



**计算机组成原理(汇编与接口) 课 程 设 计**

个人实验报告

|  |  |
| --- | --- |
| 学 院 | 计算机学院 |
| 专 业 | 计算机科学与技术 |
| 指导老师 | 刘璐 |
| 组 长 | 许哲源 |
| 组 员 | 许哲源 |
| 组长联系方式 | 15960212519 |

二O二一年 8 月

目录

[第一章 项目简述 1](#_Toc116204016)

[第二章 设计目的 1](#_Toc116204017)

[第三章 设计环境 1](#_Toc116204018)

[第四章 设计原理及内容 1](#_Toc116204019)

[4.1 CPU支持指令集 1](#_Toc116204020)

[4.2 指令功能说明 2](#_Toc116204021)

[4.3 CPU设计图 4](#_Toc116204022)

[4.4 总体设计说明 5](#_Toc116204023)

[4.6 PC模块 5](#_Toc116204024)

[4.7 regfile模块 6](#_Toc116204025)

[4.8 dmem与imem模块 7](#_Toc116204026)

[4.9 alu模块 8](#_Toc116204027)

[4.10 ctrl模块 8](#_Toc116204028)

[第五章 设计与实现 9](#_Toc116204029)

[5.1 Cpu\_top模块 10](#_Toc116204030)

[5.2 PC模块 10](#_Toc116204031)

[5.3 Regfile模块 11](#_Toc116204032)

[5.4 Dmem与Imem模块 12](#_Toc116204033)

[5.5 Alu模块 13](#_Toc116204034)

[5.6 Ctrl模块 14](#_Toc116204035)

[第六章 测试 15](#_Toc116204036)

[6.1 仿真测试用例 15](#_Toc116204037)

[6.2 测试结果及分析 15](#_Toc116204038)

[第七章 问题及解决方法 18](#_Toc116204039)

[第八章 心得体会及总结 18](#_Toc116204040)

[第九章 参考文献有价值的资源推荐 18](#_Toc116204041)

# 项目简述

利用Verilog语言，设计并实现了一个支持MIPS指令子集：Lui，Addiu，Add，Lw，Sw，Beq，J，以及个人抽取的sll指令和额外实现指令Ori的cpu。使用mips汇编编写了测试程序完成了对cpu所有功能的正确性测试。

# 设计目的

通过设计cpu，加深对于cpu整个执行过程的理解，熟悉verilog编程以及mips32指令集架构和汇编程序编写。

# 设计环境

|  |  |
| --- | --- |
| 操作系统 | Win11 家庭中文版 21H2 |
| 编程语言 | Verilog |
| EDA工具 | Vivado 2019.2 |
| 汇编语言 | Mips32 |
| 汇编程序编辑器 | Mars4\_5 |

# 设计原理及内容

4.1-4.2节给出了CPU支持的指令集以及指令的具体功能。在4.3节给出了CPU的架构图。在此基础之上，4.4节对CPU的总体设计原理和思路做出了说明。在4.5-4.10节中，给出各个模块的逻辑功能和接口，并对模块完成的功能进行说明。

## CPU支持指令集

本CPU支持8条指令，下表列举了实现的指令：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 指令 | opcode | function | 功能 |
| 1 | LUI | 001111 | / | rt = imm<<16 |
| 2 | ADDIU | 001001 | / | rd=rs+imm(无符号数) |
| 3 | ADD | 000000 | 100000 | rd=rs+rt |
| 4 | LW | 100011 | / | rt=mem[base+offset] |
| 5 | SW | 101011 | / | Mem[base+offset]=rt |
| 6 | BEQ | 000100 | / | rs=rt？跳转:不跳转 |
| 7 | J | 000010 | / | 无条件跳转 |
| 8 | SLL | 000000 | 000000 | rd=rs and rt |
| 9 | ORI | 001101 | / | rt=rs or imm |

表4-1 8条指令

## 指令功能说明

1. LUI

指令名：立即数赋值指令

汇编格式：LUI rt, immediate

汇编举例：LUI $2, 10

功能描述：：(rt)←immediate<<16 & 0FFFF0000H 即(rt)←immediate×65536;。首先 16 位立即数赋给 rt 寄存器的高 16 位，低 16 位用 0 填充。也就是将 16 位立即数左移16位赋值给 rt 寄存器。

1. ADDIU

指令名：无符号立即数加法指令

汇编格式：ADDIU rt, rs, immediate

汇编举例：ADDIU $4, $2, -100

功能描述：(rt)←(rs) + (Sign-Extend)immediate。首先将 16 位有符号立即数扩展到 32 位，然后加上 rs 中的数，结果给 rt 寄存器。加法操作溢出时不会抛出错误。

1. ADD

指令名：加法指令

汇编格式：add rd, rs, rt

汇编举例：add $4, $2, $3

功能描述：rd ← rs + rt。 32 位整数加法，源操作数分别在 rs, rt 两个通用寄存器中，结果放在 rd 寄存器。

1. LW

指令名：存储器读（字操作）

汇编格式：LW rt, offset(rs)

汇编举例：LW $3, 10($2) 或 LW $3, buff($2)

功能描述：(rt)←Memory[(rs)+(sign\_extend)offset]。以 rs 寄存器的内容为基地址，offset 通过符号扩展后形成 32 位的偏移，将基地址 加上偏移形成一个 32 位的地址，以此地址从 RAM 中读出一个字（4 字节）赋给 rt 寄存器。

1. SW

指令名：存储器写（字操作）

汇编格式：SW rt, offset(rs)

汇编举例：SW $3, 10($2)

功能描述：Memory[(rs)+(sign\_extend)offset]←(rt); PC ← NPC (PC + 4)。以 rs 寄存器的内容为基地址，offset 通过符号扩展后形成 32 位的偏移，将基地址 加上偏移形成一个 32 位的地址，将 rt 寄存器的内容写入到 RAM 中该地址开始的一个字（4 字节）单元。

1. BEQ

指令名：相等则转移指令

汇编格式：BEQ rt, rs, immediate

汇编举例：BEQ $3, $2, 10

功能描述：if ((rt)=(rs)) then (PC)←(PC)+4+( (Sign-Extend) offset<<2); else PC ← PC + 4。如果 rt 和 rs 的值相等，则转移到新的地址。新地址是当前指令的下一条指令地址 （PC+4）加上一个 32 位偏移量。该 32 位偏移量是将 16 位 offset 符号扩展到 32 位，然后左移 2 位（即乘 4）后取低 32 位得到。

1. J

指令名：无条件转移指令

汇编格式：J target

汇编举例：J 100

功能描述：(PC)←( (Zero-Extend) address<<2)。无条件转移到新的地址。新地址是 26 位 address 零扩展到 32 位，然后左移 2 位（即 乘 4）后取低 32 位得到。

1. SLL

指令名： 逻辑左移

汇编格式：sll rd,rt,sa

汇编举例：sll $4, $2, 1

功能描述：GPR[rd] ← GPR[rt] << sa。Rt寄存器的内容逻辑左移sa位，结果放在 rd 寄存器。

1. ORI

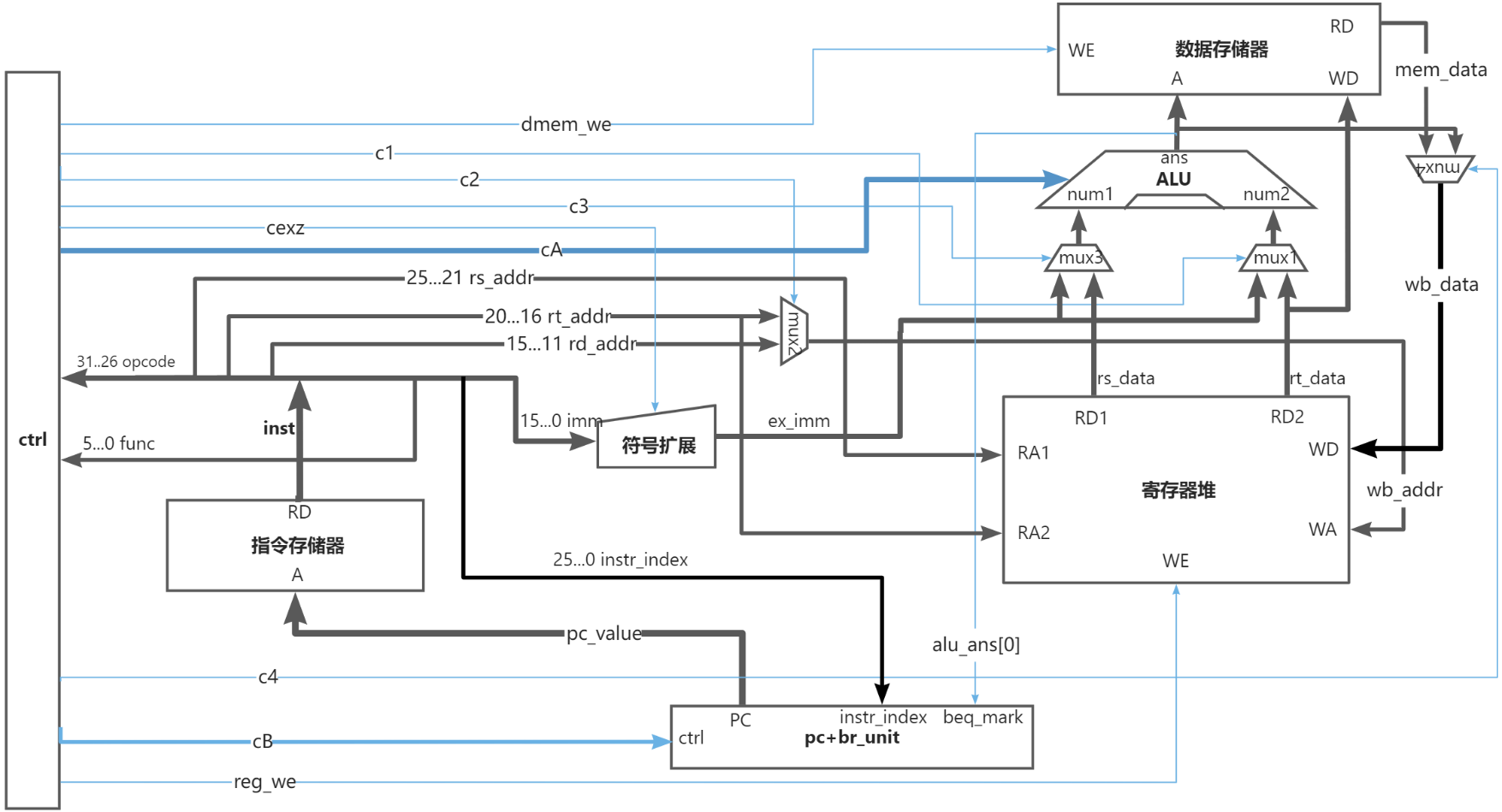
指令名： 或立即数

汇编格式：ori rt,rs,imm

汇编举例：ori $4, $2, 0x3cf1

功能描述：GPR[rt] ← GPR[rs] or immediate。 将16位立即数零扩展到32位后与rt寄存器中的内容做按位或运算，结果放在 rd 寄存器。

## CPU设计图



**方框**代表模块。框内文字所在处表示模块的接口

**黑色连线**代表数据传输通路。其中线的粗细能够反应数据的相对位宽的多少，线上的名字代表这条线路的名称。

**蓝色连线**代表控制信号通路。其中线的粗细能够反应数据的相对位宽的多少，线上的名字代表这条线路的名称。

## 总体设计说明

本次实验，我完成的CPU是基于哈佛结构的，即指令和数据分开存放。对于可能涉及到的地址，可以分成下面三类：PC中存放的是指令存储器的地址；访存指令lw，sw给出的是数据存储器中的地址；跳转指令涉及的地址则是指令存储器中的地址。

传统CPU的设计可以分为5个阶段：取指，译码，执行，访存，写回。一条指令在设计的单周期CPU中的执行过程也可以分为以上几个步骤。

取指阶段由PC给出指令地址，从指令存储器取出指令。译码阶段，将取出的指令送控制模块，控制模块传出控制信号，指令的不同字段送相应模块。执行阶段，各模块在控制信号的约束下，完成各自对应的功能。写回阶段，alu单元的模块输出作为寄存器堆的输入写回相应的寄存器。

在单周期CPU的设计中，所有指令在一个周期内完成执行，为了统一不同类型的指令的周期，下面考虑MIPS中的三种指令：

分支跳转指令(e.g. beq、j)：取指周期+译码/执行周期（+1 cycle delay on MIPS）

运算指令(e.g. add、and、srl)：取指周期+译码/执行周期

取数指令(e.g lw)：取指周期+译码/执行周期+取数周期

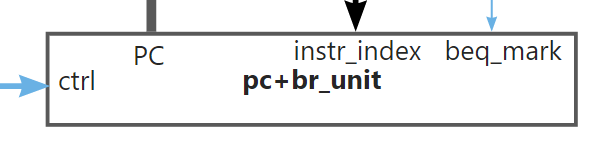
取最大的指令周期时间，将统一的单周期设定为取指周期+译码/执行周期（1 cycle）+取数周期。由于在实现中，用寄存器模拟内存，以及使用组合逻辑实现取指译码执行的阶段，因此一个时钟周期边缘触发了PC值的变化之后就可以完成取指周期+译码/执行周期+取数周期。在实现的指令集中，在各阶段唯一涉及到时序逻辑的指令是对数据存储器的写回指令，该条指令会在下一个时钟周期上边缘来临的时候将运算的结果写回内存。

明确了单周期的指令执行方式之后，下一步就是设计基本的功能部件，以及按照指令本身的需要在设计框架上加入新的组合逻辑。从一条简单的指令开始，逐步扩充cpu的逻辑设计。

## 4.6 PC模块

PC模块在项目中封装为pc.v，该模块实际上完成了PC + Next\_PC的功能。因此，该模块需要接受指令中的 [25:0] 字段内容作为跳转的依据，需要接受由ctrl模块译码得到的控制信号，根据ctrl字段来判断pc的输出值是跳转到远地址或者下一条指令。特别地，对于beq指令，原则上需要通过标志寄存器中相等标志的结果来判断是否需要跳转，这里为了简化设计，直接将相等比较的alu运算的第一位作为pc模块beq\_mark的输入，来判断输入的两个数是否相等，从而判断是否需要发生跳转。

在CPU启动时，PC模块内pc地址寄存器的值被初始化到0x00000000，配合从零地址写入的指令存储器的指令，驱动整个CPU工作。



module pc(

    input clk,

    input rst,

    input [1:0] ctrl, *// ctrl=00,no jump-off; ctrl=01,beq; ctrl=10,j*

    input [25:0] instr\_index,

    input beq\_mark, *// when ctrl=01 && beq\_mark == 0 , cast beq\_jmp; otherwise not*

    output [31:0] pc\_value

    );

## 4.7 regfile模块

寄存器堆模块由若干32位寄存器组成。在写使能为读有效的时候。通过指令译码得到的rs，rt地址作为输出，相应的rsrt寄存器的内容作为输出。在写使能为写有效的时候，寄存器堆模块会在下一个时钟周期上边缘到来时，将写回地址和写回数据端口中得到的寄存器地址和数据作为写入操作的对象。

具体来讲，读取操作采用组合逻辑实现，而对寄存器的写操作则采用时序逻辑实现。两种逻辑的区分通过使能信号的不同来区别。

module regfile(

    input clk,

    input rst,

    input reg\_we, *//write enable signal. :: =1,write back ; =0 read*

    input [4:0]rs\_addr,*//source register for reading*

    input [4:0]rt\_addr,*//target register for reading*

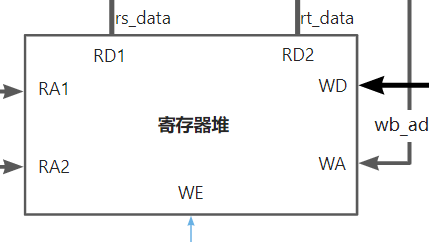
    input [4:0]wb\_addr,*//write-back register address*

    input [31:0]wb\_data,*//write-back register data*

    output [31:0] rs\_data,*//data from source register*

    output [31:0] rt\_data*//data from target register*

    );



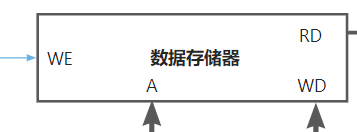
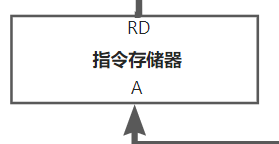
## 4.8 dmem与imem模块

存储器模块可以分为数据存储器模块以及指令存储器模块。两者的功能类似。本实验实现的CPU是哈弗结构，指令和数据分开存放。

为了简化设计，将指令存储器设定为只读，指令的加载在CPU启动之前就从0x00000地址开始全部写入。指令存储器的读出采用组合逻辑实现，在地址端口接收到pc给出的地址值之后，就会立刻读出当前应该执行的指令。

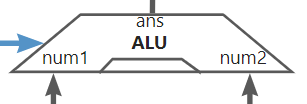
数据存储器设定为可读可写，同寄存器堆的思路一样，对于可读可写的部分，读入部分采用组合逻辑实现，写入部分采用时序逻辑实现，即在下一个时钟来临的正边缘，将数据写入到相应的数据存储器地址当中。

存储器均采用字节编址，能够接受的地址长度设定为32位，与采用的汇编指令MIPS32保持一致，能够索引4GB的地址空间。



## alu模块

alu模块是cpu的运算单元，本实验设计的是32位cpu，alu接受两个4字节的数据作为运算输入，输出一个32位的结果。Alu单元的输出由组合逻辑实现。输出的结果由控制信号指定，在几种算术运算和逻辑运算中进行选择。



## 4.10 ctrl模块

该模块接受指令的opcode（31：26）以及function字段（5：0）作为输入，输出相应指令对于各个模块和多路选择器的控制信号。指令的译码是根据mips32中给出的指令格式以及输入的两个字段来确定是哪一条指令。根据这条指令应完成的功能，来确定各个部件的控制信号，从而协同完成一条指令要求的功能。

module ctrl(

        input [5:0] opcode,

        input [5:0] func,

        output c1,

        output c2,

        output c3,

        output c4,

        output cexz,

        output [3:0] cA,

        output [1:0] cB,

        output dmem\_we,

        output reg\_we

    );

下面对控制信号c1，c2，c3，c4，cexz，cA，cb，dmem\_we, reg\_we 的含义作说明：

* + - 1. c1-rt和立即数之间2选1作为第二个操作数 【0对应rt】
      2. c2-rt和rd两个寄存器之间2选1作为目的寄存器
      3. c3-rs和立即数之间2选1作为第一个操作数
      4. c4-ALU运算结果和数据存储器读出数据2选1作为写回数据
      5. cexz-零扩展和符号扩展立即数2选1作为立即数输入
      6. dmem\_we-数据存储器写使能，0是读，1是写
      7. reg\_we-寄存器堆写使能，0是读，1是写
      8. cA-ALU6种运算使能(同一时间点最多只能某一个运算使能置位)
      9. cB-跳转控制信号

其中，当控制信号输出为0时选择前者，为1时选择后者。

下表给出了实现的9条指令对应的控制信号表。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | 汇编格式 | C1 | C2 | C3 | C4 | cexz | dmem\_we | reg\_we | cA(指令序号) | cB(跳转控制) |
| Lui | lui rt,imm | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 00 |
| Addiu | addiu rt,rs,imm | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 00 |
| Add | add rd,rs,rt | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 2 | 00 |
| Lw | lw rt,off(base) | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 3 | 00 |
| Sw | sw rt,off(base) | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 4 | 00 |
| Beq | beq rs,rt,off | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 01 |
| J | j target | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 10 |
| SLL | sll rd,rt,sa | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 7 | 00 |
| Ori | ori rt,rs,imm | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 8 | 00 |
| 保留指令 |  |  |  |  |  |  |  |  | 10 |  |

补充说明：

1. cA表示对alu运算的选择，由于指令数较少，对每条指令分别设置一个alu控制使能，使其完成对应的运算。实际上，这里还可以进一步抽象，将alu的运算简化为几种特定的运算，指令所使用的运算类型事实上是有重复的
2. cB是2位跳转控制信号，不同编码的含义如下：

ctrl=00, no jump-off //default;

ctrl=01, beq;

ctrl=10, j

# 设计与实现

## Cpu\_top模块

对应cpu\_top.v文件。

Cpu\_top模块依据整体设计图，将各个模块连线，同时，多路复用器mux都被放置在顶层模块中，由于本实验采用寄存器堆模拟存储器，因此地址与数据总线放在该模块内，没有对存储器进行与CPU的隔离。

由于不同模块需要的符号扩展的方法不一致，因此在顶层模块中，添加符号扩展的小模块：

*//sig\_extend module*

    assign imm = inst[15:0];

    assign exz\_imm = {16'b0,imm};

    assign exs\_imm = { {16{imm[15]}}, imm};

    assign ex\_imm = (cexz==0)?exz\_imm:exs\_imm;

其余的部件从各自的verilog模块中实例化，并按照CPU架构图进行线与模块之间的连接。

## PC模块

对应pc.v文件。在4.6节给出了该模块的设计原理和内容。本节将阐述具体的实现方法。

对于pc地址寄存，采用一个内部的pc寄存器 ：

reg [31:0] pc\_val 来保存程序当前的pc值，并将该寄存器连接到输出线上：

assign pc\_value = pc\_val;

对于pc地址初始化和在时钟周期正边缘来临跳变的功能，采用initial以及always块书写时序逻辑完成：

initial begin

        pc\_val = 32'b0;//default address to 0x00000000

    end

    always@(posedge clk)begin

        if(ctrl == 2'b01 && beq\_mark == 1'b0)begin

*//beq*

            pc\_val <= pc\_val + offset;

        end

        else if(ctrl == 2'b10)begin

*//j*

            pc\_val <= target;

;

        end

        else begin

*//default*

            pc\_val <= pc\_val + 4;

        end

    end

在每个时钟周期上边缘来临的时候，根据当前的控制信号来选择进行何种操作，自增或任意跳转或增加一个偏移量。其中，由于一条指令占4字节，且内存为字节编址，所以自增时pc指针增加4。

wire [31:0] target;*//target for 'j'*

    wire [31:0] offset; *//offset for 'beq'*

    wire [17:0] beq\_imm; *//intermediate variable for 'beq'*

    assign target = {pc\_value[31:28],{instr\_index,2'b0}};

    assign beq\_imm = {instr\_index[15:0],2'b0};

    assign offset = { {14{beq\_imm[17]}} , beq\_imm};

在模块中还定义了两个量offset和target，其中target供j指令使用，由指令的instr\_index字段右移两位添0，再和PC的[31:28]位拼接得到。而offset供beq指令使用，是由instr\_index字段右移两位添0，再对最高位符号扩展到32位得到。若beq的相等标志为真，则触发跳转。这里省略了状态标志寄存器，而直接用alu的判断相等输出来决定，若cpu需要扩充，可以加入标志寄存器。

## Regfile模块

对应pc.v文件。在4.7节给出了该模块的设计原理和内容。本节将阐述具体的实现方法。采用32个32位的寄存器来构成寄存器堆，地址由5位地址输入决定。

*//registers*

    reg [31:0] regs[31:0];

*//initialize regfiles whenever rst button is pushed.*

    integer i;

    always @(posedge clk or negedge rst)begin

        if(!rst)begin

            for(i=0;i<32;i=i+1) regs[i] <= 32'b0;

        end

        else if(reg\_we) regs[wb\_addr] <= wb\_data;

    end

*//reading*

    assign rs\_data = (rs\_addr==0)? 32'b0: regs[rs\_addr];

    assign rt\_data = (rt\_addr==0)? 32'b0: regs[rt\_addr];

整个寄存器模块的设计可以分为写和读两个部分。读出部分采用组合逻辑实现，若读取的是寄存器$0，则输出结果为全零，否则读出相应地址的寄存器内容。写入部分则是在每个时钟周期的上边缘来临的时候，若写使能有效，则将当前写回端口上的数据写入相应的寄存器中。

## Dmem与Imem模块

对应于dmem.v与imem.v这两个文件。由于指令寄存器和数据寄存器的编址以及寻址方式均相同，唯一的区别在于数据寄存器可读可写，需要dmem\_we额外使能信号控制；指令寄存器只读。因此，本节以较为复杂的dmem为例说明存储器的实现方法。下图是通过256个8位宽的寄存器来模拟一个大小为256B的dmem：

*//simulate mem*

    reg [7:0] dmem [255:0];

*//initialize dmem from local txt file*

    initial begin

        $readmemb("C:/projects/BIT-MIPS/bitmips\_experiments/test/singletest/dmem\_init.txt",dmem);

    end

*//read data from dmem*

    wire [7:0] addr = \_addr[7:0];

    assign r\_data = {dmem[addr+3],dmem[addr+2],dmem[addr+1],dmem[addr]};

*//write dmem if required(we=1)*

    always @(posedge clk) begin

        if(we) begin

*//little-endian as experiments required*

            dmem[addr] <= w\_data[7:0];

            dmem[addr+1] <= w\_data[15:8];

            dmem[addr+2] <= w\_data[23:16];

            dmem[addr+3] <= w\_data[31:24];

        end

    end

初始化时，通过vavado提供的$readmemh对存储器模块进行初始化。

读写部分和regfile模块一样，同样是分开实现。读出部分采用组合逻辑实现，写入部分则是在每个时钟周期的上边缘，若写使能有效，则按照小端序的方式写入存储器。

## Alu模块

对应于alu.v文件。Alu实现较为简单，alu内部通过组合逻辑，对输入的数一次性得到各种结果，再根据控制信号，选择相应的运算结果连接到输出端口即可。下图为alu支持的各种运算结果，包括符号扩展的相加结果，lui移位结果，beq所需要的相等标志，逻辑左移结果，以及ori或运算结果等。由于本处理器支持的指令很少，因此alu中的运算结果是基于指令而非基于通用的几大类算术或者逻辑运算而指定的。

wire [32:0] e\_alu\_num1={alu\_num1[31],alu\_num1}; *//signal\_extended\_num for overflow check*

    wire [32:0] e\_alu\_num2={alu\_num2[31],alu\_num2};

    wire [32:0] e\_add\_ans = e\_alu\_num1 + e\_alu\_num2;

    wire [31:0] lui\_ans = {alu\_num2[15:0],16'b0};

    wire [31:0] sw\_lw\_addr = e\_add\_ans[31:0];

    wire [31:0] beq\_ans = (alu\_num1==alu\_num2)?32'b0:32'b1;*//when equal, output 0,means that a jmp should be cast*

    wire [31:0] sll\_ans = alu\_num2 << alu\_num1;

    wire [31:0] ori\_ans = alu\_num1 | alu\_num2;

根据传入的控制信号，选择对应的结果连接到输出线上，实际上是多路选择器形成的组合逻辑：

assign ans =

        (ctrl==0)? lui\_ans:

        (ctrl==1||ctrl==2)? e\_add\_ans[31:0]:

        (ctrl==3||ctrl==4)? sw\_lw\_addr:

        (ctrl==5)?beq\_ans:

        (ctrl==7)?sll\_ans:

        (ctrl==8)?ori\_ans:

        32'b0;

## Ctrl模块

对应于ctrl.v文件。Ctrl模块完全为组合逻辑，实现的依据是项目文件夹下我整理的cpu.xlsx文件中不同指令对应的控制信号表。

首先通过输入的opcode和func字段来确定是哪一条指令，用inst\_type来存储当前指令的记号：

assign inst\_type =

        (opcode==6'b001111)? 0: *//lui*

        (opcode==6'b001001)? 1: *//addiu*

        (opcode==6'b100011)? 3: *//lw*

        (opcode==6'b101011)? 4: *//sw*

        (opcode==6'b000100)? 5: *//beq*

        (opcode==6'b000010)? 6: *//j*

        (opcode==6'b001101)? 8: *//ori*

        (opcode==6'b000000)? (

            (func==6'b100000)? 2: *//add*

            (func==6'b000000)? 7: *//sll*

            10):10;*//reserved*

然后根据控制信号表来设计不同指令输出控制信号的值。对每个控制信号，通过三目运算符快速直观的确定所有取值下分别对应哪些指令：

assign c1 = (inst\_type==0||inst\_type==1||inst\_type==3||inst\_type==4||inst\_type==8)? 1:0;

    assign c2 = (inst\_type==2||inst\_type==7)? 1:0;

    assign c3 = (inst\_type==7)?1:0;

    assign c4 = (inst\_type==3)?1:0;

    assign cexz = (inst\_type==10)?1:0;

    assign cA = inst\_type[3:0];

    assign cB =

        (inst\_type==5)? 2'b01:

        (inst\_type==6)? 2'b10: 2'b00;

    assign dmem\_we = (inst\_type==4)? 1:0;

    assign reg\_we = (inst\_type==4||inst\_type==5||inst\_type==6)?0:1; *//attention [0:1]*

这样就完成了控制模块的输入与输出。

# 测试

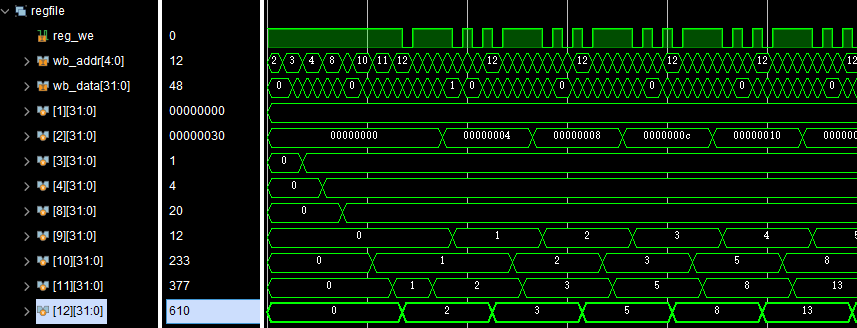
## 6.1 仿真测试用例

通过两个程序来验证CPU指令的正确性。

第一份是斐波那契数列前22项的计算和存储程序。第二份是在第一份的基础上，补充其余未被测试到的指令的一个简短的测试程序。通过两个程序，验证了单条指令的正确执行以及整个cpu连续的，指令间有前序依赖的正确执行。

## 6.2 测试结果及分析

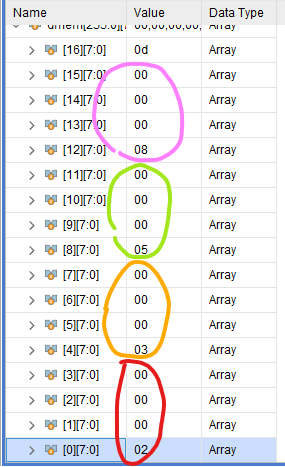
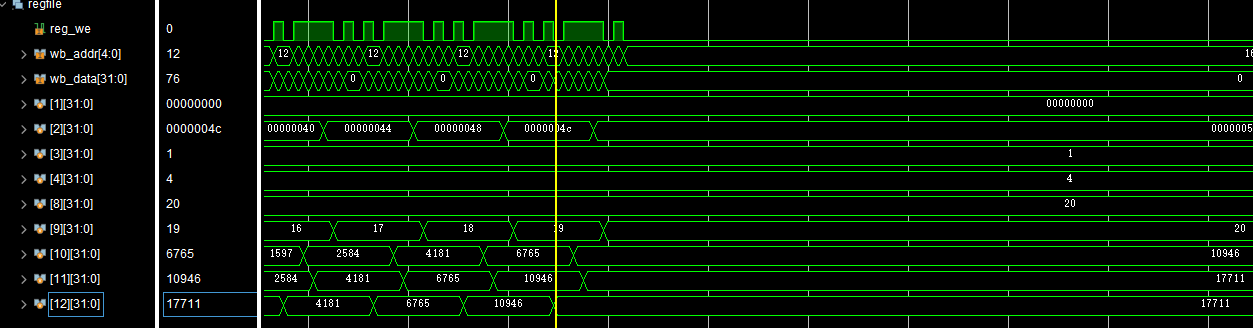
**斐波那契数列结果：**

本程序计算斐波那契数列的20项，并从2开始写入内存，每个数据占4个字节。寄存器$2保存的是当前程序应写入内存的地址；寄存器$3$4分别为常数1和4，供运行过程中一些指令调用；寄存器$8$9分别存储总共计算的项和已经计算的项，前者总是20，后者从1开始增加到和前者相同时停止循环；寄存器$10$11$12分别为Fn，Fn+1,Fn+2项，每循环一次，就有 $12 = $10 + $11，并将Fn+2写入（$2）内存地址，再令$10 = $11，$11 = $12，$2地址增加4，$9计数增加1。

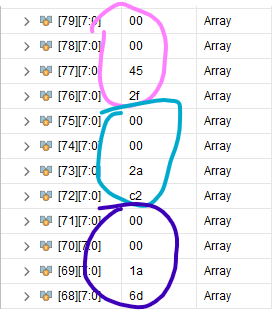
下面通过仿真结果说明CPU执行的正确性。

可以直观的从1,2,3,4,8,9,10,11,12寄存器的内容中看到计算过程的正确性，$12寄存器中保存的是Fn+2的结果，可以看到成斐波那契数列2，3，5，8，13，…

继续仿真一段时间，可以看到如下结果：

此时为程序结束的状态，寄存器$12的值为17711，是F22项，且此时$9为20达到中止的标准，说明程序在CPU上计算正确，下面看数据内存的写入情况：

从vivado的object视图中获取数据内存的写入情况，可以看到，按照程序中规定的一个数据占4个字节的方式，从0地址开始依次写数据存储器。写入的结果正是2，3，5，8，…这一计算出的斐波那契数列。

再看内存中存储的最后几项：这里还体现了数据存储器按小端序读写数据的正确性。地址76-79的内容为2f 45 00 00，按照小端序存储，实际的含义应为： 00 00 45 2f，换底为十进制是17711，由前面的仿真过程可知，F22为17711，这说明了存储器的正确性以及sw指令的正确完成。

斐波那契测试程序实际用到的指令有lui，ori，sw，add，beq，j，nop(sll)。这几条指令的在CPU上的运行能通过该程序的测试。

在实现的指令子集中，还有addiu和lw指令没有被测试验证，因此写了一个额外的小测试程序来验证其正确性。

**额外程序测试结果：**

lui     $2, 0x0000

ori $2, 0xabcd  *# $2 = 0x0000\_abcd, which is the data for sw/lw instruction test*

lui $3, 0x0000

ori $3, 0x0000  *# $3 = 0x0000\_0000, which is the address where data stores.*

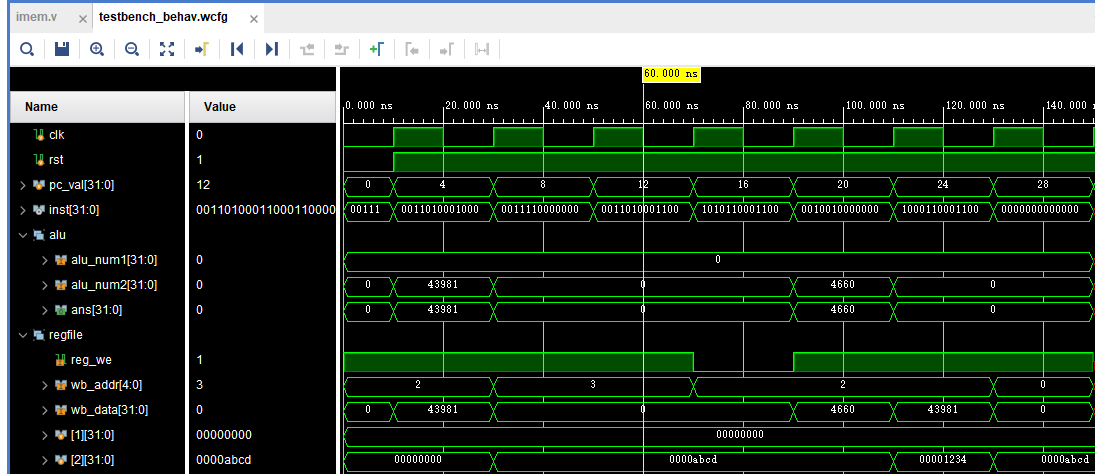
sw  $2, 0x0($3)

addiu   $2, $0, 0x1234  *# test 'addiu'*

lw  $2, 0x0($3) *# set $2 to the previous-stored word*

nop

使用上程序对addiu和lw指令进行测试。仿真结果如下图所示。

可见寄存器$2在addiu指令执行结束后，正确存储了计算结果0x00001234，接着在lw指令执行结束后，从数据存储器中成功加载了先前存储的字数据0x0000abcd。

两个测试程序对完成的所有指令进行了测试，同时对于CPU的实现的各种功能的正确性进行了验证，这由结果的正确性来保证。

# 问题及解决方法

在前文中已经对遇到的问题和相应的解决思路做了详细的解释，这里不再赘述。

# 心得体会及总结

通过本次实验完成了一个单指令周期cpu，对指令在cpu里的执行过程有了实践上的了解。在设计的过程中，对cpu结构的理解也加深了。设计上的细节仍有改进的空间，如alu单元抽象出几类通用的运算，增加cpu和存储器之间的独立性，完善alu运算异常的报错处理，设计图灵结构的cpu-内存系统，改进为不等长周期cpu，增加流水等等，可以继续挖掘的方向还有很多。

# 参考文献有价值的资源推荐

* + - 1. 乐学pdf课件
      2. 《自己动手写CPU》
      3. Mips32手册