器写使能有效，根据数据通路图便可得知，在时钟下降沿到来时，rt寄存器的值（即8）被写入数据存储器，写入地址即是ALU计算结果，即12。

RegWre信号为0，结果不需写回寄存器堆。

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

如图4-4所示，1700~1800ns。当前PC为0x00000034，PCSrc为00，顺序执行；

rs为1号，其值为8；rt为9号，在执行前值为0；

ALUOp为000，做加法运算；ALUSrcA为0，运算数A为寄存器堆的rs中的值，ALUSrcB为1，表示运算数B为符号扩展（ExtSel为1）后的立即数；最后ALU运算出来的结果为 $8+4=12$ ，表示数据存储器地址。

此时mWR信号变为0，不可写；mRD信号变为1，数据存储器读使能有效；

DBDataSrc信号为1，数据总线（DataBus）上的数据来自数据存储器的输出（DataOut），值为2。

RegWre信号为1，数据总线上的数据被写入寄存器堆。

13. bltz \$10, -2

如图4-4所示，1800~1900ns。当前PC为0x00000040，PCSrc为00，顺序执行；

rs和rt分别为10号（十六进制的0a）和0号，值分别为-1（补码为0xffffffff）和0；ALUOp为000，做加法运算，这是因为将运算数与0相加结果仍为本身，这样就可以利用ALU的sign输出。ALUSrcA和ALUSrcB信号都为0，ALU的操作数都来自寄存器堆。ALU将算出 $-1+0=-1$ ，因此sign输出为1，bltz条件为真。因此PCSrc为10，表示nextIAddr为 $\text{currentIAddr}+4-2\times 4$ ，下一步将跳转到分支去。由波形图可以看出，当前指令地址为0x00000040，下一条指令地址为0x0000003c，因此确实发生了跳转。

该指令不涉及写回寄存器堆，也不涉及数据存储器，因此RegWre、mRD、mWR信号均为0。

另外2200~2300ns也是bltz指令，只不过这时rs的值为0，ALU的sign输出为0，因此PCSrc信号为00，将顺序执行，不做跳转。

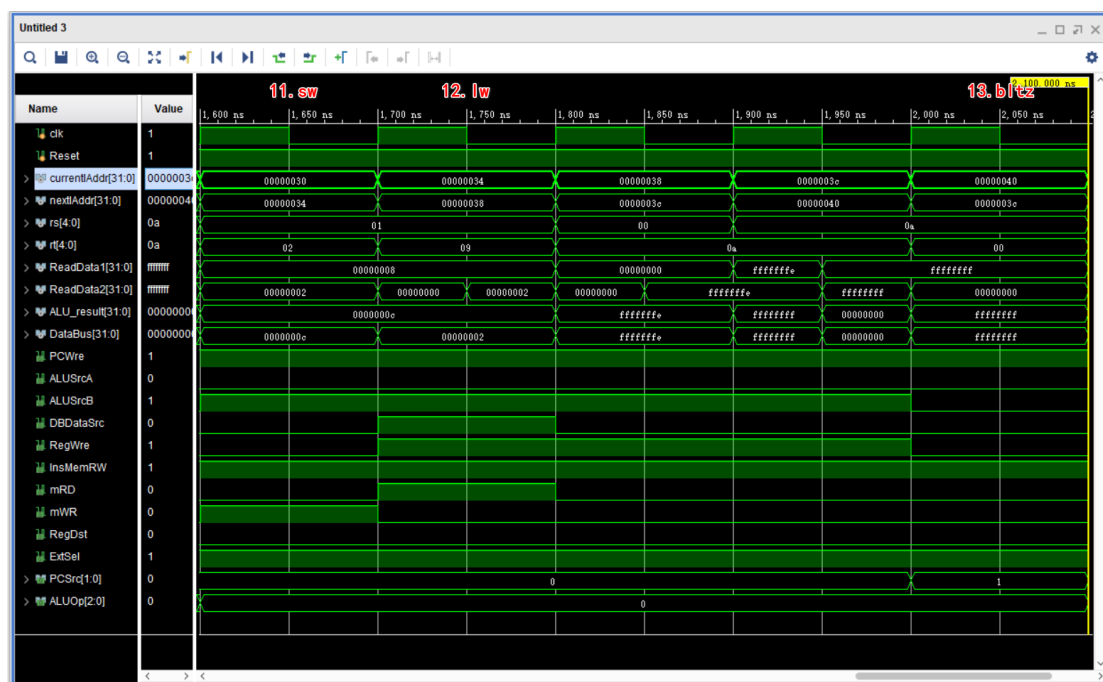


图4-4 波形图，第11~13条指令

14. andi \$11, \$2, 2

如图4-5所示，2300~2400ns。当前PC为0x00000044。PCSrc为00，顺序执行；

rs和rt分别为2号和1号（十六进制的0b），执行后值分别为2和2；

ALUOp为100，做逻辑与运算，结果为2。ALUSrcA信号为0，输入rs寄存器的值；

ALUSrcB信号为1，输入零扩展（ExtSel信号为0）后的立即数；

RegDst信号为0，因此目的寄存器是rt；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

15. j 0x00000050

如图4-5所示，2400~2500ns。当前PC为0x00000048。PCSrc为10，为无条件跳转。跳转目标地址是32位的，其高4位和当前PC的高4位相同，低28位是指令的低26位左移两位的结果，这个地址就是0x00000050。从波形图也可以看到，nextIAddr为0x00000050。

J型指令不涉及寄存器堆、数据存储器、ALU，无关控制信号不做讨论。

16. halt

如图4-5所示，2500ns以后。当前PC为0x00000050，PCSrc为11，因此nextIAddr于currentIAddr相同（见实验内容 - 表1）。halt指令使得PC保持不变，从波形图可以看出，2500ns以后，所有信号、数值都保持不变了。该指令不涉及其他操作，无关信号不做讨论。

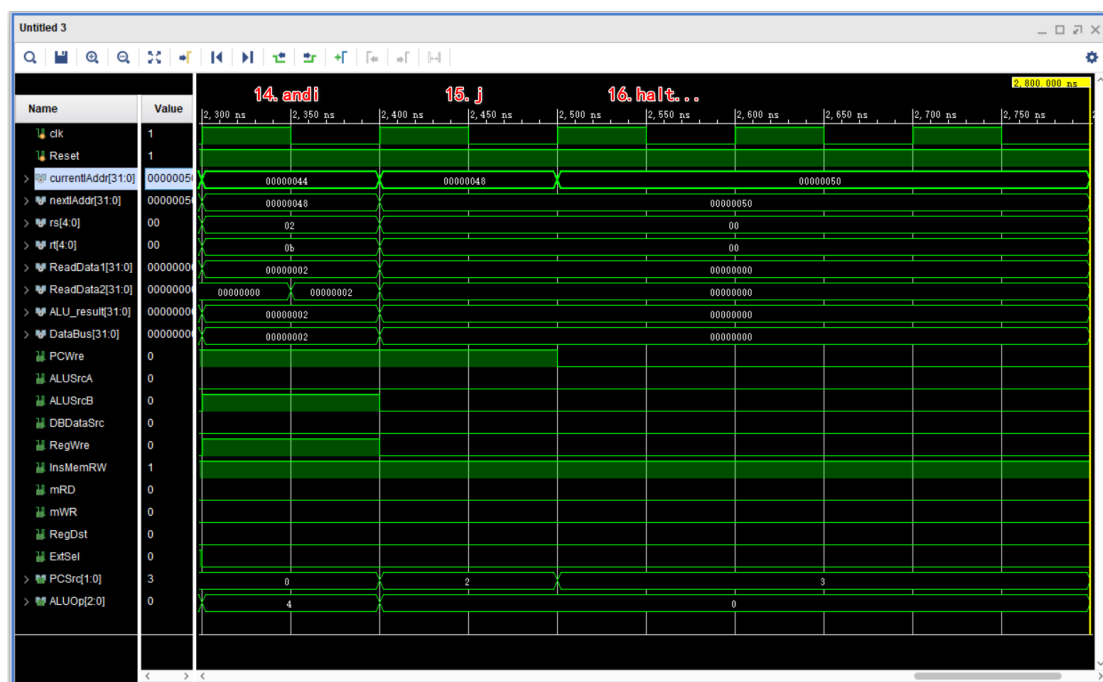


图4-5 波形图，第14~16条指令

C. 在Basys3上实现

● 思想方法

成功仿真单周期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：

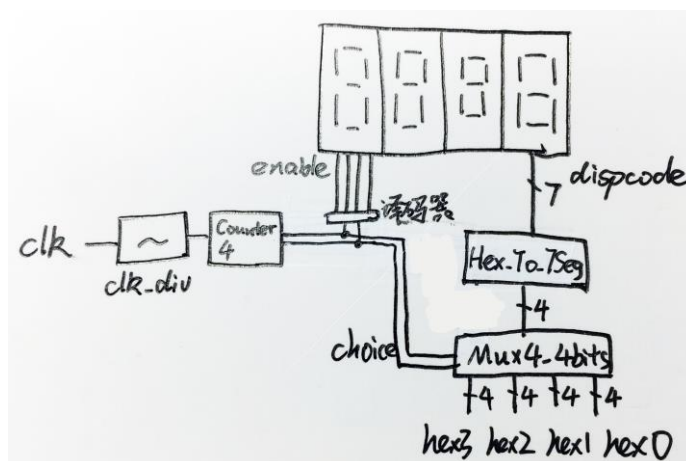


图5

该模块的完整代码见附录B。此时工程目录结构如图所示：

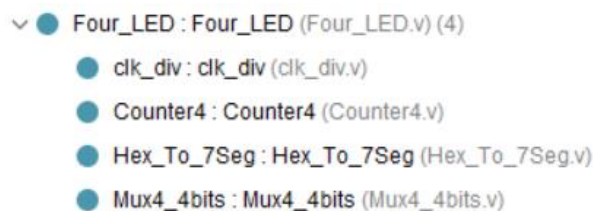


图6 Four_LED工程结构树

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：

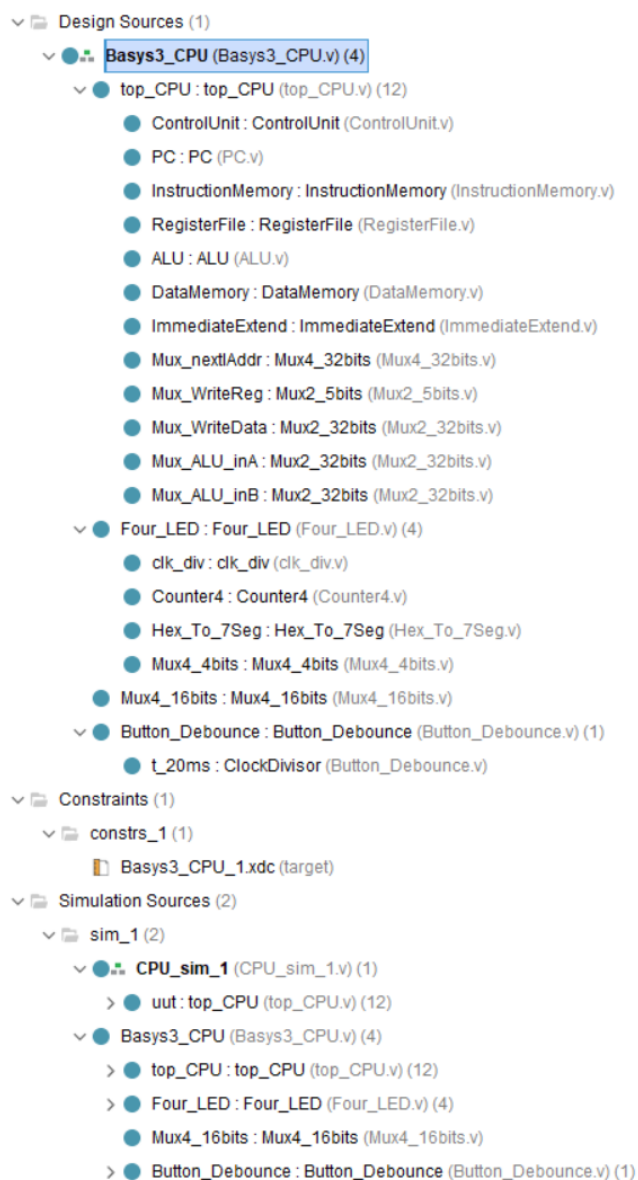
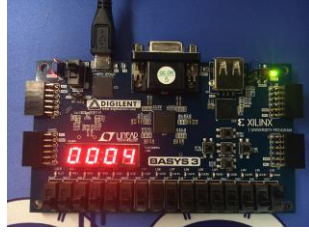





图7 完整工程结构树



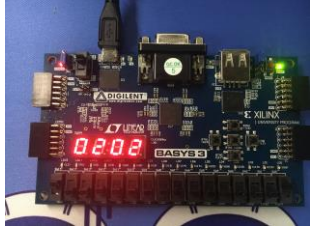
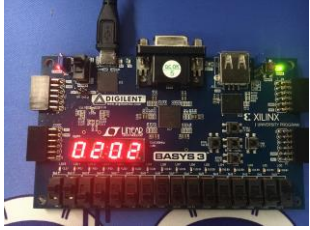
● 检验结果

展示所给测试程序段的前5条指令（完整测试程序段见附录A），下面这些照片显示的结果全部与仿真吻合，因此不再这里逐一说明了。


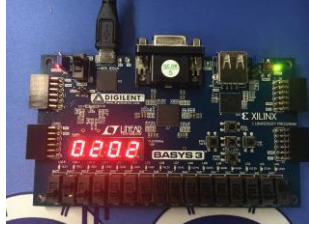
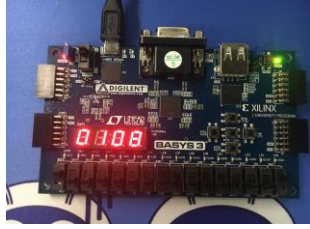

1. addiu \$1,\$0,8

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



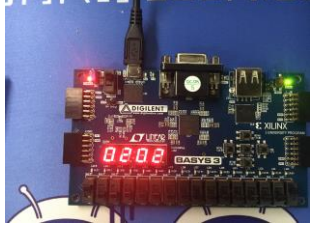

2. ori \$2,\$0,2

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

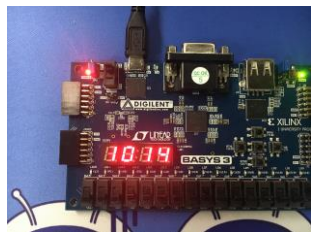
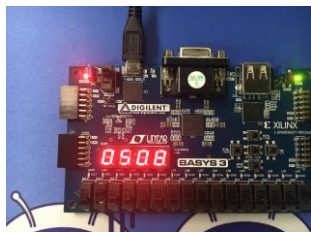
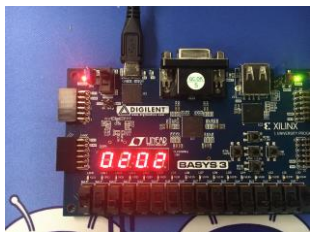
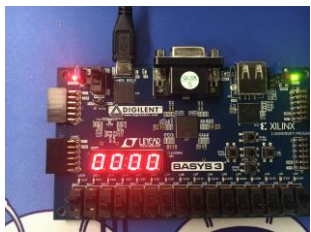
3. add \$3,\$2,\$1

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

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

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

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

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

六. 实验心得

1. 全局观念。单周期CPU设计这一项实验作业是一项比较大的工程项目，因此，在开始实验之前，必须要有全局观，要在脑海中构建出该实验的大致框架：熟悉CPU内部结构、熟悉单周期CPU的数据通路图、理解PC、指令存储器、寄存器堆、ALU、数据存储器的的工作原理，还要清楚地了解实验内容中要求的每一条指令的组成。首先要问自己一些问题，如，“一条32位长的指令哪个字段表示操作码？”“无条件跳转指令j的目标地址是如何得到的？”等。只有当有了清晰的全局观以后，才能有效率地用Verilog代码实现单周期CPU。
2. 通过本次实验，解决了绝大多数在实验一（MIPS汇编）中遗留的问题。对Verilog语言的熟练度大大提高，对阻塞赋值与非阻塞赋值、过程赋值语句与持续赋值语句、wire类型与reg类型等概念有了更清晰的认识，也学会了在合适的地方使用合适的表达式和语句。
3. 更加深入地理解了Verilog语言的“模块化”这一特性。本次设计的单周期CPU，其模块可以分为3个级别：最顶层的Basys3_CPU，其中包含top_CPU和Four_LED两个模块，再往下则是各个底层模块。
4. 关于控制单元的触发条件。在设计控制单元（ControlUnit）时，其敏感信号表达式必须为：“opcode or zero or sign”，需要注意的是不能漏掉zero和sign。

最开始我写这个模块的时候敏感信号只写了opcode，但是在仿真到分支指令的时候遇到了bug，如0x0000001c处的bne指令，第一次执行该指令时，\$8和\$1的值不相等，理应跳转。然而实际情况是，控制信号仅仅由opcode触发，而opcode的改变来自PC的改变，发生在时钟上升沿，此时ALU的输出仍然是上一条指令（sll）的结果，

新指令使得ALU的zero改变，但这个改变无法影响控制信号，所以得到的控制信号其实是错误的。sign信号同理。因此，敏感信号表达式必须包含zero和sign才能产生正确的控制信号。

5. 关于bltz指令。bltz指令格式为“bltz rs, immediate”，它只有一个寄存器操作数rs。由于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，rt就是0号寄存器了。再将ALU设置为加法，所以ALU的运算结果就是rs寄存器中的值，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。
6. 关于j指令。要清楚j指令的目标地址方式。测试程序段中的“j 0x00000050”，这个地址实际上就是目标地址了，但是要特别注意，这不是机器码中的26位地址。机器码中的26位地址是0x00000050右移2位的结果，即0x14，详见附录A表格的倒数第三行。
7. 约束文件。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]
```
8. 烧板后，按钮抖动问题。由于按钮采用机械弹性开关，在闭合和断开的瞬间可能伴随一连串的抖动，在本次实验中造成的后果就是按一次按钮执行了多条指令。解决办法是增加按键消抖代码，检测按钮按下后执行20ms的延时。我使用的消抖模块见Button_Debounce.v文件。
9. 烧板后，第一条指令没有写回的问题。首先明确“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”。在Basys3板上，按钮松开为0，按下为1，因此按下按钮的动作对应一个时钟上升沿。

烧板后，我们首先把Reset置位，这时PC已经初始化为为第1条指令的地址（0x0）。然后按下按钮并松开，这时CPU时钟先是上升沿再是下降沿，对应的行为是PC递增为第2条指令的地址（0x4）、写入寄存器堆。这里第一条指令的结果并没有被写入到寄存器堆中去！这将影响后续用到\$1的指令！解决办法是，先按下按钮不松开，然后将Reset置位，最后再松开按钮。这样，CPU时钟首先能获得一个下降沿，因而数据总线上的数据顺利写回寄存器堆。

事实上，我认为这个问题有更好的解决方式，比如让Reset信号置位时写入一次寄

寄存器堆。但实际操作却失败了，没有产生预期效果，暂时还不知道为什么，希望将来能够解决这一问题。

10. 写Four_LED模块的时候，用到了上个学期数字电路的知识，唯一的不同是数字电路中用的是连接芯片的方式，而这次实验中用的是Verilog语言。因此，只要了解数字电路原理和Verilog语法，就可以用Verilog为工具实现数字电路的思想。
11. 烧板过程中造成了一次系统蓝屏（Windows 10 版本1803），在周边同学中简单考察了一下，很多同学也都出现了这一问题，但原因未知。

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	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
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	addiu \$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
0x00000030	sw \$2,4(\$1)	100110	00001	00010	0000 0000 0000 0100	=	98220004
0x00000034	lw \$9,4(\$1)	100111	00001	01001	0000 0000 0000 0100	=	9C290004
0x00000038	addiu \$10,\$0,-2	000010	00000	01010	1111 1111 1111 1110	=	080AFFFE
0x0000003C	addiu \$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
0x00000044	andi \$11,\$2,2	010000	00010	01011	0000 0000 0000 0010	=	404B0002
0x00000048	j 0x00000050	111000	00 0000 0000 0000 0000 0001 0100			=	E0000050
0x0000004C	or \$8,\$4,\$2	010011	00100	00010	01000 00000 000000	=	4C824000
0x00000050	halt	111111	00000	00000	0000 0000 0000 0000	=	FC000000

附录B: test_instructions.txt

```

00001000 00000001 00000000 00001000
01001000 00000010 00000000 00000010
00000000 01000001 00011000 00000000
00000100 01100010 00101000 00000000
01000100 10100010 00100000 00000000
01001100 10000010 01000000 00000000
01100000 00001000 01000000 01000000
11000101 00000001 11111111 11111110
01110000 01000110 00000000 00000100
01110000 11000111 00000000 00000000
00001000 11100111 00000000 00001000
11000000 11100001 11111111 11111110
10011000 00100010 00000000 00000100
10011100 00101001 00000000 00000100
00001000 00001010 11111111 11111110
00001001 01001010 00000000 00000001
11001001 01000000 11111111 11111110
01000000 01001011 00000000 00000010
11100000 00000000 00000000 00010100
01001100 10000010 01000000 00000000
11111100 00000000 00000000 00000000

```