



# 《计算机组成原理与接口技术实验》 实验报告

学 院 名 称 : 数据科学与计算机学院

学 生 姓 名 : 黄羽盼

学 号 : 14331106

专业（班级）: 14 软件工程三（6）班

合 作 者 : 蒋子涵

时 间 : 2016 年 4 月 25 日

成绩：

## 实验二：单周期CPU设计

## 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。需设计的指令与格式如下：

## ==&gt; 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： $rd \leftarrow rs + rt$ 。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： $rt \leftarrow rs + (\text{sign-extend})\text{immediate}$ ；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
--------	---------	---------	---------	----------

完成功能： $rd \leftarrow rs - rt$

## ==&gt; 逻辑运算指令

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

010000	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。

(5) **and rd, rs, rt**

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

功能： $rd \leftarrow rs \& rt$ ；逻辑与运算。

(6) **or rd, rs, rt**

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

功能： $rd \leftarrow rs \mid rt$ ；逻辑或运算。

## ==&gt; 传送指令

(7) **move rd, rs**

100000	rs(5 位)	00000	rd(5 位)	reserved
--------	---------	-------	---------	----------

功能： $rd \leftarrow rs + \$0$ ； $\$0 = \$zero = 0$ 。

## ==&gt; 存储器读/写指令

(8) **sw rt, immediate(rs) 写存储器**

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

功能:  $\text{memory}[\text{rs} + (\text{sign-extend})\text{immediate}] \leftarrow \text{rt}$ ; **immediate** 符号扩展再相加。

(9) **lw rt, immediate(rs)** 读存储器

100111	rs(5 位)	rt(5 位)	<b>immediate</b> (16 位)
--------	---------	---------	-------------------------

功能:  $\text{rt} \leftarrow \text{memory}[\text{rs} + (\text{sign-extend})\text{immediate}]$ ; **immediate** 符号扩展再相加。

==> 分支指令

(10) **beq rs,rt,immediate**

110000	rs(5 位)	rt(5 位)	<b>immediate</b> (位移量, 16 位)
--------	---------	---------	------------------------------

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

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”。

==> 停机指令

(11) **halt**

111111	0000000000000000000000000000(26 位)
--------	------------------------------------

功能: 停机; 不改变 PC 的值, PC 保持不变。

### 三. 实验原理

简述实验原理和方法, **必须有原理结构图**。

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。

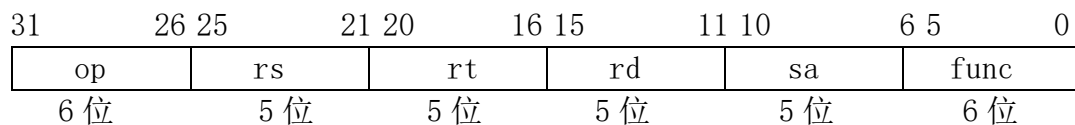
单周期 CPU, 是在一个时钟周期内完成这五个阶段的处理。



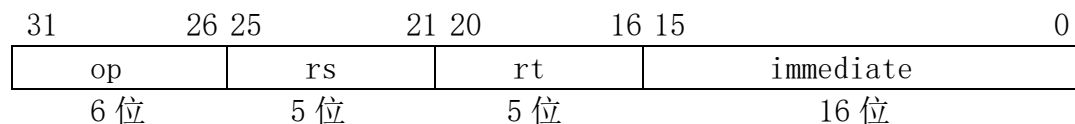
图 1 单周期 CPU 指令处理过程

MIPS32 的指令的三种格式：

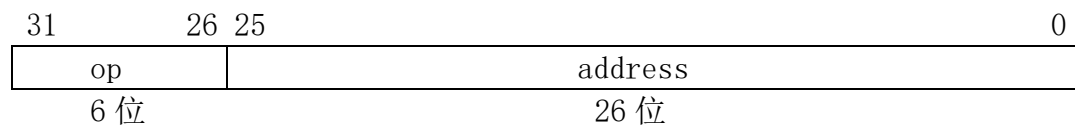
**R 类型：**



**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；

**func:** 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。

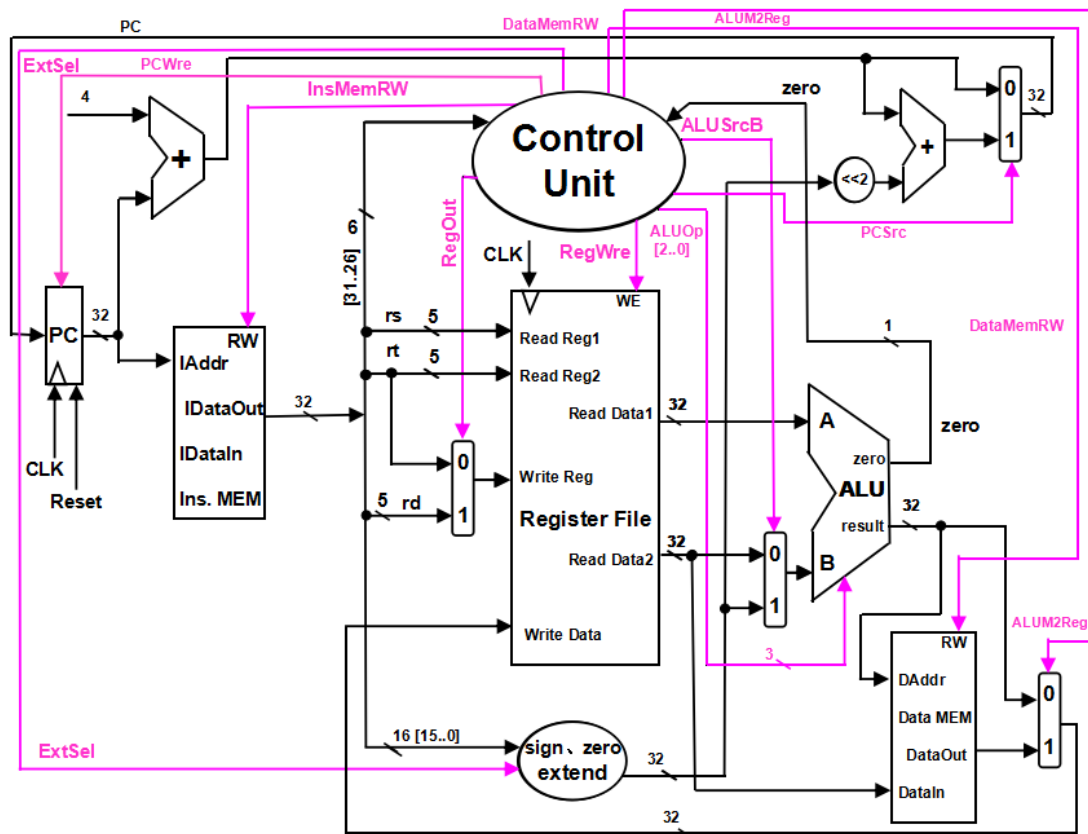


图 2 单周期 CPU 数据通路和控制线路图

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，先给出地址，然后由读/写信号控制（1-写，0-读。当然，也可以由时钟信号控制，但必须在图上标出）。对于寄存器组，读操作时，先给出地址，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，在时钟边沿触发写入。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”
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、move、beq	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、sw、lw
ALUM2Reg	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、move	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：sw、halt	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、move、lw
InsMemRW	读指令存储器(Ins. Data)，初始化为 0	写指令存储器
DataMemRW	读数据存储器，相关指令：lw	写数据存储器，相关指令：sw

ExtSel	相关指令: ori, (zero-extend) <b>immediate</b> (0 扩展)	相关指令: addi、sw、lw、beq, (sign-extend) <b>immediate</b> (符号扩展)
PCSrc	PC←PC+4, 相关指令: add、sub、ori、or、and、move、sw、lw、beq(zero=0)	PC←PC+4+(sign-extend) <b>immediate</b> , 同时 zero=1, 相关指令: beq
RegOut	写寄存器组寄存器的地址, 来自 rt 字段, 相关指令: addi、ori、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、move
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 1 写, 为 0 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
- RW, 数据存储器读写控制信号, 为 1 写, 为 0 读

**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, 在时钟上升沿写入

**ALU:**

- result, ALU 运算结果
- zero, 运算结果标志, 结果为 0 输出 1, 否则输出 0

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	A + B	加
001	A - B	减
010	B - A	减
011	A ∨ B	或
100	A ∧ B	与
101	/A ∧ B	A 非与 B
110	A ⊕ B	异或

111	A ⊙ B	同或
-----	-------	----

表 3 控制信号与指令的关系表

指令	控制信号										
	Z	PCWre	ALUSrcB	ALUM2Reg	RegWre	InaMemRW	DataMemRW	ExtSel	PCSrc	RegOut	ALUOp[2..0]
add	x	1	0	0	1	x	x	x	0	1	000
addi	x	1	1	0	1	x	x	1		0	000
sub	x	1	0	0	1	x	x	x	0	1	001
ori	x	1	1	0	1	x	x	0	0	0	011
and	x	1	0	0	1	x	x	x	0	1	100
or	x	1	0	0	1	x	x	x	0	1	011
move	x	1	0	0	1	x	x	x	0	1	000
sw	x	1	1	x	0	x	1	1	0	x	000
lw	x	1	1	1	1	x	0	1	0	0	000
beq	0	1	0	x	0	x	x	1	0	x	001
	1	1	0	x	0	x	x	1	1	x	001
halt	x	0	x	x	x	x	x	x	x	x	xxx

需要说明的是根据要实现的指令功能要求画出以上数据通路图，和确定 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(表三)，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。

具体来说，以上指令相应的变量假定是这样表示的，指令助记符前加“i”，如“add”，则变量为“i\_add”，以此类推。指令代码 32 为长度，最高 6 位为操作码，唯一标识一条汇编指令，如规定操作码 op=“000000”为加法指令，即助记符 add。如果从指令存储器中读出来的指令代码，其操作码为“000000”，则规定在控制器中 i\_add=1，否则 i\_add=0。以此类推。

以上‘z’表示标志位，运算结果为 z=1，否则 z=0，以下用 zero 表示。

逻辑表达式：

```
ALUSrcB=i_addi | i_ori | i_sw | i_lw
.....
RegWre=i_add | i_addi | i_sub | i_ori | i_and | i_or | i_move | i_lw
.....
ALUOp[2]=i_and
ALUOp[1]=i_ori | i_or
ALUOp[0]=i_sub | i_ori | i_or | i_beq
.....
```

当然，也可以这样考虑控制单元的子模块内容：

```
module ControlUnit(
    opcode,
    Zero,
    RegWre,
```

```

    PCWre,
    ALUSrcB,
    ALUOp,
    ALUM2Reg,
    RegOut,
    DataMemRW,
    PCSrc,
    ExtSel
);

    input [5:0] opcode, zero; //指令操作码
    output reg RegWre, PCWre, ALUSrcB, ALUM2Reg, RegWre, DataMemRW,
PCSrc, ExtSel;
    output reg [2:0] ALUOp;

    //若 opcode(指令操作码)有变化，都会触发以下部分产生新的控制信号
    // 如果 beq 指令时，PCSrc=zero。
    always@( opcode) begin
        case( opcode )
            // add, R-format
            6'b000000:
                Begin    //以下都是控制单元产生的控制信号
                    RegWre = 1;
                    PCWre = 1;        // PC 写使能
                    ALUSrcB = 0;      // ALU B 口数据选择控制
                    ALUOp = 000;      // ALU 功能码
                    ALUM2Reg = 0;     // ALU 或存储器的数据送往寄存器
                    RegWre = 1;       // 寄存器写使能
                    DataMemRW = 0;    // 1 为写，0 为读
                    PCSrc = 0;        // PC 转移控制
                    ExtSel = 0;       // immediate 扩展控制
                end
            // sub, R-format
            6'b000001:
                begin
                    RegWre = 1;
                    PCWre = 1;
                    ALUSrcB = 0;
                    ALUOp = 001;
                    ALUM2Reg = 0;
                    RegWre = 1;
                    DataMemRW = 0;
                    PCSrc = 0;
                    ExtSel = 0;
                end
        endcase
    end

```



```
                end
.....
            end
endmodule
```

指令执行的结果总是在下个时钟到来前开始保存到寄存器、或存储器中，PC 的改变也是在这个时候进行。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 ISE 软件一套。

#### 五. 实验分析与设计

##### 1. 实验步骤

- (1) 阅读课件及提供的资料，了解本次实验的内容。
- (2) 对实验进行分析（见2. 分析）
- (3) 对实验进行设计（见3. 设计）
- (4) 编写各子模块代码
- (5) 编写测试代码（见4. 测试单周期CPU）
- (6) debug（见心得和体会部分的 1. 实验过程中遇到的问题及解决的办法）
- (7) 反思总结写报告

##### 2. 分析

中央处理器（CPU）是计算机取指令和执行指令的部件，它由算术逻辑单元（ALU）、寄存器和控制器组成，是计算机系统的核心部件。

CPU设计的第一步应当根据指令系统来建立数据路径，再定义各个部件的控制信号，确定时钟周期，完成控制器的设计。进而可以进行数字设计、电路设计，最后完成物理实现。本实验重点是单周期数据路径的建立和组合逻辑实现的控制器。

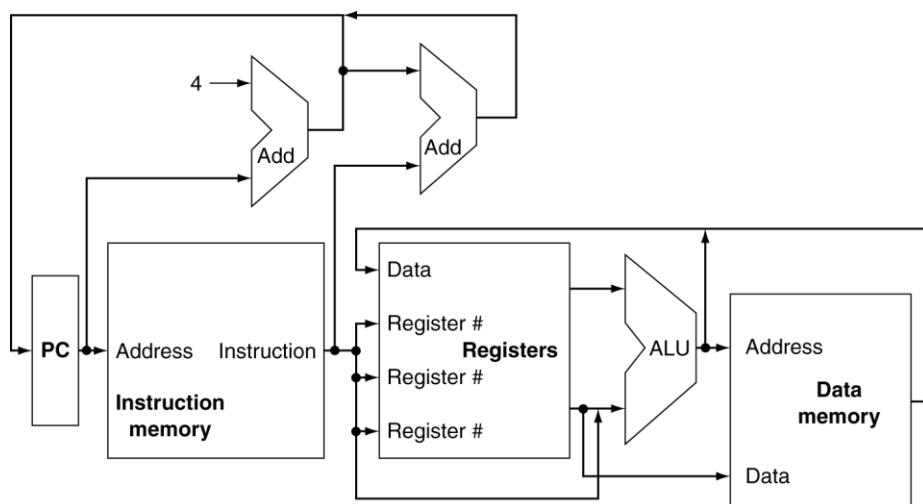


图 3 CPU 粗略概览

实验原理中已经详细阐述了本实验的主要原理依据，用我自己的话来分析，这个实验需要设计一个硬件电路，实现从存储器中读取指定的指令并解析指令从而调动各个模块执行相应的操作。这些指令的执行在一个时钟周期内完成，因此是单周期CPU。要实现这个设计，需要知道CPU是怎样处理指令的。首先是取指令（PC，存储器），然后需要对指令译码，即剖析对应二进制数代表的操作和含义，此时需要产生相应的操作控制信号来告诉各个模块要做什么（控制单元），下一步就是执行了。

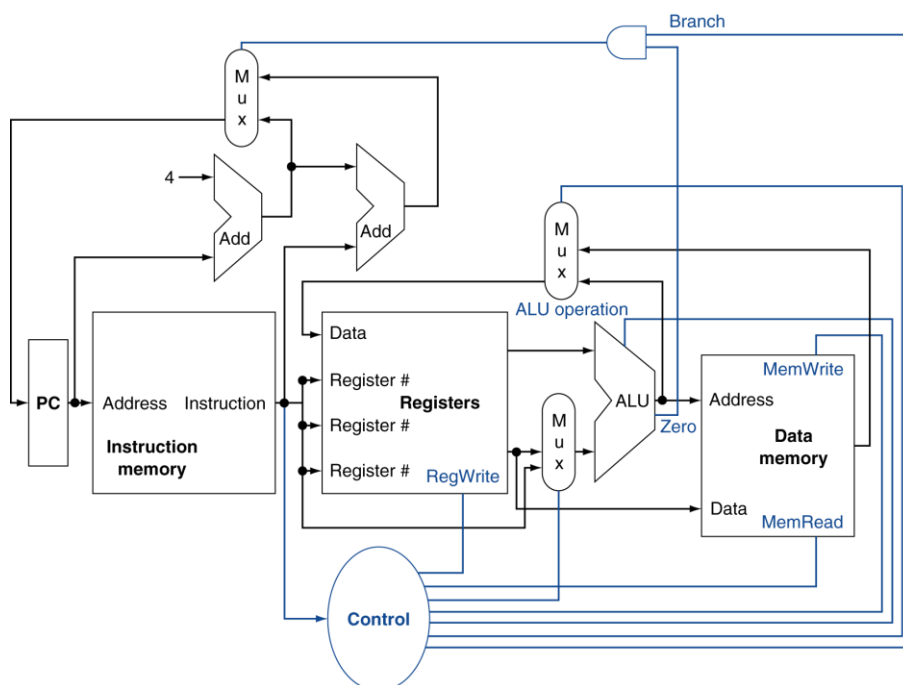


图 4 控制单元的控制信号

整个过程中需要对数据进行操作，而数据在寄存器或存储器中，不同的指令对数据的处理不一样，有的要读有的要写。因此需要分析不同类型指令的特点和异

同导致的数据传输路径和控制信号的不同，可以先画出它们各自的数据通路图，再结合起来。数据通路指CPU中处理数据和地址的流动路径。各子模块的数据通路如下。

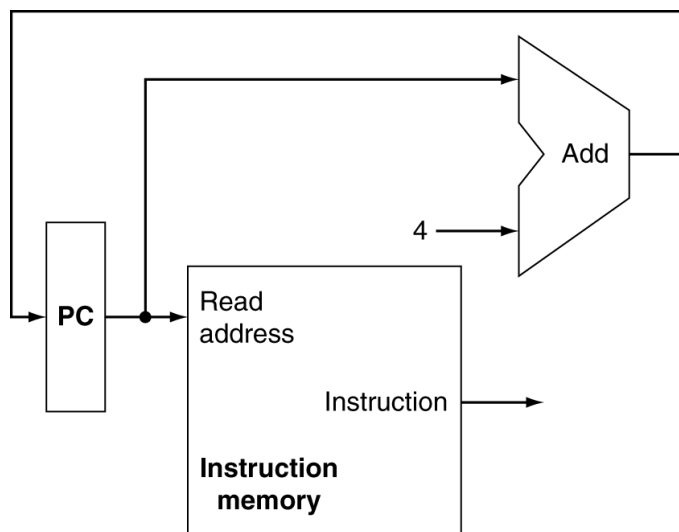


图 5 取指令

对于R型指令，从寄存器读取两个操作数送入逻辑运算单元进行运算并把结果写回寄存器。

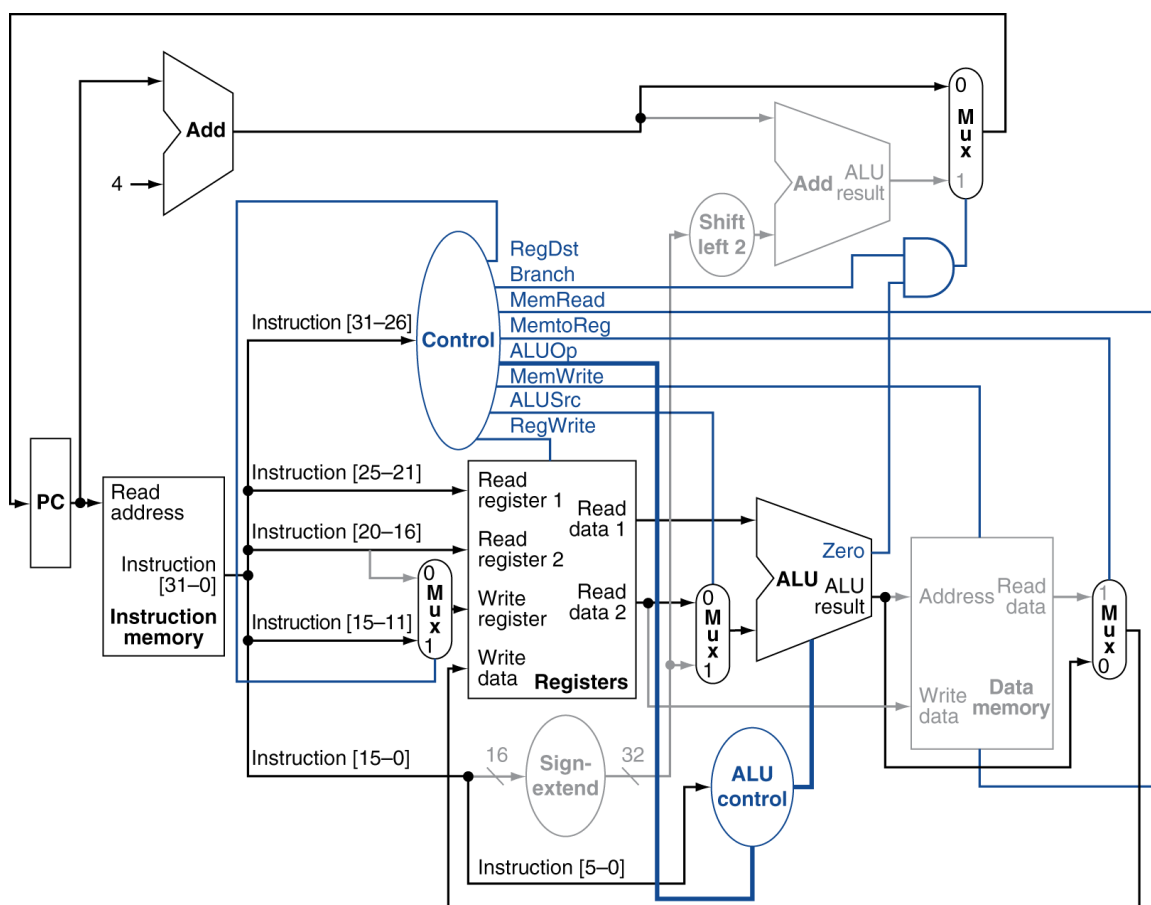


图 6 R 型指令数据通路

对于load/store指令，都要先从寄存器读取一个操作数送入ALU与另一个从16位立即数经符号扩展到32位的数进行运算，如果是load指令，则运算结果作为地址进入数据存储器寻址写回寄存器；如果是store指令，则将寄存器的写入数据存储器的寻址后的地方。

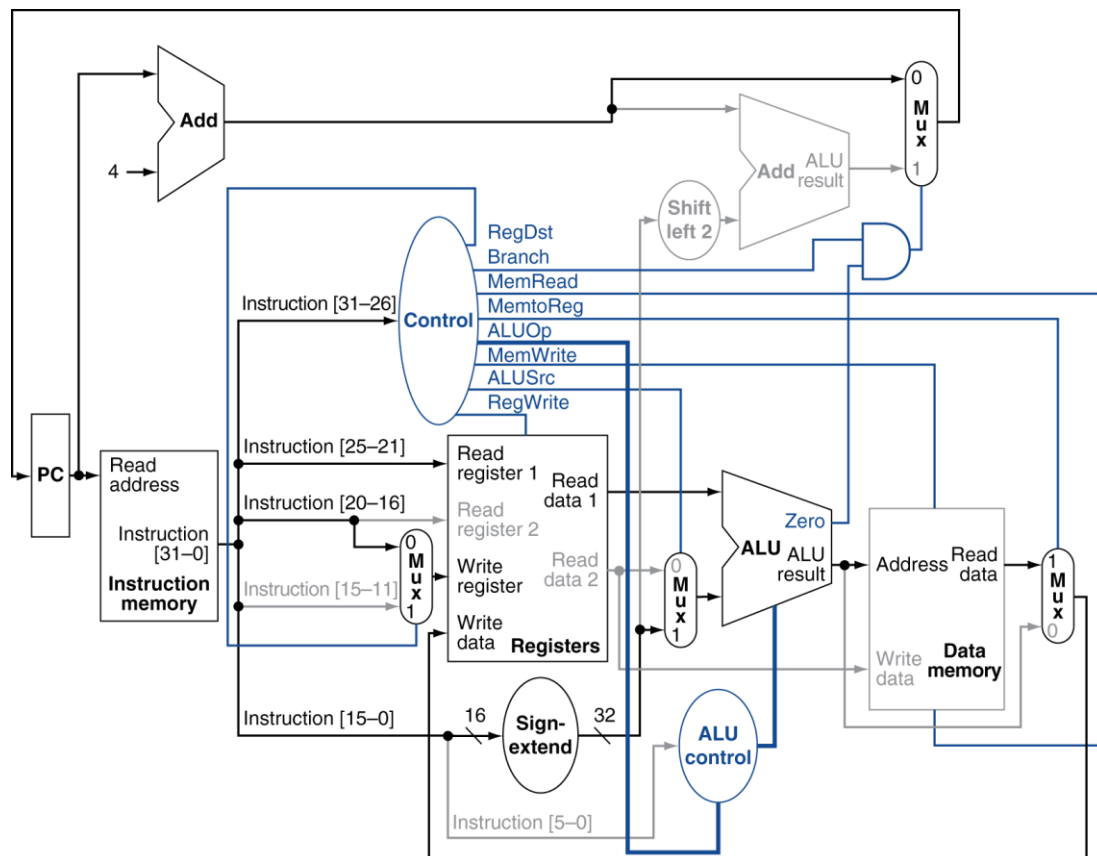
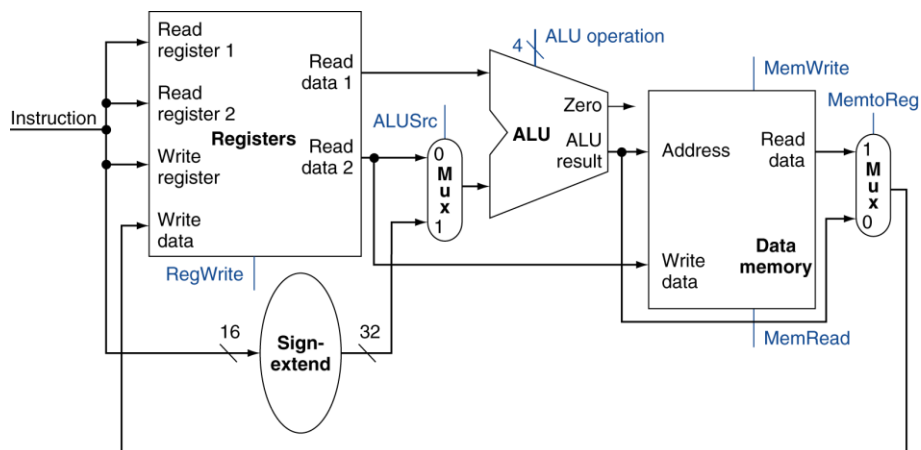


图 7 Load 指令的数据通路

对于跳转指令，先读取寄存器的操作数，再用ALU进行比较（做减法运算，并用zero来标志结果是否为0）并分别对应两种情况。



**图 8 R-Type/Load/Store Datapath**

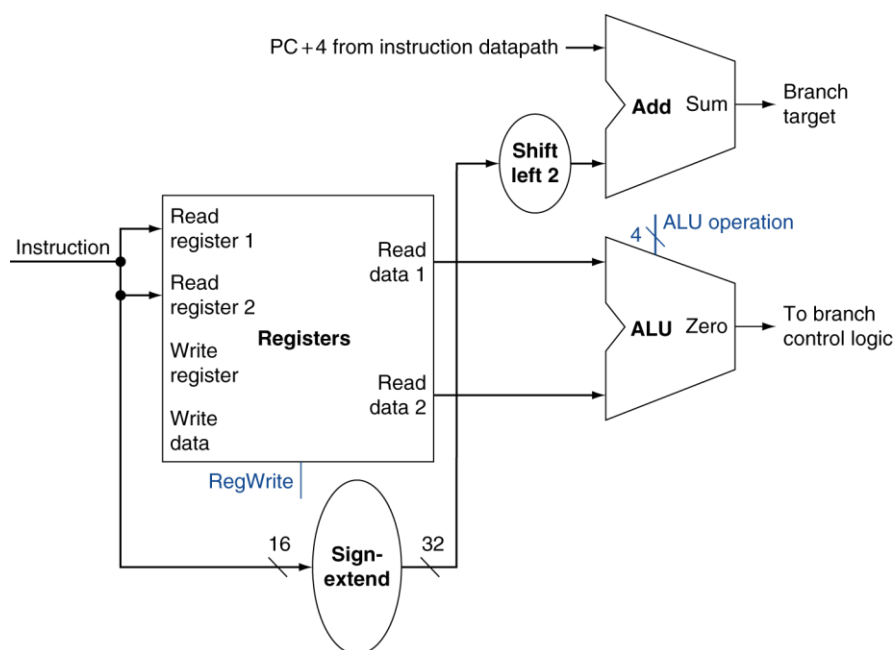


图 9 branch instruction

将各数据通路整合即可得图2的总的单周期CPU数据通路图。并可以分析出控制信号与指令的关系表（实验原理中有详细说明）。

### 3. 设计

用模块化的思想方法设计。

查阅相关资料（Verilog模块概念和实例化<sup>1</sup>）可知，模块（module）是verilog最基本的概念，是v设计中的基本单元，每个v设计的系统中都由若干module组成。具体来说，模块有以下性质：

- 1) 模块在语言形式上是以关键词module开始，以关键词endmodule结束的一段程序。
- 2) 模块的实际意义是代表硬件电路上的逻辑实体。
- 3) 每个模块都实现特定的功能。
- 4) 模块的描述方式有行为建模和结构建模之分。
- 5) 模块之间是并行运行的。
- 6) 模块是分层的，高层模块通过调用、连接低层模块的实例来实现复杂的功能。
- 7) 各模块连接完成整个系统需要一个顶层模块（top-module）。

由于单周期CPU不是一个非常简单的系统，而无论多么复杂的系统，总能划分成多

<sup>1</sup> [http://jason0214.lofter.com/post/30cbe4\\_12a8f72](http://jason0214.lofter.com/post/30cbe4_12a8f72)

个功能模块。因此我按照下面四个步骤进行单周期CPU的设计：

1) 把系统划分成子模块;

由于单周期CPU的设计涉及到了很多小部件，如加法器、二选一数据选择器、有无符号数扩展等，为了使设计起来简洁明了、打代码时思路简单，我选择将系统尽可能分解成小模块。比如单独定义二选一数据选择器的模块而不是将它们直接通过分析逻辑关系和旁边的部件组合起来。但是由于PC的操作是逻辑清晰而连贯的，模块PC可以整合PC+4及遇到跳转指令时  $pc \leftarrow pc + 4 + (\text{sign-extend})\text{immediate} \ll 2$  的功能，而不再细分左移、加法器等模块。具体分出的模块见图10、图11。

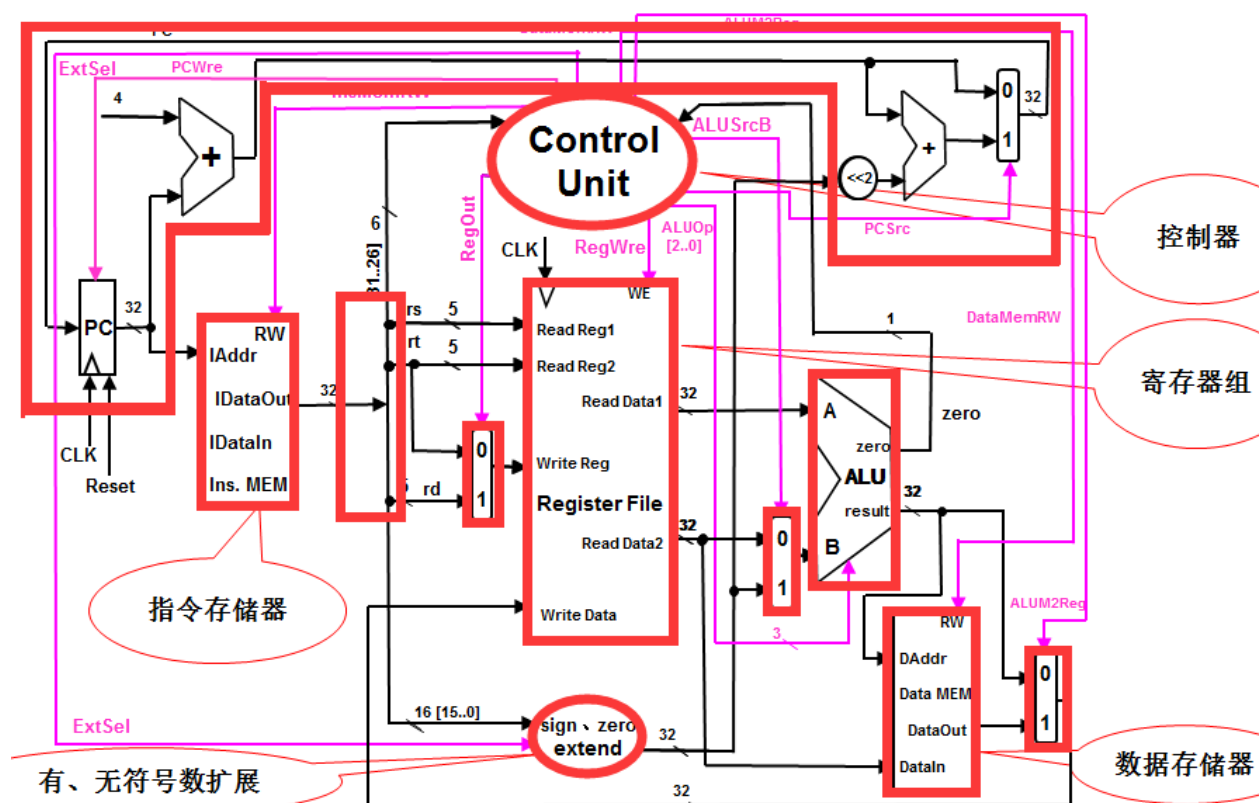


图 10 子模块划分

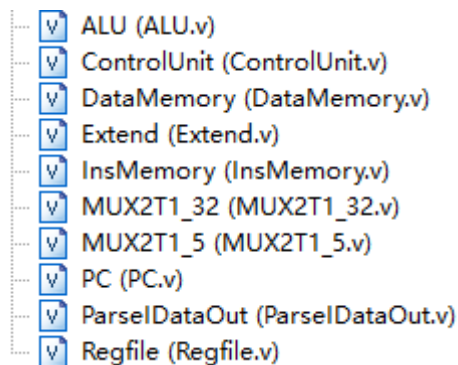


图 11 子模块名称

其中,模块MUX2T1\_32是32位的二选一数据选择器,MUX2T1\_64则是32位的。而模块ParseDataOut是将指令寄存器输出32位的指令分解为6位的Opcode, 5位的rs,rt,rd和16位的immediate。

2) 规定各模块的接口;

各模块接口直接采用数据通路图中的名称, 统一规范。

3) 进行子模块设计

a) ALU的设计

在表2 (ALU运算功能表列出的8个功能中, 本实验只用到了前5个, 因此只写前5个。通过输入的ALUOp来判断应当执行哪个功能。最后还要判断结果是否等于0, 从而为标志位Zero赋值, 输出Zero作为控制单元的输入用来控制beq是否发生跳转。

b) 存储器设计

数据存储器: 考虑读和写。

指令存储器: 本设计只考虑读不考虑写。

它们内部均有一个数组存储数据, 每次读或写都进行更新。

c) 寄存器组设计

寄存器组有32个寄存器, 每个寄存器用32位的D触发器实现。



图 12 D 触发器

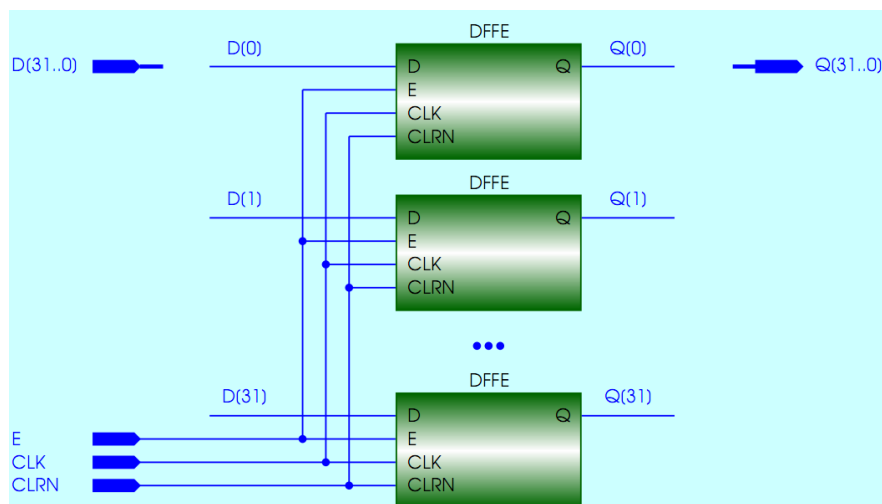


图 13 D 触发器组

寄存器堆有两个读端口，一个写端口，见图14.

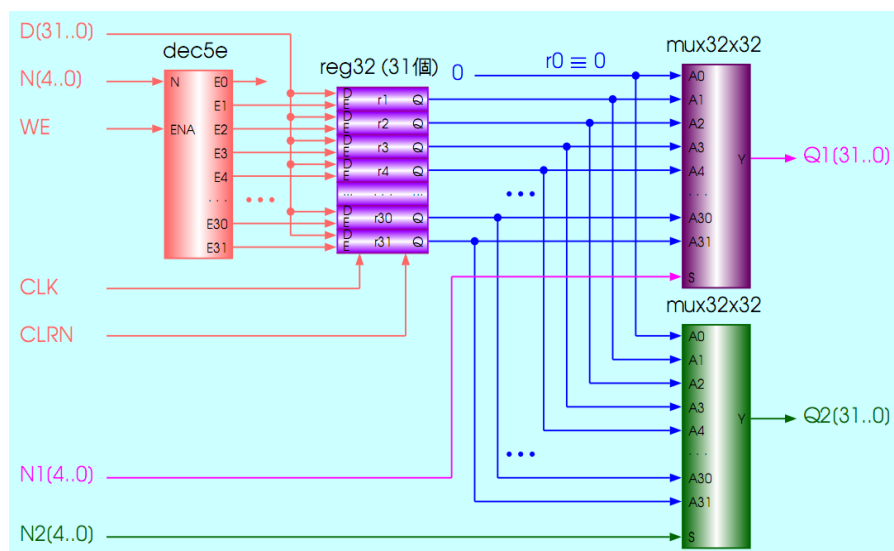


图 14 寄存器堆的结构描述

#### 4) 对模块编程;

//Control Unit的部分代码在实验原理部分已列出，不再赘述

//

```
module ALU(A, B, Zero, result, ALUOp);
```

```
    input [31:0] A, B;
```

```
    input [2:0] ALUOp;
```

```
    output reg Zero;
```

```
    output reg [31:0] result;
```

```
    always @(A or B or ALUOp or result) 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;
    endcase
    if (result == 32'b0) Zero = 1;
    else Zero = 0;
end
endmodule
/////////////////////////////////////////////////////////////////
module DataMemory(RW, DAddr, DataIn, DataOut, CLK);
    input [31:0] DAddr, DataIn;
    input RW, CLK;
    output reg[31:0] DataOut;
    reg [31:0] memory[0:511];

    always @(negedge CLK) begin
        if (RW == 1) memory[DAddr] = DataIn; // write
        else DataOut = memory[DAddr]; // read
    end
endmodule
/////////////////////////////////////////////////////////////////
module Extend(bits_16, ExtSel, bits_32);
    input [15:0] bits_16;
    input ExtSel;
    output reg [31:0] bits_32;

    always@(ExtSel) begin
        // zero extent
        if(ExtSel == 0) assign bits_32 = {16'h0000, bits_16};
        // sign extent
        else assign bits_32 = bits_16[15] ? {16'hffff, bits_16} : {16'h0000,
bits_16};
    end
endmodule
/////////////////////////////////////////////////////////////////
module InsMemory(IAddr, IDataOut);
    // 本设计中指令寄存器只读不写，RW=InsMemRW=0
    input [31:0] IAddr;
    output reg[31:0] IDataOut;
    reg [31:0] memory[0:511];

    initial
        $readmemb("instructionData.txt", memory, 32'h100);
    always @(IAddr) begin

```

```

        IDataOut = memory[32'h100 + (IAddr - 32'h100) / 32'd4]; // read
    end
endmodule
/////////////////////////////////////////////////////////////////
module MUX2T1_32(i0, i1, out, select);
    input [31:0] i0;
    input [31:0] i1;
    output reg [31:0] out;
    input select;

    always @(i0 or i1 or select) begin
        if(select == 0) out = i0;
        else out = i1;
    end
endmodule
/////////////////////////////////////////////////////////////////
module MUX2T1_5(i0, i1, out, select);
    input [4:0] i0;
    input [4:0] i1;
    output reg [4:0] out;
    input select;

    always @(i0 or i1 or select) begin
        if(select == 0) out = i0;
        else out = i1;
    end
endmodule
/////////////////////////////////////////////////////////////////
module PC(pcln, CLK, Reset, PCSrc, PCWre, pcOut);
    input [31:0] pcln;
    input CLK;
    input Reset;
    input PCSrc;
    input PCWre;
    output reg [31:0] pcOut;
    reg [31:0] pc;
    always @(posedge CLK or posedge Reset) begin
        if (Reset == 1) begin
            // reset
            pcOut = 32'h100;
            pc = 32'h100;
        end
        else if (PCWre != 0) begin
            if(PCSrc == 0) pcOut = pc + 4;
        end
    end
endmodule

```

```

        else pcOut = pc + 4 + (pcIn << 2);
        pc = pcOut;
    end
end
endmodule
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module ParseIDataOut(iDataOut, Opcode, rs, rt, rd, immediate);
    input [31:0] iDataOut;
    output reg [5:0] Opcode;
    output reg [4:0] rs, rt, rd;
    output reg [15:0] immediate;
    always @(iDataOut) begin
        Opcode = iDataOut[31:26];
        rs = iDataOut[25:21];
        rt = iDataOut[20:16];
        rd = iDataOut[15:11];
        immediate = iDataOut[15:0];
    end
end
endmodule
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module Regfile(
    ReadReg1, ReadReg2, WriteData, WriteReg, WE, CLK, clrn, ReadData1,
    ReadData2);
    input [4:0] ReadReg1, ReadReg2, WriteReg;
    input [31:0] WriteData;
    input WE, CLK, clrn;
    output [31:0] ReadData1, ReadData2;
    reg [31:0] register[1:31]; //r1-r31
    integer i; // 注意要在外面声明
    assign ReadData1 = (ReadReg1 == 0) ? 0 : register[ReadReg1]; //read
    assign ReadData2 = (ReadReg2 == 0) ? 0 : register[ReadReg2]; //read
    always @(posedge CLK or negedge clrn) begin
        if (clrn == 1) begin
            // reset
            for(i=1;i<31;i=i+1)
                register[i] <= 0;

        end
        else begin
            // write
            if((WriteReg!=0) && (WE == 1))
                register[WriteReg] <= WriteData;
        end
    end
end
end

```

endmodule

#### 5) 连接各模块完成系统设计。

verilog是通过模块调用或称为模块实例化的方式来实现这些子模块与高层模块的连接。调用模块实例的一般形式为：〈模块名〉〈参数列表〉〈实例名〉(〈端口列表〉)；信号端口可通过名称关联即.PortName(port\_expr)。要注意的是引用时要严格按照模块定义的端口顺序来连接，不用标明原模块定义时规定的端口名：

Design u\_1(u\_1的端口1, u\_1的端口2, u\_1的端口3, u\_1的端口……)；

用‘.’符号来标明原模块定义时规定的端口名：

```
Design u_2(. (端口1(u_1的端口1),
              . (端口2(u_1的端口2),
              . (端口3(u_1的端口3),
              ..... ) ;
```

各子模块是通过线连接起的，由于每根线两端连接两个模块，不为单独一个模块所有，我采用为衔接的线另外命名（下图蓝色的线）。但控制单元模块输出的十个控制信号（下图枚红色）及输入的Opcode和zero含义明显，不再单独命名。所有线都写做wire [size - 1:0] wireName;两个子模块的两个端口通过一根线连接，则分别写做.PortName1(port\_expr)和.PortName2(port\_expr)。由此就可以轻松表示各模块之间的连接关系了。具体见下图：

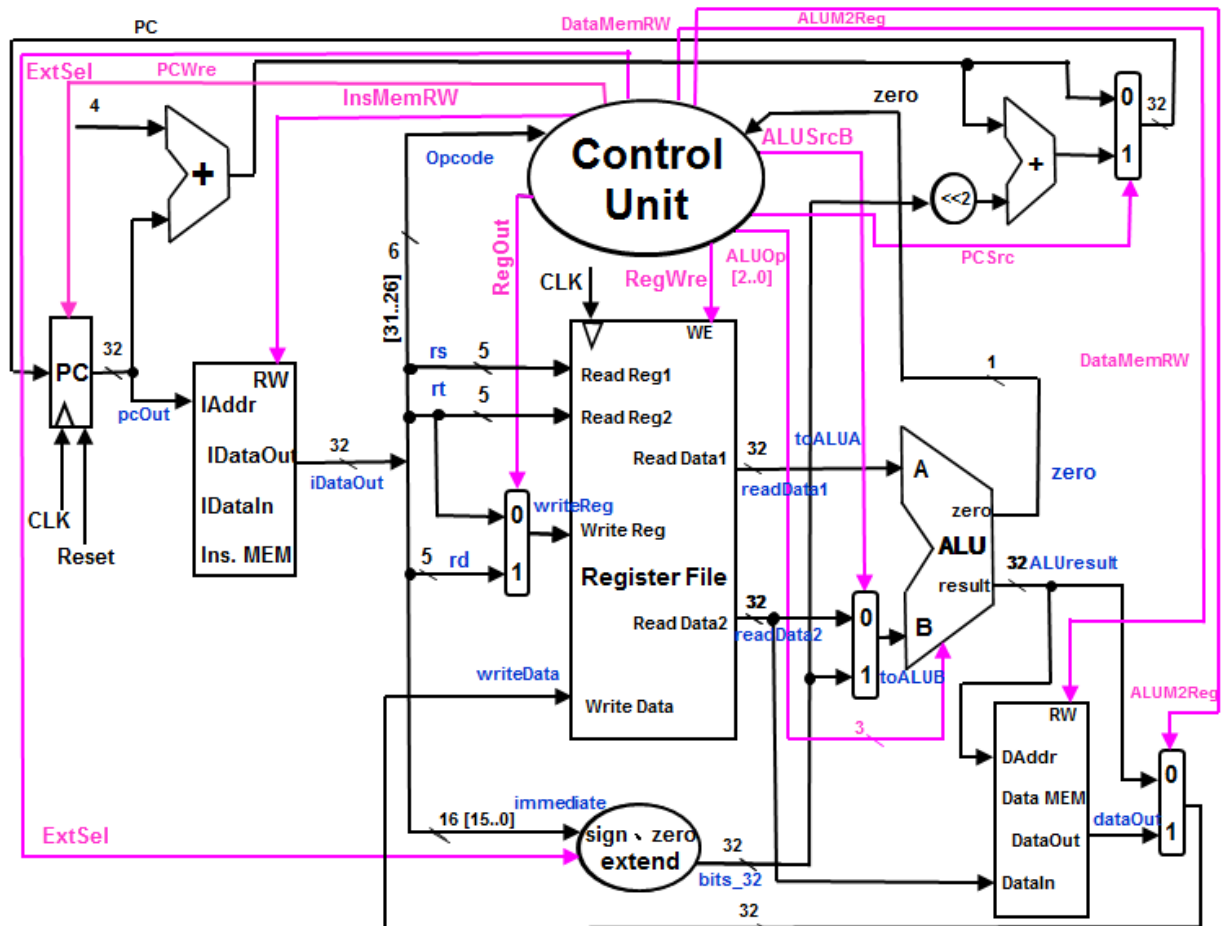


图 15 子模块的连接

代码如下：

```
module Test;
```

```
    // Inputs
    reg clk, reset;
```

```
    // Outputs
```

```
    // ControlUnit
    wire RegWre;
    wire PCWre;
    wire ALUSrcB;
    wire [2:0] ALUOp;
    wire ALUM2Reg;
    wire RegOut;
    wire DataMemRW;
    wire PCSrc;
    wire ExtSel;
```

```
    // ALU
```

```
    wire [31:0] toALUA;
    wire [31:0] toALUB;
    wire [31:0] ALUresult;
    wire Zero;
```

```
    // DataMemory
    wire [31:0] readData2;
    wire [31:0] dataOut;
```

```
    // Extend
    wire [31:0] bits_32;
```

```
    // InsMemory
    wire [31:0] pcOut;
    wire [31:0] iDataOut;
```

```
    // Regfile
    wire [4:0] writeReg;
    wire [31:0] writeData;
```

```

        .bits_16(immediate),
        .ExtSel(ExtSel),
        .bits_32(bits_32)
    );

    InsMemory insMemory (
        .lAddr(pcOut),
        .lDataOut(iDataOut)
    );

    PC pC (
        .pcIn(bits_32),
        .CLK(clk),
        .Reset(reset),
        .PCSrc(PCSrc),
        .PCWre(PCWre),
        .pcOut(pcOut)
    );

    Regfile regfile (
        .ReadReg1(rs),
        .ReadReg2(rt),
        .WriteData(writeData),
        .WriteReg(writeReg),
        .WE(RegWre),
        .CLK(clk),
        .clrn(reset),
        .ReadData1(toALUA),
        .ReadData2(readData2)
    );

    ParseIDataOut
    parseIDataOut(
        .iDataOut(iDataOut),
        .Opcode(Opcode),
        .rs(rs),
        .rt(rt),
        .rd(rd),
        .immediate(immediate)
    );

    MUX2T1_32
    mux2t1_32_DataMemory (
        .i0(ALUresult),

```

```

// ParseIDataOut
wire [4:0] rs;
wire [4:0] rt;
wire [4:0] rd;
wire [5:0] Opcode;
wire [15:0] immediate;

// Instantiate the Unit Under
Test (UUT)
ControlUnit controlUnit (
    .Opcode(Opcode),
    .Zero(Zero),
    .RegWre(RegWre),
    .PCWre(PCWre),
    .ALUSrcB(ALUSrcB),
    .ALUOp(ALUOp),

    .ALUM2Reg(ALUM2Reg),
    .RegOut(RegOut),

    .DataMemRW(DataMemRW)
,
    .PCSrc(PCSrc),
    .ExtSel(ExtSel)
);

ALU alu (
    .A(toALUA),
    .B(toALUB),
    .Zero(Zero),
    .result(ALUresult),
    .ALUOp(ALUOp)
);

DataMemory dataMemory (
    .RW(DataMemRW),
    .DAddr(ALUresult),
    .DataIn(readData2),
    .DataOut(dataOut),
    .CLK(clk)
);

Extend extend (

```

```

        .i1(dataOut),
        .out(writeData),
        .select(ALUM2Reg)
    );

    MUX2T1_32
    mux2t1_32_Regfile (
        .i0(readData2),
        .i1(bits_32),
        .out(toALUB),
        .select(ALUSrcB)
    );

    MUX2T1_5
    mux2t1_5_InsMemory (
        .i0(rt),
        .i1(rd),
        .out(writeReg),
        .select(RegOut)
    );

    initial begin
        // Initialize Inputs
        clk = 1;
        reset = 1;

        #5;
        reset = 0;
        forever #5 clk = ~clk;
    end

endmodule

```

#### 4. 测试单周期CPU

##### (1) 测试程序段

表 3 测试程序段

地址	汇编程序	指令代码					16 进制数代码	执行指令
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)			
0x00000100	ori \$1,\$0,11	010000	00000	00001	0000 0000 0000 1011	=	4001000B	\$1=0 11=11
0x00000104	addi \$2,\$1,-3	000001	00001	00010	1111 1111 1111 1101	=	0422FFFD	\$2=11-3=8
0x00000108	and \$3,\$1,\$2	010001	00001	00010	0001 1000 0000 0000	=	44221800	\$3=11&8=8
0x0000010C	or \$4,\$1,\$2	010010	00001	00010	0010 0000 0000 0000	=	48222000	\$4=11 8=11
0x00000110	add \$5,\$3,\$4	000000	00011	00100	0010 1000 0000 0000	=	00642800	\$5=19
0x00000114	move \$3,\$5	100000	00101	00000	0001 1000 0000 0000	=	80A01800	\$3=19
0x00000118	beq \$3,\$4,4 (转 012C)	110000	00011	00100	0000 0000 0000 0100	=	C0640004	\$3!= \$4
0x0000011C	sw \$5,40(\$4)	100110	00100	00101	0000 0000 0010 1000	=	98850028	Mem[51]=19
0x00000120	lw \$4,32(\$5)	100111	00101	00100	0000 0000 0010 0000	=	9CA40020	\$4=Mem[51]
0x00000124	sub \$6,\$2,\$1	000010	00010	00001	0011 0000 0000 0000	=	08413000	\$6=8-11=-3
0x00000128	beq \$3,\$4,-5(转 0118)	110000	00011	00100	1111 1111 1111 1011	=	C064FFFB	\$3=\$4=19
0x0000012C	halt	111111	00000	00000	0000000000000000	=	FC000000	停止

##### (2) 将指令代码初始化到指令存储器，直接写入。

```

module InsMemory(
    IAddr, IDataOut
);
// 本设计中指令寄存器只读不写, RW=InsMemRW=0
input [31:0] IAddr;
output reg[31:0] IDataOut;
reg [31:0] memory[0:511];

initial
    $readmemb("instructionData.txt", memory, 32'h100);
always @(IAddr) begin
    // 错误的! 每个指令顺序存储在数组中, 但指令地址是+=4, 导致取到了ori的后四条指令and
    // IDataOut = memory[IAddr];
    IDataOut = memory[32'h100 + (IAddr - 32'h100) / 32'd4]; // read
end

endmodule

```

instructionData.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```

010000000000000010000000000001011
0000010000100010111111111111101
01000100001000100001100000000000
01001000001000100010000000000000
00000000011001000010100000000000
10000000101000000001100000000000
11000000011001000000000000000100
100110001000010100000000000101000
100111001010010000000000000100000
11000000011001001111111111111100
00001000010000010011000000000000
11111100000000000000000000000000

```

- (3) 初始化PC的值, 也就是以上程序段首地址PC=0x00000100, 假设以上程序段从0x00000100地址开始存放。

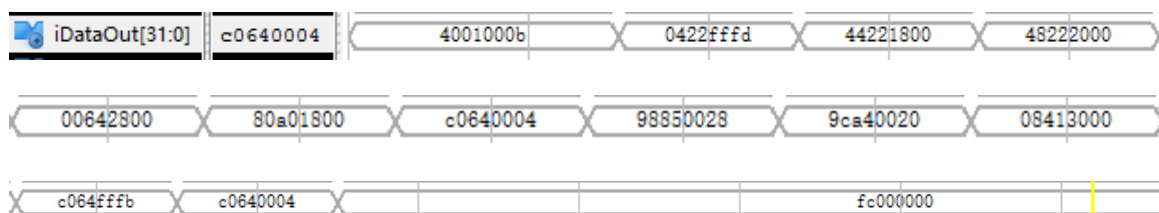
```

initial
    $readmemb("instructionData.txt", memory, 32'h100);

```

- (4) 运行Xilinx ISE进行仿真, 看波形。

指令代码为iDataOut, 将其转为十六进制, 以方便对比。本测试代码的仿真结果如下:



对比步骤1中的测试表, 可以看到, 每个时钟周期都正确获得了相应的指令, 一开始顺序执行, 到第一个beq指令不跳转, 顺序执行到第二个beq指令, 判等成立, 跳转到第一个beq, 再跳转到最后一条指令halt后停止。





寄存器内的结果如下：

	0	1	2	3
0x1	11	8	19	19
0x5	19	-3	0	0
0x9	0	0	0	0
0xD	0	0	0	0
0x11	0	0	0	0
0x15	0	0	0	0
0x19	0	0	0	0
0x1D	0	0	X	

\$1到\$6的值和分析一致。

存储器内的结果如下：

	0	1	2	3
0x0	X	X	X	X
0x4	X	X	X	X
0x8	X	X	X	X
0xC	X	X	X	X
0x10	X	X	X	X
0x14	X	X	X	X
0x18	X	X	X	X
0x1C	X	X	X	X
0x20	X	X	X	X
0x24	X	X	X	X
0x28	X	X	X	X
0x2C	X	X	X	X
0x30	X	X	X	19
0x34	X	X	X	X
0x38	X	X	X	X
0x3C	X	X	X	X
0x40	X	X	X	X
0x44	X	X	X	X
0x48	X	X	X	X
0x4C	X	X	X	X

和分析一致。

## 5. 实验结果及分析

通过以上自己编写的测试代码执行的结果，加上我同时还测试了老师所提供的测试指令，可以判断设计实现的CPU是正确的。实验结果均和期望一致。

## 六. 实验心得

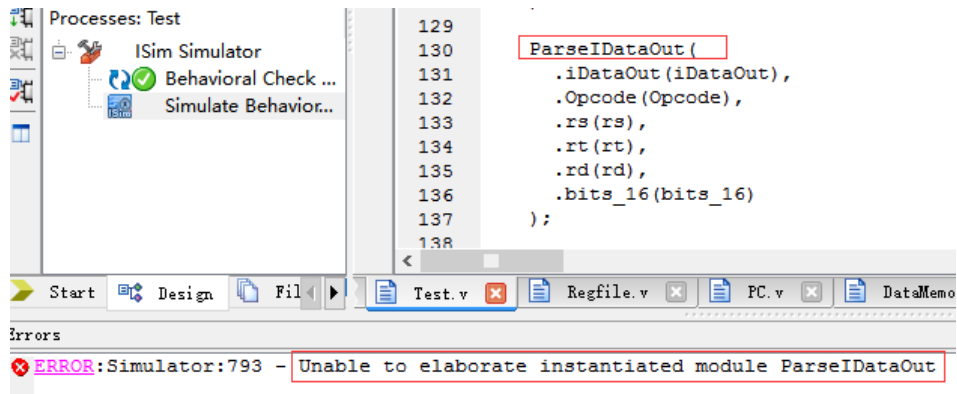
### 1. 实验过程中遇到的问题及解决的办法

主要是在编写完模块代码及连接的测试代码，仿真时，遇到了一下一些报错。

(1) 在Behavioral Check Syntax即语法检查阶段遇到的问题如下。

1) ERROR:Unable to elaborate instantiated module ParseIDataOut.

即“无法描述实例化模块ParseIDataOut”，截图如下。



解决办法：由于第一次遇到这个错误，需要理解这句报错的语义，在stackoverflow网站上查到了相似的问题<sup>2</sup>：

There has to be a name to the instantiated module. Please try this.

```

mux8bit A (in,w1,shift[0],w2);

```

实例化模块需要有对应名称。找到ParseIData模块，果然漏加名称了，改正如下（加上红框内的名称）：

```

ParseIDataOut parseIDataOut{
    .iDataOut (iDataOut),
    .Opcode (Opcode),
    .rs (rs),
    .rt (rt),
    .rd (rd),
    .bits_16 (bits_16)
};

```

2) Empty port in module declaration.

<sup>2</sup> <http://stackoverflow.com/questions/35495441/unable-to-elaborate-instantiated-module-in-verilog>

```

20 .....
21 module InsMemory(
22     RW, IAddr, IDataOut,
23 );
24     input [31:0] RW, IAddr;
25     output reg[31:0] IDataOut;
26     reg [31:0] memory[0:310];
27
28     always @(RW or IAddr) begin

```

Summary | Test.v | Regfile.v | PC.v | DataMemory.v

TOP/CPU\_design/MUX2T1\_32.v" Line 36: Target <out> of concurrent assign  
 TOP/CPU\_design/InsMemory.v" Line 23: Empty port in module declaration

解决方法：找到出错的代码，很容易发现在模块声明的参数中，最后一个参数后面多了一个逗号。

- 3) Formal port size is 32-bit while actual signal size is 1-bit.

```

133
134 ParseIDataOut parseIDataOut(
135     .iDataOut(iDataOut),
136     .Opcode(Opcode),
137     .rs(rs),
138     .rt(rt).

```

sign Summary | Test.v | Regfile.v | PC.v | DataMemory.v

ParseIDataOut>. Formal port size is 32-bit while actual signal size is 1-bit.

运行时，没有语法（error）错误，但有一些提示（warning），对此要认真看待，因为虽然一些提示的问题不会引起报错，但是正是一些代码编写的不规范、小问题会接连引起其他未报错的错误。

解决办法：

这个提示的语义是实例化模块时规定的iDataOut的端口是32位的，但实际接收的信号大小是1位的。找到与iDataOut的相关代码。

实例化模块的接口：

```

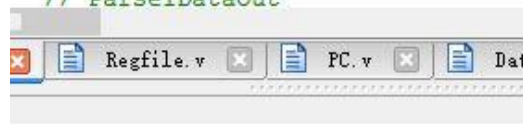
////////////////////////////////////
module ParseIDataOut(
    iDataOut, Opcode, rs, rt, rd, immediate
);
    input [31:0] iDataOut;
    output reg [5:0] Opcode;
    output reg [4:0] rs, rt, rd;
    output reg [15:0] immediate;
    always @(iDataOut) begin
        Opcode = iDataOut[31:26];
        rs = iDataOut[25:21];
        rt = iDataOut[20:16];
        rd = iDataOut[15:11];
        immediate = iDataOut[15:0];
    end
endmodule

// InsMemory
wire [31:0] pcOut;
wire [31:0] idataOut;

// Regfile
wire [4:0] writeReg;
wire [31:0] writeData;

// ParseIDataOut

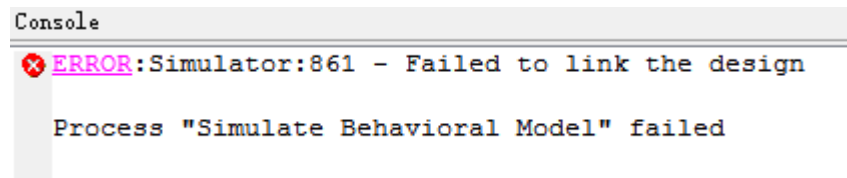
```



乍一看觉得没有错，都是32位的，但仔细一看，发现wire [31:0] 的名称为”idataOut”，大小的D错打成了小写的d，因此实际上iDataOut是没有对应的wire的。错误得到了解释。

## (2) 在Simulate Behavioral Model阶段遇到的问题

- 1) ERROR:Failed to link the design. Process “Simulate Behavioral Model” failed



```

Console
ERROR:Simulator:861 - Failed to link the design
Process "Simulate Behavioral Model" failed

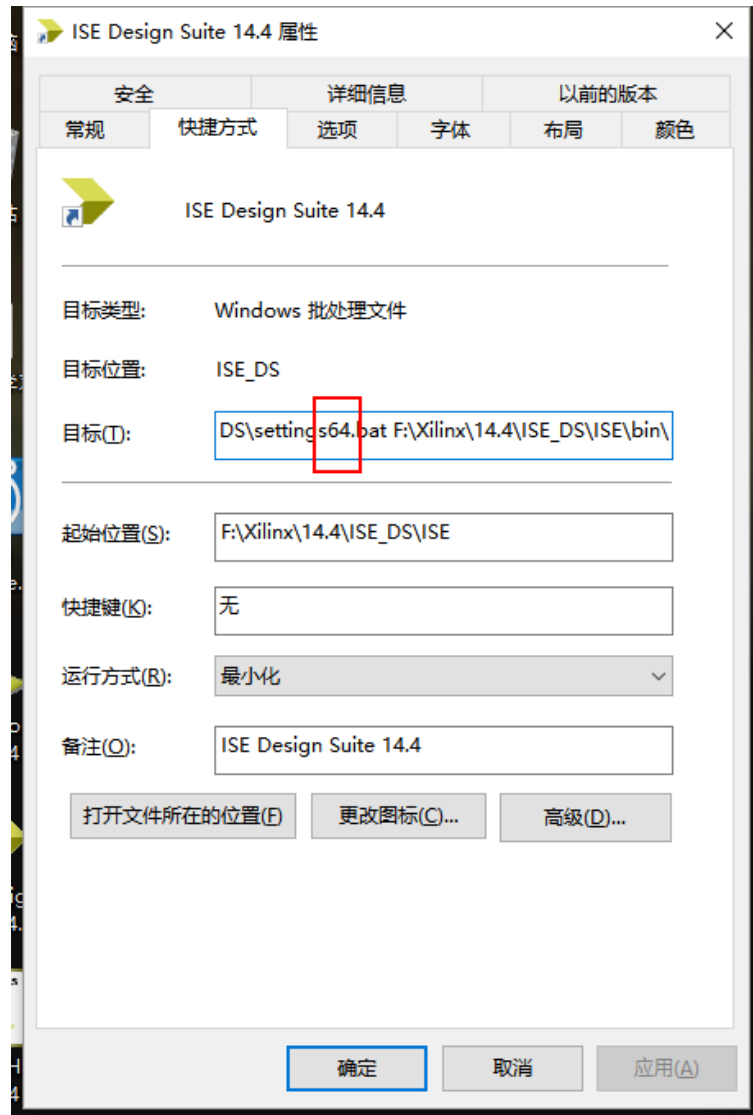
```

解决办法：出现这个报错时，我一直以为是代码错误，没有正确连接各模块导致的无法连接整个设计。但是我将各个模块的接口都检查了一遍，甚至采用了数一共有多少根线，检查每根线是否写了的方法也还是报错。经同学提示，这应该是软件本身的问题，ISE32位仿真时可能出现问题。

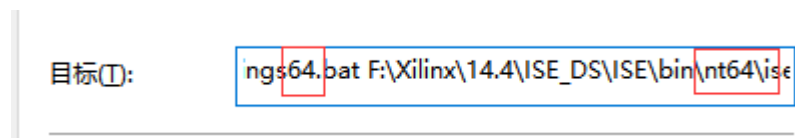
我下载的ISE是64位的，但在软件中，只要处于中文输入法模式，软件就会崩

崩溃退出，于是改成了32位解决了这个问题。改的方法很简单，打开属性菜单，将目标路径中的settings64改成settings32并将后面的nt64的64删除即可。如下图所示。

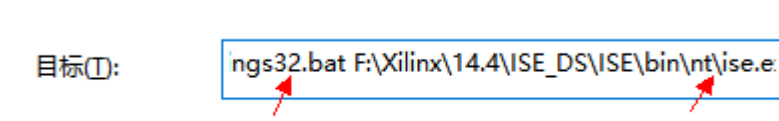
属性：



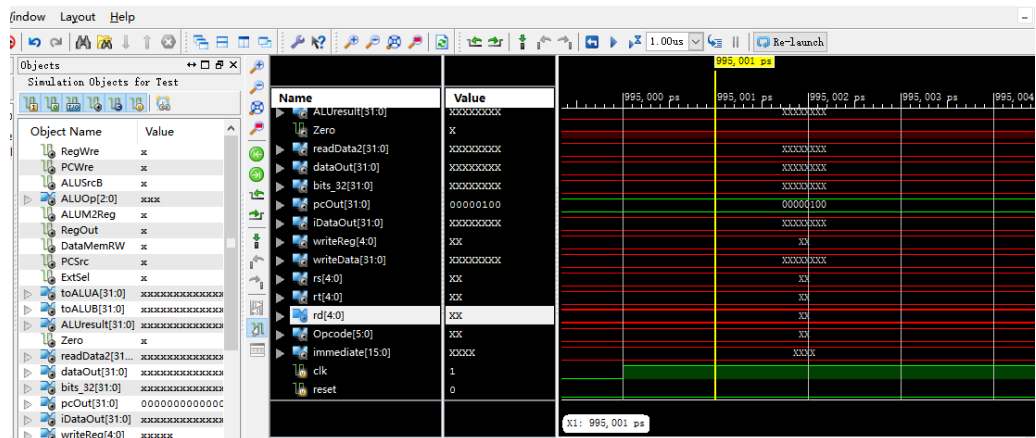
64位：



32位：



为了解决” Failed to link the design.” 的问题，我改回了64位，果然好了。出现了下图的模拟器界面：



注意到所有的地址及数据都是XXXX的形式，因为没有将指令代码初始化到指令存储器中。这一步我选择在InsMemory模块的初始化中用readmemb方法<sup>3</sup>来将指令代码初始化到指令寄存器。

```
$readmemb("file_name", memory_name); //初始化数据为十六进制
```

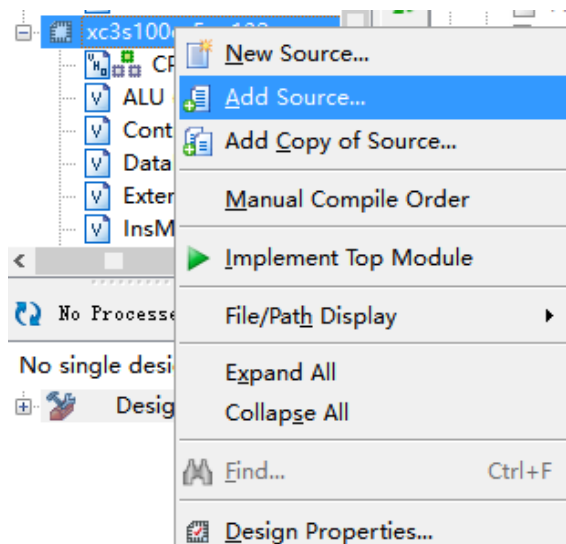
```
$readmemb("file_name", memory_name); //初始化数据为二进制
```

```
module InsMemory(
    IAddr, IDataOut
);
// 本设计中指令寄存器只读不写，RW=InsMemRW=0
input [31:0] IAddr;
output reg[31:0] IDataOut;
reg [31:0] memory[0:511];

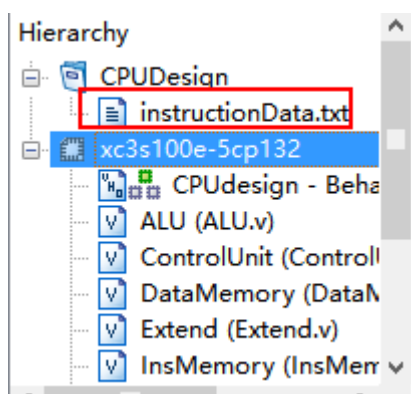
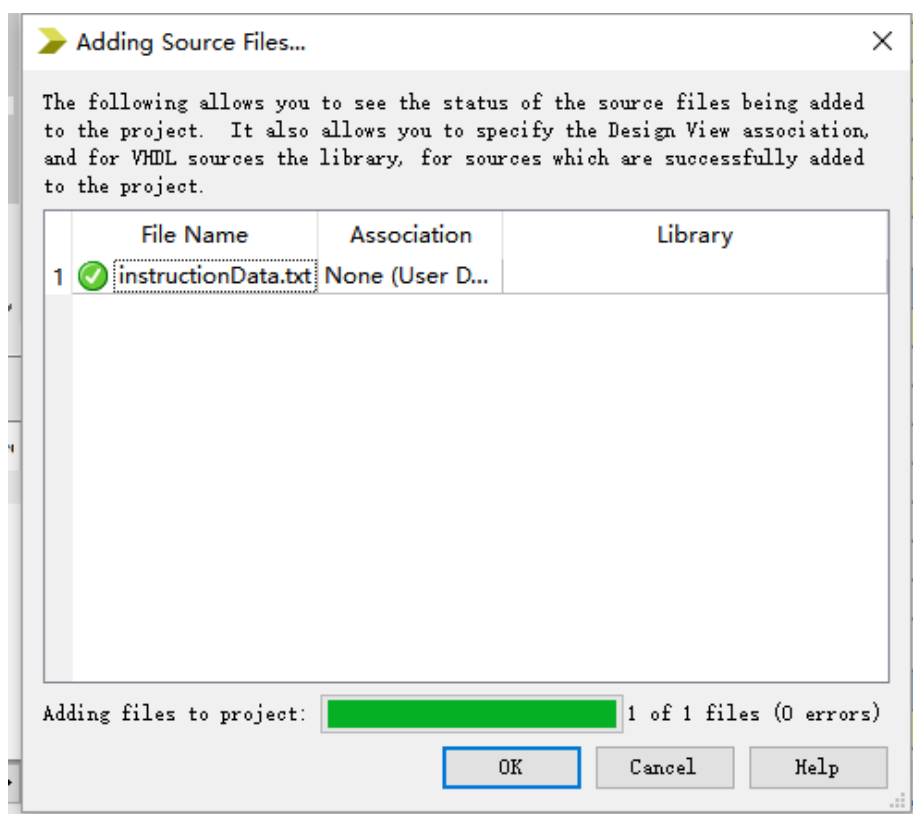
initial
    $readmemb("instructionData.txt", memory, 32'h100);
always @(IAddr) begin
    // 错误的！每个指令顺序存储在数组中，但指令地址是+=4，导致取到了ori的后四条指令and
    // IDataOut = memory[IAddr];
    IDataOut = memory[32'h100 + (IAddr - 32'h100) / 32'd4]; // read
end
endmodule
```

用文件” instructionData.txt” 文件设定存储器初值。下一步需要将这个文件添加到项目中。我先将其拷贝进了次项目的文件夹下但在Add Source时只要一点，软件又直接崩溃退出了。

<sup>3</sup> [http://blog.sina.com.cn/s/blog\\_7e2e98ad0101aq9a.html](http://blog.sina.com.cn/s/blog_7e2e98ad0101aq9a.html)



这次我很快意识到可能是ISE的位数问题。再次改回32位确实可以了。





我的体会：这次安装的ISE普遍会出现崩溃的问题，已发现的起因有64位下在中文输入法下输入、64位下为项目添加目标文件（Add Source），另外在32位下无法仿真。而上学期我们安装的ISE没有出现此类问题，因此推测是不同版本导致或者是软件和操作系统(Win10)的兼容问题。

### （3） 进入仿真后遇到的问题

#### 1) 指令存储器存指不正确问题

仿真时，一步步运行每一条指令，应当执行ori指令，pcOut输出的PC地址是正确的，但是经过指令存储器后取出的iDataOut却对应的是add指令：

name	Value
ALUresult[31:0]	00000000000000000000000000000000
Zero	1
readData2[31:0]	00000000000000000000000000000000
dataOut[31:0]	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
bits_32[31:0]	00000000000000000000100000000000
pcOut[31:0]	00000104 ori
iDataOut[31:0]	01000100001000100010000000000000
writeReg[4:0]	00100
writeData[31:0]	00000000000000000000000000000000
rs[4:0]	00001
rt[4:0]	00010
rd[4:0]	00100
Opcode[5:0]	010001 add
immediate[15:0]	0010000000000000
clk	1
reset	0

回到指令存储器模块的代码，红框标示了数据的存储方式：

```

module InsMemory(
    IAddr, IDataOut
);
// 本设计中指令寄存器只读不写，RW=InsMemRW=0
input [31:0] IAddr;
output reg[31:0] IDataOut;
reg [31:0] memory[0:310];

initial
    $readmemb("instructionData.txt", memory, 32'h100);
always @(IAddr) begin
    IDataOut = memory[IAddr]; // read
end

endmodule

```

直接用取值地址IAddr作为memory数组下标访问，数组中指令是顺序存储的，

而IAddr是PC的输出，至少是4的倍数，因为ori的前一条不是跳转指令，因此PC加了4，取到了ori的后四条指令and。因此应当改成IAddr每加4就对应数组下一位置，注意要从初始地址32'h100开始。最后改正如下：

```
always @(IAddr) begin
    // 错误的！每个指令顺序存储在数组中，但指令地址是+=4，导致取到了ori的后四条指令and
    // IDataOut = memory[IAddr];
    IDataOut = memory[32'h100 + (IAddr - 32'h100) / 32'd4]; // read
end
```

正确结果如下：

pcOut[31:0]	00000104
iDataOut[31:0]	010000000000000010000000000000001100
writeReg[4:0]	00010
writeData[31:0]	000000000000000000000000000000001100
rs[4:0]	00000
rt[4:0]	00010
rd[4:0]	00000
Opcode[5:0]	010000
immediate[15:0]	00000000000000001100
clk	1
reset	0

## 2) 指令解析问题

观察到了halt指令，Opcode理应为111111，从iDataOut解析Opcode的过程是正确的，txt文件中也有111111，但得到的iDataOut不对：

pcOut[31:0]	00000120
iDataOut[31:0]	11111000000000000000000000000000
writeReg[4:0]	00000
writeData[31:0]	00000000000000000000000000000000
rs[4:0]	00000
rt[4:0]	00000
rd[4:0]	00000
Opcode[5:0]	111110
immediate[15:0]	00000000000000000000
clk	1
reset	0

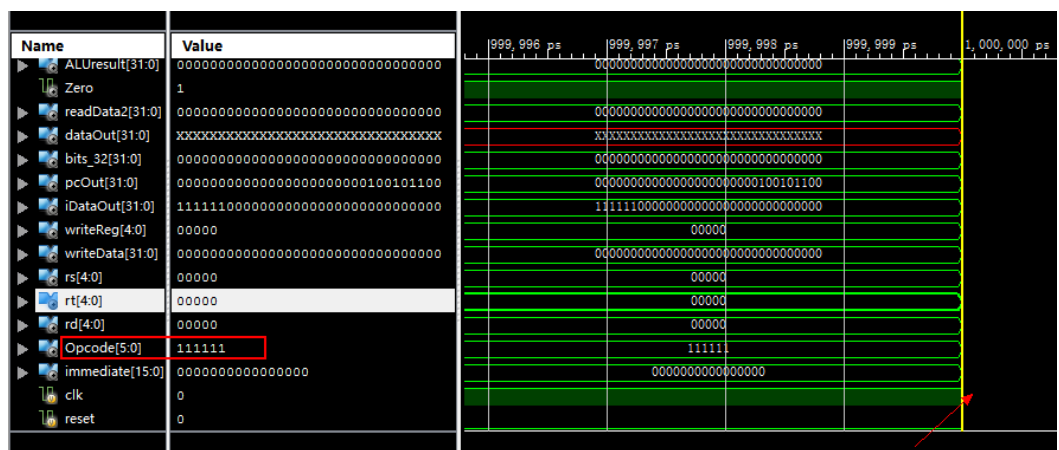
解决办法：找到初始指令存储器的文件instructionData.txt,发现是halt指令对应的32位二进制码错输成了33位，导致截断了最高位1。

```

1  000001000000000010000000000001010
2  010000000000000010000000000001100
3  000000000001000100001100000000000
4  000010000100000100101000000000000
5  010001000010001000100000000000000
6  010010000010001001000000000000000
7  110000000010001000000000000000100
8  100000010000000001011000000000000
9  100110000010001000000000100101100
10 100111000100000100000000100101010
11 1100000000100010111111111111011
12 111111000000000000000000000000000
13

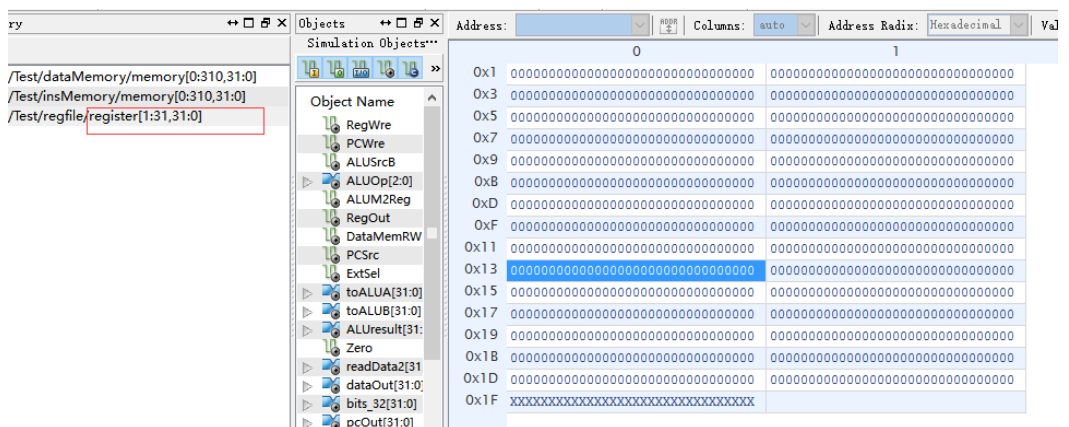
```

改正后得到了正确的结果：



3) 在应当跳转的指令beq处没有跳转。

解决办法：指令不跳转，怀疑没有判断\$1, \$2相等，查看寄存器的内容，全为0：



找到寄存器相应代码：

```

module Regfile(
    ReadReg1, ReadReg2, WriteData, WriteReg, WE, CLK, clrn, ReadData1, ReadData2
);
    input [4:0] ReadReg1, ReadReg2, WriteReg;
    input [31:0] WriteData;
    input WE, CLK, clrn;
    output [31:0] ReadData1, ReadData2;
    reg [31:0] register[1:31]; //r1-r31
    integer i; // 注意要在外面声明
    assign ReadData1 = (ReadReg1 == 0) ? 0 : register[ReadReg1];
    assign ReadData2 = (ReadReg2 == 0) ? 0 : register[ReadReg2];
    always @(posedge CLK or negedge clrn) begin
        if (clrn == 0) begin
            // reset
            for(i=1;i<31;i=i+1)
                register[i] <= 0;

        end
        else begin
            // write
            if((WriteReg!=0) && (WE == 1))
                register[WriteReg] <= WriteData;
        end
    end
endmodule

```

红框内语义错误，在程序的其他地方，clrn=1时表示清零，而此次反了，改为clrn=1时清零即可。

4) 控制信号有误

执行到beq时仍然不跳转，查看对应数据：

[illegible]

发现writeReg=0=rd,但beq的指令中beq rs,rt,**immediate**, writeReg应等于rt。

又知道控制单元的RegOut控制该二选一选择器, 应为0. 找到ControlUnit中对应

beq部分，果然错了：

```
// beq, R-format
6'b110000:
begin
    RegWre = 0;
    PCWre = 1;
    ALUSrcB = 0;
    ALUOp = 001;
    ALUM2Reg = 0;
    RegOut = 1;
    DataMemRW = 0;
    if(Zero==0) PCSrc = 0;
    else PCSrc = 1;
    ExtSel = 0;
end
```

##### 5) PC存储有误

错误：pc=0x118应当转pc=0x12c时，跳转到了pc=0x140。

解决办法：找到相关代码，

```
module PC(
    pcIn, CLK, Reset, PCSrc, PCWre, pcOut
);
input [31:0] pcIn;
input CLK;
input Reset;
input PCSrc;
input PCWre;
output reg [31:0] pcOut;
reg [31:0] pc;
initial
    pc = 32'h100;
always @(posedge CLK or posedge Reset) begin
    if (Reset == 1) begin
        // reset
        pcOut = 32'h100;
    end
    else if (PCWre != 0) begin
        pc = pc + 4;
        if(PCSrc == 0) pcOut = pc;
        else pcOut = pc + (pcIn << 2);
    end
end
endmodule
```

注意到PC模块的内部存储地址的pc每次都加了4，但在执行了跳转指令时却没有跟新，应当更新reg [31:0] pc到最新计算出的pc：

```

module PC(
    pcIn, CLK, Reset, PCSrc, PCWre, pcOut
);
input [31:0] pcIn;
input CLK;
input Reset;
input PCSrc;
input PCWre;
output reg [31:0] pcOut;
reg [31:0] pc;
always @(posedge CLK or posedge Reset) begin
    if (Reset == 1) begin
        // reset
        pcOut = 32'h100;
        pc = 32'h100;
    end
    else if (PCWre != 0) begin
        if (PCSrc == 0) pcOut = pc + 4;
        else pcOut = pc + 4 + (pcIn << 2);
        pc = pcOut;
    end
end
endmodule

```

## 2. 体会

本次实验具有很强的系统性，需要进行统筹考虑，从分析、设计到实现，每一步骤都必不可少。由于刚着手实验时，还没有学习CPU设计的理论知识，看着实验课件有些许茫然，也不知如何分析和设计。

我选择从小的方面切入，比如我先学习课件中给出的控制单元的代码是如何编写的，首先是Verilog的语法如何理解（看相关课件），再看给出的加和减操作对应控制信号的变化。我先照葫芦画瓢补充了剩下操作的代码。这建立在分析填写好表3. 控制信号与指令的关系表的基础上。在填写这个表的过程中，我需要去了解什么是控制单元，为什么有那么多控制信号，每个控制信号对应不同操作的变化是如何分析给出的，从而对CPU的基本架构有了一个感性认识。

下一步同样我选择填写可操作性强的表3 测试程序段，先填写老师给出的实例测试程序，填写指令时，需要去搞明白每个操作对应的指令如何构成、指令有哪些类型、一条条语句执行顺序是如何变化的、对应的地址是什么、寄存器和存储器中存了哪些数据发生了什么变化等等。

显然要分模块设计，先编写简单的模块如符号位扩展，了解了输入、输出、

端口及内部逻辑的编写规则，就可以融会贯通，对照图2. 单周期CPU数据通路和控制线路图进行其他子模块的编写了。各子模块写完后要写测试代码连接子模块并测试功能是否正常，需要查阅资料，了解语法规则及语义。

当然少不了出错debug的过程，通过看波形、检查代码、分析逻辑不难解决。

以上写代码的过程总体来说比较顺利，可以举一反三。然而真正让我理解整个单周期CPU设计过程的是写实验报告总结归纳的过程。

一开始也是无头绪的，不知从何着手写分析、设计的内容，于是我先写“遇到的问题”这个部分，再写一些编写代码的过程。这时免不了要重复看课件、资料来对自己的代码解释，一些之前没有仔细看的内容也要斟酌斟酌。由于第九周学习了CPU设计的理论知识，有了一个更清晰的认识。再看老师提供的各种资料，就能将它们联系起来了。

理论和实验是相辅相成的。做实验可以深入理论的细节、验证理论，学理论可以知其所以然、剖析实验。从任一方开始进行摸索，同时结合另一方是最好的学习方式。

也可以看到，单周期CPU的设计方式多样，比如子模块的划分方法、接口的定义、子模块的实现都不尽相同，对应的代码量、逻辑强度、性能也各有千秋，需要在多次实验中对比学习、多思考、多总结，找到更规范、更科学、更适合自己的方法。