



《计算机组成原理与接口技术实验》 实验报告

(实验二)

学 院 名 称 : 数据科学与计算机学院

专业 (班级) : 16 软件工程三 (5) 班

学 生 姓 名 : 邱奕浩

学 号 : 16340186

时 间 : 2018 年 6 月 1 日

成绩：

实验二：单周期CPU设计与实现

一、实验目的

- (1) 掌握单周期 CPU 数据通路图的构成、原理及其设计方法；
- (2) 掌握单周期 CPU 的实现方法，代码实现方法；
- (3) 认识和掌握指令与 CPU 的关系；
- (4) 掌握测试单周期 CPU 的方法；
- (5) 掌握单周期 CPU 的实现方法。

二、实验内容

设计一个单周期 CPU，该 CPU 至少能实现以下指令功能操作。指令与格式如下：

==> 算术运算指令

(1) `add rd, rs, rt` (说明：以助记符表示，是汇编指令；以代码表示，是机器指令)

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

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

(2) `addi rt, rs, immediate`

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

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

(3) `sub rd, rs, rt`

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

功能：rd←rs - rt

==> 逻辑运算指令

(4) `ori rt, rs, immediate`

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

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

(5) `and rd, rs, rt`

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

功能：rd←rs & rt; 逻辑与运算。

(6) `or rd, rs, rt`

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

功能：rd←rs | rt; 逻辑或运算。

==> 移位指令

(7) `sll rd, rt, sa`

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

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

==>比较指令

(8) slti rt, rs,immediate 带符号

011011	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

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

==> 存储器读/写指令

(9) sw rt,immediate(rs) 写存储器

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

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

(10) lw rt, immediate(rs) 读存储器

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

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

==> 分支指令

(11) beq rs,rt,immediate

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

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

(12) bne rs,rt,immediate

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

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

==>跳转指令

(13) j addr

111000	addr[27..2]
--------	-------------

功能: $pc \leftarrow -\{(pc+4)[31..28], \text{addr}[27..2], 2\{0\}\}$, 无条件跳转。

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

==> 停机指令

(14) halt

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

功能：停机；不改变 PC 的值，PC 保持不变。

在本文档中，提供的相关内容对于设计可能不足或甚至有错误，希望同学们在设计过程中如发现有问题，请你们自行改正，进一步补充、完善。谢谢！

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：

R 类型:

31	26 25	21 20	16 15	11 10	6 5	0
op	rs	rt	rd	sa	funct	
6 位	5 位	5 位	5 位	5 位	6 位	

I 类型:

31	26 25	21 20	16 15	0
op	rs	rt	immediate	
6 位	5 位	5 位	16 位	

J 类型:

31	26 25	0
op	address	
6 位	26 位	

其中,

op: 为操作码;

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

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

rd: 只写。为目的操作数寄存器, 寄存器地址 (同上);

sa: 为位移量 (shift amt), 移位指令用于指定移多少位;

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

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

address: 为地址。

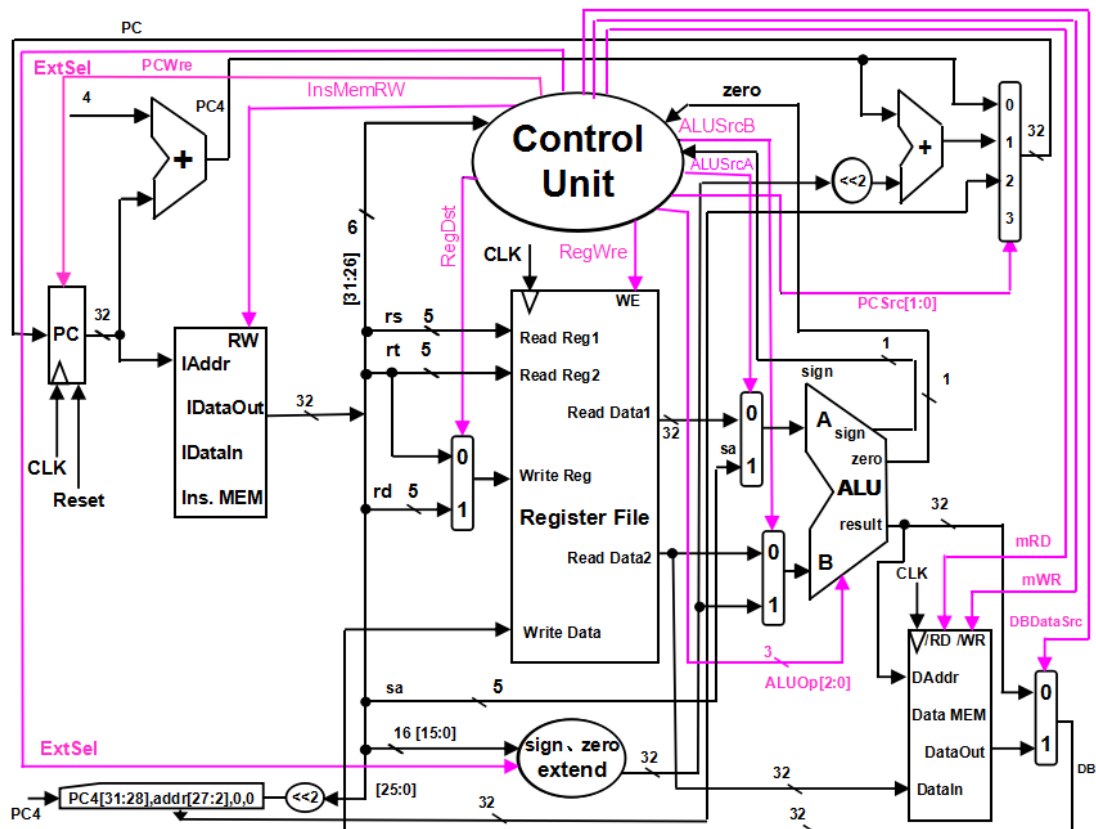


图 2 单周期 CPU 数据通路和控制线路图

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

表 1 控制信号的作用

控制信号名	状态 “0”	状态 “1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre 是否停机	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addi、or、and、ori、beq、bne、slti、sw、lw	来自移位数 sa，同时，进行 (zero-extend)sa，即 {{27{0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、sll、beq、bne	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、slti、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：	寄存器组写使能，相关指令：add、

	beq、bne、sw、halt、j	addi、sub、ori、or、and、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	输出高阻态	读数据存储器，相关指令：lw
mWR	无操作	写数据存储器，相关指令：sw
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、lw、slti	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll
ExtSel	(zero-extend)immediate(0 扩展)，相关指令：ori	(sign-extend)immediate (符号扩展)，相关指令：addi、slti、sw、lw、beq、bne
PCSrc[1..0]	00: pc←pc+4，相关指令：add、addi、sub、or、ori、and、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)； 01: pc←pc+4+(sign-extend)immediate，相关指令：beq(zero=1)、bne(zero=0)； 10: pc←-{(pc+4)[31:28],addr[27:2],2{0}}，相关指令：j； 11: 未用	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

相关部件及引脚说明：

Instruction Memory: 指令存储器，

- Iaddr，指令存储器地址输入端口
- IDataIn，指令存储器数据输入端口（指令代码输入端口）
- IDataOut，指令存储器数据输出端口（指令代码输出端口）
- RW，指令存储器读写控制信号，为 0 写，为 1 读

Data Memory: 数据存储器，

- Daddr，数据存储器地址输入端口
- DataIn，数据存储器数据输入端口
- DataOut，数据存储器数据输出端口
- /RD，数据存储器读控制信号，为 0 读
- /WR，数据存储器写控制信号，为 0 写

Register File: 寄存器组

- Read Reg1，rs 寄存器地址输入端口
- Read Reg2，rt 寄存器地址输入端口
- Write Reg，将数据写入的寄存器端口，其地址来源 rt 或 rd 字段
- Write Data，写入寄存器的数据输入端口
- Read Data1，rs 寄存器数据输出端口
- Read Data2，rt 寄存器数据输出端口
- WE，写使能信号，为 1 时，在时钟边沿触发写入

ALU: 算术逻辑单元

- result，ALU 运算结果
- zero，运算结果标志，结果为 0，则 zero=1；否则 zero=0

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	$Y = A + B$	加
001	$Y = A - B$	减
010	$Y = B \ll A$	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

需要说明的是以上数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定 ALU 的运算功能(当然，以上指令没有完全用到提供的 ALU 所有功能，但至少必须能实现以上指令功能操作)。从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出以上表 1，这样，从表 1 可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表，再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC 的改变是在时钟上升沿进行的，这样稳定性较好。另外，值得注意的问题，设计时，用模块化、层次化的思想方法设计，关于如何划分模块、如何整合成一个系统等等，是必须认真考虑的问题。

四、实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五、实验过程与结果

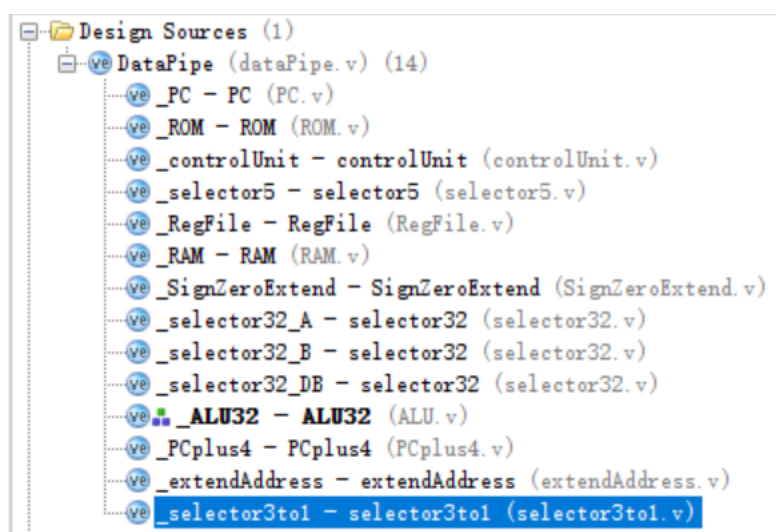
● CPU设计方法和过程

本次单周期CPU的设计采用模块化层次化的设计思想，根据数据通路图把整个CPU划分为12个子模块：

1. PC程序计数器模块：得到当前指令以及获取下一条指令的地址；
2. PCPlusFour模块：指令地址加4；
3. ROM指令存储器：根据PC传送的指令地址输出指令代码
4. RegFile寄存器组模块：进行指令数据的读写；

5. ALU算数运算单元：对操作数进行运算；
 6. SignZeroExtend-0符号位扩展模块：对字段进行符号位或者0位扩展至32位；
 7. extendAddress跳转地址扩展模块：针对j跳转指令，通过扩展得到下一步指令的地址；
 8. RAM数据存储器：对运算结果或者寄存器内容进行保存；
 9. ControlUnit控制逻辑单元：通过发出控制信号控制以上各个子模块，达到子模块进行不同功能实现的目的；
 10. Selector5 五位数据选择器：根据控制信号筛选五位的rd，rt进入寄存器组；
 11. Selector32 32位数据选择器：根据控制信号筛选进入ALU的操作数以及写入寄存器组的值；
 12. Selector3to1 3选1数据选择器（其实是四选一的方法），筛选下一条指令的地址。
- 最后是通过构造一个顶层模块DataPipe来将各个子模块进行连接。

Vivado的各个设计文件如图



各个模块的代码说明

1. PC程序计数器模块，根据数据通路图确定输入输出信号。一开始程序计数器必须初始化为0，当未停机时，处于时钟上升沿时，输入的地址直接输出，停机时输出地址不变。

```
// PC 程序计数器
module PC(
    clk,    //时钟信号
    reset,  //复位
```

```

    input_pc, //输入下一条指令地址
    output_pc, //输出当前指令地址
    PCWre //停机指令
);

    input wire[31:0] input_pc;
    output reg[31:0] output_pc;
    input clk, reset, PCWre;

    //等待时钟上升沿
    always@(posedge clk)begin
        if(reset == 0)begin //初始化
            output_pc = 0;
        end
        else if(!PCWre)begin //停机
            output_pc = output_pc;
        end
        else if(PCWre)begin //不停机, 下一条指令地址成为当前指令地址
            output_pc = input_pc;
        end
    end

endmodule

```

2. PCPlusFour模块: 每一条指令占四个字节 (32bits) ,增加一条指令, 地址加4, 产生连续的指令地址;

```

module PCplus4(
    inputPC,
    outputPC
);

    input wire [31:0] inputPC;
    input wire [31:0] outputPC;
    //PC+4
    assign outputPC[31:0] = inputPC[31:0] + 4;

endmodule

```

3. ROM指令存储器: 根据PC传送的指令地址输出相应指令存储器相应位置的指令代码,

在使用指令存储器之前需要对存储器进行指令的初始化。

```

module ROM(
    rd,
    addr,
    dataOut
); // 指令存储器模块

    input rd; // 读使能信号
    input [31:0] addr; // 存储器地址 32 条指令
    output reg [31:0] dataOut; // 输出的指令
    reg [7:0] mem[127:0]; // 存储器定义必须用 reg 类型，存储器存储单元 8 位
    长度，共 128 个存储单元

    initial begin // 初始化指令内容
        $readmemb ("E:/rom_data.txt", mem); // 数据文件 rom_data (.coe
    或.txt)。
    end

    // 当前指令地址进入，和读取信号
    always @( rd or addr ) begin
        if (rd==1) begin // 为 1，读存储器。大端数据存储模式
            dataOut[31:24] = mem[addr];
            dataOut[23:16] = mem[addr+1];
            dataOut[15:8] = mem[addr+2];
            dataOut[7:0] = mem[addr+3];
        end
    end

endmodule

```

4. RegFile寄存器组模块：接受指令存储器指令代码的相应字段如rs rd,, 读入或者写入相应位置寄存器的值；此处需要注意需要等待时钟信号的下降沿触发，效果会比较稳定。

```

// 寄存器堆
module RegFile(
    input CLK, // 时钟信号
    input reset, // 复位信号
    input RegWre, // 寄存器写使能信号
    input [4:0] ReadReg1, // 读取寄存器 1
    input [4:0] ReadReg2, // 读取寄存器 2
    input [4:0] WriteReg, // 写寄存器 1
    input [31:0] WriteData, // 写入寄存器

```

```

    output [31:0] ReadData1, // 输出 readReg1
    output [31:0] ReadData2 // 输出 readreg2
);

reg [31:0] regFile[0:31]; // 寄存器定义必须用 reg 类型
integer i;
initial begin
    regFile[0] = 0;
end
assign ReadData1 = regFile[ReadReg1]; // 直接输出 rs
assign ReadData2 = regFile[ReadReg2];
always @ (negedge CLK or negedge reset) begin // 必须用时钟边沿触发
    if (reset==0) begin
        for(i=1;i<32;i=i+1)
            regFile[i] <= 0; // 寄存器初始化赋值为 0
        end
        else if (RegWre == 1 && WriteReg != 0) // WriteReg != 0,
$0 寄存器不能修改
            regFile[WriteReg] <= WriteData; // 写寄存器, 将要写
入的值赋给寄存器
        end
    endmodule

```

5. ALU算术逻辑单元：对操作数进行运算，根据操作码的不同，对输入ALU的两个数据值进行不同的运算操作，同时产生0信号和输出结果；

```

module ALU32(
    ALUOp,
    rega,
    regb,
    result,
    zero
);
    input [2:0] ALUOp;
    input [31:0] rega;
    input [31:0] regb;
    output reg [31:0] result;
    output zero;
    initial begin
        result = 0;
    end
end

```

```

assign zero = (result==0)?1:0;    // 0 信号
always @( ALUOp or rega or regb ) begin
    case (ALUOp) // 根据操作码来判断执行的操作
        3'b000 : result = rega + regb;
        3'b001 : result = rega - regb;
        3'b010 : result = regb << rega;
        3'b011 : result = rega | regb;
        3'b100 : result = rega & regb;
        3'b101 : result = (rega < regb)?1:0; // 不带符号比
较
        3'b110 : begin // 带符号比较
            result = (((rega<regb) && (rega[31] ==
rega[31] )) || ( ( rega[31] ==1 && regb[31] == 0))) ? 1:0;
        end
        3'b111: result = rega^regb;
        default : begin
            result = 0;
        end
    endcase
end
endmodule

```

6. SignZeroExtend-0符号位扩展模块:对立即数字段进行符号位或者0位扩展至32位,应用于有立即数字段的指令。

```

module SignZeroExtend(
    ExtSel,
    inputNum,
    outputNum
);

input ExtSel;
input wire[15:0] inputNum;
output reg[31:0] outputNum;

initial begin
    outputNum = 0;
end
always @ (ExtSel or inputNum)begin
    if(ExtSel)begin
        outputNum = {{16{inputNum[15]}}, inputNum[15:0]};
    end
end

```

```

        end
    else begin
        outputNum = {{16{1'b0}},inputNum[15:0]};
    end
end

endmodule

```

7. extendAddress跳转地址扩展模块: 针对j跳转指令, 通过扩展得到下一步指令的地址; 这里需要注意的是地址扩展的方式是: 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, 这样, 除了最高6位操作码外, 还有26位可用于存放地址, 事实上, 可存放28位地址了, 剩下最高4位由pc+4最高4位拼接上。

```

module extendAddress(
    toExtendAddress,
    PCPlusFourAddress,
    JumpAddress
);

    input wire[25:0] toExtendAddress;
    input wire[31:0] PCPlusFourAddress;
    output wire[31:0] JumpAddress;
    assign JumpAddress = {PCPlusFourAddress[31:28],
toExtendAddress[25:0],{2{0}}};
    //assign ExtendAddress = {pcPlusFour[31:28], Address[25:0], 2{0}};

endmodule

```

8. RAM数据存储器: 当数据存储器处于写状态时, 会将writeData数据内容保存在首地址为address的存储空间; 处于读取状态的时候会根据地址输出首地址为address(来自ALU运算结果)的数据DataOut。注意: 数据的存放是遵循大端存放的原则, 即数据高位有效字节存储在存储器低地址位置。

```

module RAM(
    clk,

```

```

address,
writeData, // [31:24], [23:16], [15:8], [7:0]
mRD, // 为1, 正常读; 为0, 输出高阻态
mWR, // 为1, 写; 为0, 无操作
DataOut
);
input clk, mRD, mWR;
input [31:0] address;

input [31:0] writeData;
reg [7:0] ram[0:63]; // 数据存储器定义
output reg [31:0] DataOut;

initial begin
    DataOut = 0;
end

//
always@( negedge clk ) begin
    if( mWR==1 ) begin // 写
        ram[address] <= writeData[31:24];
        ram[address+1] <= writeData[23:16];
        ram[address+2] <= writeData[15:8];
        ram[address+3] <= writeData[7:0];
    end
    else begin
        if(mRD == 1)begin // 读
            DataOut[31:24] = ram[address+3];
            DataOut[23:16] = ram[address+2];
            DataOut[15:8] = ram[address+1];
            DataOut[7:0] = ram[address];
        end
        else if(mRD == 0)begin // 高阻态
            DataOut[31:24] = 8'bz;
            DataOut[23:16] = 8'bz;
            DataOut[15:8] = 8'bz;
            DataOut[7:0] = 8'bz;
        end
    end
end
endmodule

```

9. ControlUnit控制逻辑单元: 通过发出控制信号控制以上各个子模块, 达到子模块进行

不同功能实现的目的。输入的操作码确定不同的指令功能，从而使用case子句来确定各个控制信号为0或1，指挥其他子模块发挥不同的功能。

```

module controlUnit(
    ops,
    zero,
    ExtSel,
    PCWre,
    insMemRW,
    RegDst,
    RegWre,
    ALUOp,
    PCSrc,
    ALUSrcA,
    ALUSrcB,
    mRD,
    mWR,
    DBDataSrc
);

    initial begin
        ExtSel = 0;
        PCWre = 1;    // 更改 pc
        insMemRW = 1; // 写指令
        RegDst = 0;
        RegWre = 0;
        ALUOp = 0;
        PCSrc = 0;
        ALUSrcA = 0;
        ALUSrcB = 0;
        mRD = 0;
        mWR = 0;
        DBDataSrc = 0;
    end

```

10. Selector5 五位数据选择器：根据控制信号筛选五位的rd, rt进入寄存器组；

```

// 五位数据选择器
module selector5(

```



```

        control,
        data1,
        data2,
        result
    );

    input  control; //控制信号
    input [4:0] data1;
    input [4:0] data2;
    output [4:0] result;
    assign result = (control == 0)? data1:data2; //控制信号为1, 选 data1,
    否则 data2

endmodule

```

11. Selector32 32位数据选择器（设计方法同上）：根据控制信号筛选进入ALU的操作数以及写入寄存器组的值；

```

//32 位数据选择器
module selector32(
    control,
    Data1,
    Data2,
    result
);

    input control;
    input wire[31:0] Data1;
    input wire[31:0] Data2;
    output wire[31:0] result;
    assign result = (control == 0) ? Data1: Data2;

endmodule

```

12. Selector3to1 3选1数据选择器（其实是四选一的方法），筛选下一条指令的地址。当控制信号为00时筛选出PC加四地址，为01时筛选出PC+4+(immediate<<2)地址，为10时，筛选出（来自extendAddress）跳转地址。

```

module selector3to1(
    pcPlusFour,

```

```

        extendImm,
        jumpAddress,
        PCsrc,
        result
    );

    input wire[31:0] pcPlusFour;    // 0 1
    input wire[31:0] extendImm;    // 1
    input wire[31:0] jumpAddress;  // 2
    input [1:0] PCsrc;             // 控制信号量

    output reg[31:0] result;        // 产生下一条指令地址

    always @( pcPlusFour or extendImm or jumpAddress or PCsrc)begin
        case (PCsrc)
            2'b00: result = pcPlusFour;
            2'b01: result = pcPlusFour + (extendImm << 2);
            2'b10: result = jumpAddress;
            default:begin
                result = result; // 不变
            end
        endcase
    end

endmodule

```

建立顶层模块DataPipe，根据数据通路图将各个子模块连接起来。以下是每个子模块实例化的代码。

```

//PC 时钟运行
PC _PC(clk, reset, NextAddress, CurrentAddress, PCWre);

// 指令存储器模块
// mRD = 1;
ROM _ROM(InsMemRW, CurrentAddress, IDataOut);

```

```

//控制单元实例化
controlUnit _controlUnit(opcode, zero, ExtSel, PCWre, InsMemRW,
RegDst,RegWre, ALUOp, PCSrc, ALUSrcA, ALUSrcB, mRD, mWR, DBDataSrc);

// 五位选择器选择进入写寄存器的内容
selector5 _selector5(RegDst, rt, rd, WriteReg);

// 寄存器组模块化
RegFile _RegFile(clk, reset, RegWre, rs, rt, WriteReg, WriteData,
ReadData1, ReadData2);

//实例化数据存储器，命名一律用 _name 格式
RAM _RAM(clk, ALUResult, ReadData2, mRD, mWR, DataOut);

//0 符号扩展位
SignZeroExtend _SignZeroExtend(ExtSel, immediate, ExtendResult);

//32 位数据选择器，输入 ALU
selector32 _selector32_A(ALUSrcA, ReadData1, sa, dataA);
selector32 _selector32_B(ALUSrcB, ReadData2, ExtendResult, dataB);
selector32 _selector32_DB(DBDataSrc, ALUResult, DataOut, WriteData);
// alu32 模块，接受数据选择器的数据，产生结果和 0
ALU32 _ALU32(ALUOp, dataA, dataB, ALUResult, zero);

wire[31:0] PCPlusFourAddress; // PC 加 4 模块，进入三选一模块
//PC + 4 模块产生地址
PCplus4 _PCplus4(CurrentAddress, PCPlusFourAddress);

wire[31:0] JumpAddress; //跳转地址，进入三选一模块
//产生跳转地址
extendAddress
_extendAddress(toExtendAddress,PCPlusFourAddress,JumpAddress);

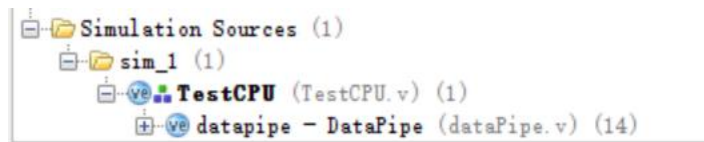
//三选一选择器确定下一条指令地址
selector3to1 _selector3to1(PCPlusFourAddress, ExtendResult,
JumpAddress, PCSrc, NextAddress);

```

● 测试验证CPU

测试单周期CPU的方法是编写一个仿真文件TestCPU.v，其中实例化DataPipe数据通

路顶层模块，运行仿真得到波形图。初始化PC的值，也就是以上程序段首地址
 PC=0x00000000,以上程序段从0x00000000地址开始存放。其中对时钟信号做处理：
 每隔50ns，时钟信号改变。



以下是测试程序段，在运行仿真时会写入指令存储器。

地址	汇编程序	指令代码					16 进制数代码	
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)			
0x00000000	addi \$1,\$0,8	000001	00000	00001	0000 0000 0000 1000	=	04010008	
0x00000004	ori \$2,\$0,2	010000	00000	00010	0000 0000 0000 0010		40020002	
0x00000008	add \$3,\$2,\$1	000000	00010	00001	0001 1000 0000 0000		00411800	
0x0000000C	sub \$5,\$3,\$2	000010	00011	00010	0010 1000 0000 0000		08622800	
0x00000010	and \$4,\$5,\$2	010001	00101	00010	0010 0000 0000 0000		44a22000	
0x00000014	or \$8,\$4,\$2	010010	00100	00010	0100 0000 0000 0000		48824000	
0x00000018	sll \$8,\$8,1	011000	00000	01000	0100 0000 0100 0000		60084040	
0x0000001C	bne \$8,\$1,-2 (≠, 转 18)	110001	01000	00001	1111 1111 1111 1110		C501FFFE	
0x00000020	slti \$6,\$2,8	011011	00010	00110	0000 0000 0000 1000		6C460008	
0x00000024	slti \$7,\$6,0	011011	00110	00111	0000 0000 0000 0000		6CC70000	
0x00000028	addi \$7,\$7,8	000001	00111	00111	0000 0000 0000 1000		04E70008	
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	j 0x00000040	111000	00000	00000	0000 0000 0001 0000		E0000010	
0x0000003C	addi \$10,\$0,10	000001	00000	01010	0000 0000 0000 1010		040A000A	
0x00000040	halt	111111	00000	00000	0000000000000000	=	FC000000	

```

module TestCPU();
    reg _clk;
    reg _reset;
    wire _zero;
    wire[31:0] _CurrentAddress;
    wire[31:0] _NextAddress;
    wire[1:0] _PCSrc;
  
```

```

wire [2:0] _ALUOp;
wire[31:0] _ReadData1, _ReadData2, _ALUResult;
wire[31:0] _ExtendOut, _DataOut, _IDataOut;
wire _InsMemRW, _mRD, _mWR;
wire[4:0] _WriteReg;
wire[31:0] _WriteData;
DataPipe datapipe(
    .clk(_clk),
    .reset(_reset),
    .zero(_zero),
    .CurrentAddress(_CurrentAddress),
    .NextAddress(_NextAddress),
    .PCSrc(_PCSrc),
    .ALUOp(_ALUOp),
    .ReadData1(_ReadData1),
    .ReadData2(_ReadData2),
    .ALUResult(_ALUResult),
    .ExtendOut(_ExtendOut),
    .DataOut(_DataOut),
    .WriteReg(_WriteReg),
    .InsMemRW(_InsMemRW),
    .mRD(_mRD),
    .mWR(_mWR),
    .IDataOut(_IDataOut),
    .WriteData(_WriteData)
);

initial begin
    _clk = 0;
    _reset = 0; //初始化输入

    #50; //50 ns 后开始 重置
    _clk = 1;
    #50; //不再重置 pc 0
    _reset = 1;
    forever #50 begin //产生时钟信号
        _clk = !_clk;
    end
end
endmodule

```
















以下说明每条指令的正确性。

地址	汇编程序	指令代码					16 进制数代码	
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)			
0x00000000	addi \$1,\$0,8	000001	00000	00001	0000 0000 0000 1000	=	04010008	













（右边栏为16进制，sa字段为了处理方便采用32位扩展显示。以下同）















根据波形图，可以看到当前地址为00000000，下一条指令地址为00000004

指令代码为04010008 ,ALU操作码为 000, 可以看出运算结果为8

	_clk	1	
	_reset	0	
	_zero	0	
	_CurrentAddress[31:0]	00000000	当前指令地址
	_NextAddress[31:0]	00000004	下一条指令地址
	_PCSrc[1:0]	0	
	_ALUOp[2:0]	0	ALU操作码
	_ReadData1[31:0]	00000000	
	_ReadData2[31:0]	00000000	
	_ALUResult[31:0]	00000008	ALU运算结果
	_dataA[31:0]	00000000	
	_dataB[31:0]	00000008	
	_ExtendOut[31:0]	ZZZZZZZZ	
	_DataOut[31:0]	00000000	
	_IDataOut[31:0]	04010008	指令代码

将运算结果 8 存入 1 号寄存器

 _WriteReg[4:0]	01	8 写入寄存器 \$1
 _WriteData[31:0]	00000008	
 _rs[4:0]	00	字段
 _rt[4:0]	01	
 _rd[4:0]	00	
 _sa[31:0]	00000000	控制单元操作码
 _opcode[5:0]	01	
 _PCPlusFour...ress[31:0]	00000004	立即数和立即数0扩展
 _immediate[15:0]	0008	
 _ExtendResult[31:0]	00000008	
 _toExtendAddress[25:0]	0010008	
 _JumpAddress[31:0]	00000000	

Name	Value	Data ..
 ops[5:0]	01	Array
 zero	0	Logic
 ExtSel	1	Logic
 PCWre	1	Logic
 insMemRW	1	Logic
 RegDst	0	Logic
 RegWre	1	Logic
 ALUOp[2:0]	0	Array
 PCSrc[1:0]	0	Array
 ALUSrcA	0	Logic
 ALUSrcB	1	Logic
 mRD	0	Logic
 mWR	0	Logic
 DEDataSrc	0	Logic

地址	汇编程序	指令代码			
		op(6)	rs(5)	rt(5)	rd(5)/immediate
					16 进制数代码

					(16)	
0x00000004	ori \$2,\$0,2	010000	00000	00010	0000 0000 0000 0010	40020002

可以看出运算结果为 2

	_clk	1	
	_reset	1	
	_zero	0	
	_CurrentA...ss[31:0]	00000004	当前指令地址
	_NextAddress[31:0]	00000008	下一条指令地址
	_PCSrc[1:0]	0	
	_ALUOp[2:0]	3	ALU操作码
	_ReadData1[31:0]	00000000	
	_ReadData2[31:0]	00000000	
	_ALUResult[31:0]	00000002	ALU运算结果 00000002
	_dataA[31:0]	00000000	ALU两个操作数
	_dataB[31:0]	00000002	
	_ExtendOut[31:0]	ZZZZZZZZ	
	_DataOut[31:0]	ZZZZZZZZ	
	_IDataOut[31:0]	40020002	当前指令地址

运算结果 2 被放进 2 号寄存器

	_WriteReg[4:0]	02	写入寄存器的编号 02
	_WriteData[31:0]	00000002	写入02寄存器的数值
	_rs[4:0]	00	rs rt字段
	_rt[4:0]	02	
	_rd[4:0]	00	
	_sa[31:0]	00000000	
	_opcode[5:0]	10	控制单元操作码
	_PCPlusF...ss[31:0]	00000008	
	_immediate[15:0]	0002	立即数2
	_ExtendResult[31:0]	00000002	立即数2 0位扩展
	_toExten...ss[25:0]	0020002	
	_JumpAddress[31:0]	00000000	

Name	Value	Data ..
ops[5:0]	10	Array
zero	0	Logic
ExtSel	0	Logic
PCWre	1	Logic
insMemRW	1	Logic
RegDst	0	Logic
RegWre	1	Logic
ALUOp[2:0]	3	Array
PCSrc[1:0]	0	Array
ALUSrcA	0	Logic
ALUSrcB	1	Logic
mRD	0	Logic
mWR	0	Logic
DBDataSrc	0	Logic

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
0x00000008	add \$3,\$2,\$1	000000	00010	00001	0001 1000 0000 0000	00411800

可以看到 2 号寄存器和 1 号寄存器内容相加得到结果 10

_CurrentAddress[31:0]	00000008	当前指令地址
_NextAddress[31:0]	0000000c	下一条指令地址
_PCSrc[1:0]	0	
_ReadData1[31:0]	00000002	
_ReadData2[31:0]	00000008	
_ALUResult[31:0]	0000000a	ALU运算结果 = 10
ALUOp[2:0]	0	ALU操作码000
_dataA[31:0]	00000002	ALU操作数 = 2
_dataB[31:0]	00000008	ALU操作数 = 8
_ExtendOut[31:0]	ZZZZZZZZ	
_DataOut[31:0]	ZZZZZZZZ	
_IDataOut[31:0]	00411800	当前指令代码

将运算结果 10 放入寄存器 3

_WriteReg[4:0]	03	写入3号寄存器
_WriteData[31:0]	0000000a	写入的数值
_rs[4:0]	02	2号 1 号寄存器
_rt[4:0]	01	
_rd[4:0]	03	
_sa[31:0]	00000003	
_opcode[5:0]	00	控制操作码
_PCPlusFourAddress[31:0]	0000000c	PC+4地址

Name	Value
ops[5:0]	00
zero	0
ExtSel	0
PCWre	1
insMemRW	1
RegDst	1
RegWre	1
ALUOp[2:0]	0
PCSrc[1:0]	0
ALUSrcA	0
ALUSrcB	0
mRD	0
mWR	0
DBDataSrc	0

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
0x0000000C	sub \$5,\$3,\$2	000010	00011	00010	0010 1000 0000 0000	08622800

3 号寄存器内容减去 2 号寄存器内容，得到结果 8

<div><div>🔍</div><div><div>clk</div><div>reset</div><div>zero</div><div>CurrentAddress[31:0]</div><div>NextAddress[31:0]</div><div>PCSrc[1:0]</div><div>ReadData1[31:0]</div><div>ReadData2[31:0]</div><div>ALUResult[31:0]</div><div>ALUOp[2:0]</div><div>dataA[31:0]</div><div>dataB[31:0]</div><div>ExtendOut[31:0]</div><div>DataOut[31:0]</div><div>IDataOut[31:0]</div></div></div>	<div><div>1</div><div>1</div><div>0</div><div>0000000c 当前指令地址</div><div>00000010 下一条指令地址</div><div>0</div><div>0000000a 寄存器堆的输出值 10 和 2</div><div>00000002</div><div>00000008 ALU运算结果 8</div><div>1</div><div>0000000a ALU操作数10, 2</div><div>00000002</div><div>ZZZZZZZZ</div><div>ZZZZZZZZ</div><div>08622800 当前指令代码</div></div>
---	---

结果8写入5号寄存器

<div><div>WriteReg[4:0]</div><div>WriteData[31:0]</div></div>	<div><div>05 写入5号寄存器</div><div>00000008 写入的内容</div></div>
<div><div>rs[4:0]</div><div>rt[4:0]</div><div>rd[4:0]</div></div>	<div><div>03</div><div>02</div><div>05</div><div>寄存器编号3, 2, 5</div></div>
<div><div>sa[31:0]</div><div>opcode[5:0]</div><div>PCPlusFourAddress[31:0]</div></div>	<div><div>00000005</div><div>02 控制单元操作码</div><div>00000010 PC+4地址</div></div>

Name	Value
ops[5:0]	02
zero	0
ExtSel	0
PCWre	1
insMemRW	1
RegDst	1
RegWre	1
ALUOp[2:0]	1
PCSrc[1:0]	0
ALUSrcA	0
ALUSrcB	0
mRD	0
mWR	0
DBDataSrc	0

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
0x00000010	and \$4,\$5,\$2	010001	00101	00010	0010 0000 0000 0000	44a22000

5号寄存器和2号寄存器内容进行与运算，结果为0

clk	1
_reset	1
_zero	1
_CurrentAddress[31:0]	00000010 当前指令地址
_NextAddress[31:0]	00000014 下一条指令地址
_PCSrc[1:0]	0
_ReadData1[31:0]	00000008 寄存器堆输出值
_ReadData2[31:0]	00000002
_ALUResult[31:0]	00000000 ALU与运算结果为 0
_ALUOp[2:0]	4
_dataA[31:0]	00000008 ALU操作数8 和 2
_dataB[31:0]	00000002
_ExtendOut[31:0]	ZZZZZZZZ
_DataOut[31:0]	ZZZZZZZZ
_IDataOut[31:0]	44a22000 当前指令代码

将运算结果写入 4 号寄存器

_WriteReg[4:0]	04	写入4号寄存器
_WriteData[31:0]	00000000	写入的数据0
_rs[4:0]	05	读取寄存器5, 2号
_rt[4:0]	02	
_rd[4:0]	04	写入寄存器4号
_sa[31:0]	00000004	
_opcode[5:0]	11	控制单元操作码
_PCPlusFourAddress[31:0]	00000014	PC+4地址

Name	Value
ops[5:0]	11
zero	1
ExtSel	0
PCWr	1
insMemRW	1
RegDst	1
RegWr	1
ALUOp[2:0]	4
PCSrc[1:0]	0
ALUSrcA	0
ALUSrcB	0
mRD	0
mWR	0
DBDataSrc	0

地址	汇编程序	指令代码				16 进制数代码
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	
0x00000014	or \$8,\$4,\$2	010010	00100	00010	0100 0000 0000 0000	48824000

2 号寄存器和 4 号寄存器内容进行与运算，得到结果 2

_CurrentAddress[31:0]	00000014	当前指令地址
_NextAddress[31:0]	00000018	下一条指令地址
_PCSrc[1:0]	0	
_ReadData1[31:0]	00000000	
_ReadData2[31:0]	00000002	寄存器堆的输出值1 2
_ALUResult[31:0]	00000002	ALU或运算结果
_ALUOp[2:0]	3	ALU操作码
_dataA[31:0]	00000000	ALU操作数
_dataB[31:0]	00000002	
_ExtendOut[31:0]	ZZZZZZZZ	
_DataOut[31:0]	ZZZZZZZZ	
_IDataOut[31:0]	48824000	当前指令代码

将运算结果2写入寄存器8号

_WriteReg[4:0]	08	写入8号寄存器
_WriteData[31:0]	00000002	写入的内容
_rs[4:0]	04	读取寄存器4, 2
_rt[4:0]	02	
_rd[4:0]	08	写入寄存器8
_sa[31:0]	00000008	
_opcode[5:0]	12	控制单元操作码
_PCPlusFourAddress[31:0]	00000018	PC+4操作码













Name	Value
ops[5:0]	12
zero	0
ExtSel	0
PCWre	1
insMemRW	1
RegDst	1
RegWre	1
ALUOp[2:0]	3
PCSrc[1:0]	0
ALUSrcA	0
ALUSrcB	0
mRD	0
mWR	0
DEDataSrc	0

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
0x00000018	sll \$8,\$8,1	011000	00000	01000	0100 0000 0100 0000	60084040

数字 1 左移两位（8 号寄存器内容为 2），得到结果 4

	_CurrentAddress[31:0]	00000018	当前指令地址
	_NextAddress[31:0]	0000001c	下一条指令地址
	_PCSrc[1:0]	0	
	_ReadData1[31:0]	00000000	
	_ReadData2[31:0]	00000002	
	_ALUResult[31:0]	00000004	ALU左移运算结果4
	_ALUOp[2:0]	2	
	_dataA[31:0]	00000001	
	_dataB[31:0]	00000002	ALU操作数 1 << 2.
	_ExtendOut[31:0]	ZZZZZZZZ	
	_DataOut[31:0]	ZZZZZZZZ	
	_IDataOut[31:0]	60084040	当前指令代码

将结果 4 写入寄存器 8 号

	_WriteReg[4:0]	08	写入8号寄存器
	_WriteData[31:0]	00000004	写入内容为 4
	_rs[4:0]	00	
	_rt[4:0]	08	读寄存器8号，写入8号
	_rd[4:0]	08	
	_sa[31:0]	00000001	偏移量 1
	_opcode[5:0]	18	
	_PCPlusFourAddress[31:0]	0000001c	PC+4地址
	_immediate[15:0]	4040	
	_ExtendResult[31:0]	00004040	
	_toExtendAddress[25:0]	0084040	
	_JumpAddress[31:0]	00000000	











Name	Value
ops[5:0]	18
zero	0
ExtSel	1
PCSrc	1
insMemRW	1
RegDst	1
RegWrite	1
ALUOp[2:0]	2
PCSrc[1:0]	0
ALUSrcA	1
ALUSrcB	0
mRD	0
mWR	0
DEDataSrc	0

地址	汇编程序	指令代码					16 进制数代码	
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)			
0x0000001C	bne \$8,\$1,-2 (≠, 转18)	110001	01000	00001	1111 1111 1111 1110		C501	FFFE

对寄存器 8 与 1 号进行判断，不等跳回上条指令

 _CurrentAddress[31:0]	0000001c	当前指令地址
 _NextAddress[31:0]	00000018	下一条指令地址-跳回上一条
 _PCSrc[1:0]	1	PCSrc = 1分支跳转
 _ReadData1[31:0]	00000004	
 _ReadData2[31:0]	00000008	
 _ALUResult[31:0]	00000000	结果为0
 _ALUOp[2:0]	4	
 _dataA[31:0]	00000004	ALU操作数4, 8
 _dataB[31:0]	00000008	
 _ExtendOut[31:0]	ZZZZZZZZ	
 _DataOut[31:0]	ZZZZZZZZ	
 _IDataOut[31:0]	c501fffe	当前指令代码

完成跳转

 _CurrentAddress[31:0]	00000018	跳回上一条指令
 _NextAddress[31:0]	0000001c	进入跳转指令
 _PCSrc[1:0]	0	
 _ReadData1[31:0]	00000000	
 _ReadData2[31:0]	00000004	
 _ALUResult[31:0]	00000008	ALU结果
 _ALUOp[2:0]	2	
 _dataA[31:0]	00000001	ALU操作数
 _dataB[31:0]	00000004	
 _ExtendOut[31:0]	ZZZZZZZZ	
 _DataOut[31:0]	ZZZZZZZZ	
 _IDataOut[31:0]	60084040	当前指令代码


```

CurrentAddress[31:0] 0000001c 当前指令地址
NextAddress[31:0] 00000020 指令地址+4
PCSrc[1:0] 0
ReadData1[31:0] 00000008
ReadData2[31:0] 00000008
ALUResult[31:0] 00000008
ALUOp[2:0] 4
_dataA[31:0] 00000008 ALU操作数相等，进入PC+4
_dataB[31:0] 00000008
ExtendOut[31:0] ZZZZZZZZ
DataOut[31:0] ZZZZZZZZ
IDataOut[31:0] c501fffe 当前指令代码

```

重新跳回分支指令，两者相等，进入下一条指令（pc+4）。

地址	汇编程序	指令代码				16 进制数代码
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	
0x00000020	slti \$6,\$2,8	011011	00010	00110	0000 0000 0000 1000	6C460008









比较 8 和 2 号寄存器内容，后者更小，ALU 运算结果为 1

```

CurrentAddress[31:0] 00000020 当前指令地址
NextAddress[31:0] 00000024 下一条指令地址
PCSrc[1:0] 0
ReadData1[31:0] 00000002 2号寄存器的值
ReadData2[31:0] 00000000
ALUResult[31:0] 00000001 ALU运算结果
ALUOp[2:0] 5
_dataA[31:0] 00000002 ALU操作数 2 和 8
_dataB[31:0] 00000008
ExtendOut[31:0] ZZZZZZZZ
DataOut[31:0] ZZZZZZZZ
IDataOut[31:0] 6c460008 当前指令代码








```


运算结果 0 写入 7 号寄存器










 _rs[4:0]	07	读取7号 1 号寄存器
 _rt[4:0]	01	
 _rd[4:0]	1f	
 _sa[31:0]	0000001f	
 _opcode[5:0]	30	
 _PCPlusFourAddress[31:0]	00000030	
 _immediate[15:0]	ffffe	-2符号位扩展
 _ExtendResult[31:0]	fffffffe	

跳转到0x0000028指令，\$7内容变大，继续进入分支指令。

 _CurrentAddress[31:0]	00000028	当前指令地址
 _NextAddress[31:0]	0000002c	下一条指令分支
 _PCSrc[1:0]	0	
 _ReadData1[31:0]	00000008	
 _ReadData2[31:0]	00000008	
 _ALUResult[31:0]	00000010	
 _ALUOp[2:0]	0	ALU运算结果8+8 = 16
 _dataA[31:0]	00000008	
 _dataB[31:0]	00000008	
 _ExtendOut[31:0]	ZZZZZZZZ	
 _DataOut[31:0]	ZZZZZZZZ	
 _IDataOut[31:0]	04e70008	

 _WriteReg[4:0]	07	7号寄存器存入 16
 _WriteData[31:0]	00000010	
 _rs[4:0]	07	
 _rt[4:0]	07	
 _rd[4:0]	00	
 _sa[31:0]	00000000	
 _opcode[5:0]	01	

指令顺序执行，下一条指令地址PC+4.

 _CurrentAddress[31:0]	0000002c	
 _NextAddress[31:0]	00000030	
 _PCSrc[1:0]	0	
 _ReadData1[31:0]	00000008	
 _ReadData2[31:0]	00000008	
 _ALUResult[31:0]	00000000	
 _ALUOp[2:0]	5	两者相等 跳到PC+4地址
 _dataA[31:0]	00000008	
 _dataB[31:0]	00000008	

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
0x00000030	sw \$2,4(\$1)	100110	00001	00010	0000 0000 0000 0100	98220004

将 2 号寄存器内容存入数据存储器中位置为 4 + \$1 的地方

















_CurrentAddress[31:0]	00000030	
_NextAddress[31:0]	00000034	
_PCSrc[1:0]	0	
_ReadData1[31:0]	00000008	
_ReadData2[31:0]	00000002	1号寄存器的值
_immediate[15:0]	0004	立即数扩展
_ExtendResult[31:0]	00000004	
_WriteReg[4:0]	02	
_WriteData[31:0]	0000000c	写入寄存器2号

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
0x00000034	lw \$9,4(\$1)	100111	00001	01001	0000 0000 0000 0100	9C290004

从数据存储器中取出位置为 4+\$1 的值，并且写入 9 号寄存器

_clk	0	
_reset	1	
_zero	0	
_CurrentAddress[31:0]	00000034	
_NextAddress[31:0]	00000038	lw 语句
_PCSrc[1:0]	0	
_ReadData1[31:0]	00000008	
_ReadData2[31:0]	ZZZZZZZZ	
_immediate[15:0]	0004	
_ExtendResult[31:0]	00000004	
_WriteReg[4:0]	09	数据写入9号寄存器
_WriteData[31:0]	02000000	
_ALUResult[31:0]	0000000c	获取数据存储器中的数据位置
_ALUOp[2:0]	0	
_dataA[31:0]	00000008	

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

  _CurrentAddress[31:0]	00000038	
  _NextAddress[31:0]	00000000	跳回初始地址
  _PCSrc[1:0]	2	
  _ReadData1[31:0]	00000000	
  _ReadData2[31:0]	00000000	
  _immediate[15:0]	0010	
  _ExtendResult[31:0]	00000010	
  _WriteReg[4:0]	00	

	00000038	0000003e	00000068	XXXXXXXX
	0000003e	00000068	XXXXXXXX	
	0		1	

在不同指令中表示操作数的含义和指令位置值得注意,从指令寄存器中取出指令代码时需要了解其位置之后截取数据送入其他模块。(本实验选择在数据通路顶层模块进行各字段的分离)。

再来就是必须清楚指令的从产生到结束的过程:从指令存储器中取出指令,对指令进行译码(分离出rs, rt, rd immediate, sa等字段),执行指令(进入ALU运算单元还是指示下一段地址),访问存储器(可能是来自ALU运算结果也可能是来自寄存器组的值),将结果写回寄存器中。在这个过程中需要明白每个控制信号对于模块功能的影响。InsMemRW控制指令存储器的读写;RegWre控制寄存器组写使能;ALUSrcA, ALUSrcB控制进入ALU的数据,PCSrc控制指令的下一步的地址;RegDst控制rt字段写入寄存器组;ALUOp字段控制ALU运操作;ExtSel控制0符号位的扩展;mRD,mWR控制数据存储器的读与写,DBDataSrc控制数据来自ALU的运算结果或者数据存储器的结果(某些指令不需要访问数据存储器如add, sub等);PCWre控制PC程序计数器是否计数或者停机。了解以上控制信号的作用并熟记,在编写代码的过程中会有挺大的好处,由于对指令产生到结束的过程有了较为深入的了解,写代码不至于太茫然。

接下来进入写代码的部分了。由于Verilog之前接触得少,所以需要花费一些时间来弄清楚语法和结构。由于模块化的思想,每个模块文件的内容比较相似,大体上就是先声明一个模块,包括模块名和模块端口,然后具体定义这些端口和变量。声明端口可以在模块声明的括号内声明其为input或者output端口,一旦在模块声明里定义了变量类型,就无需在定义部分再次定义,这会导致变量的重定义。由于一开始是先在Sublime编辑器里写各个模块大致的代码,没有及时意识到重定义的问题,导致后面出现了许多不必要的错误。然后就是各个变量的声明和定义,对于一些需要时钟或者信号触发的模块需要使用always @(触发信号)句子来包含代码,在里面定义功能,如算术操作或者指令地址的变化。

在写代码的过程中一开始比较难体会的就是程序地址的变化,主要还是因为没有做一个明确的总结。简单总结以下, 1. $PC+4$ 是用于顺序执行的指令,由于指令32位,占据四个字节,相邻指令相差4。2. $PC+4 + (\text{immediate} \ll 2)$ 主要用于beq分支跳转指令,立即数指的是 $PC+4$ 地址到要跳转地址之间间隔的指令数,也就是说beq \$2,\$3,4,如果rs = rt那么跳转到当前指令之后的第五条指令,如果立即数是负数如-2则跳转到上一条指令(因为 $PC+4$ 之后在指令地址跳到下一条指令了,所以-2之后是退回上一条指令)。3. 扩展地址。来自 j address 指令。其中的address字段是25位,需要扩展为32位,扩展原理一开始并

不是很懂，有点照搬老师提供公式的意味，后来经过仔细揣摩明白了，地址的长度为4个字节是4的倍数，所以最低位的两个数字都为0，j 指令中的26位地址只是原始地址的精简表达（省去了最低两位0，本来是28位）这样加上PC+4的最高四位就得到真正完整的跳转地址。

总结一下写代码过程中遇到的问题。

在构建各个模块的时候，一开始没有考虑好模块之间变量的配合，比如信号量的端口长度没有注意好，模块之间的输入输出值没有很好的呼应，导致部分代码重新修改了好多次，缺乏整体性的思想，下次实验得这方面多花功夫。

在代码编写检测方面，一开始以为用代码编辑器会比较快捷一点，但后来导入文件的时候发现vivado自带的语法检测功能挺方便的，检测出了很多语法错误，这挺浪费时间的。下次要直接创建设计文件在vivado中提高效率。

代码规范方面，由于编写代码时间比较分散（这一点挺致命，下次得心无旁骛，远离手机和其他诱惑），各个模块变量的命名没有很好的统一，导致后来在顶层模块中统一连接每个模块的时候，出现了挺多耽搁。其实变量的命名，已经在数据通路图中体现得很明显了，无需按照自己的想法重新定义名称，这一点也是踩了坑之后才有所体会的，这个命名规范不能太随意，不仅仅是在本次实验，在其他代码工程中都要注意变量风格和含义的统一。同时还要及时在一些核心代码中著名代码的功能，以及自己部分定义变量的含义，这一点无论是对于他人的阅读或者自己的阅读都是极为提高好感的做法，今后一定要勤写注释，写简明易懂的注释。

整个工程的信息量挺大的，涵盖了计算机，数字电路方面的许多元件，ALU, regFile, Instruction memory 和 Data memory等等，并将这些元件有机的统一起来，整个学习流程对以后更复杂的硬件设计有了更好的基础和认识，在其他编程领域也有了一些领悟和借鉴意义。