# 计算机组成原理实验报告

20231164 张岳霖

## 一、CPU设计方案综述

### （一）总体设计概述

本CPU为Verilog实现的流水线MIPS - CPU，支持的指令集包含：{addu、subu、ori、lw、sw、beq、lui、jal、jr、nop}共10条。为了实现这些功能，CPU主要包含了PC、NPC、IM、Controller、Splitter、RF、EXT、ALU、DM。以上模块分别由一个独立的Verilog HDL文件组成，方便我们增加功能和修改代码，并在mips.v文件下实例化各模块，进行布线和连接。

### （二）关键模块定义

#### 1. PC

模块功能：程序计数器

模块定义：

module PC (

input wire clk,

input wire reset,

input wire [31:0] address\_in,

output wire [31:0] address\_out

);

表一 PC信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | Clk | I | 1 | 时钟信号，时钟上升沿更新当前指令地址为下一条指令地址 |
| 2 | Reset | I | 1 | 同步复位信号，信号为1时将指令初始化为0x0000\_3000 |
| 3 | Address\_in | I | 32 | 输入的地址信号 |
| 4 | Address\_out | O | 32 | 输出的地址信号 |

#### 2. NPC

模块功能：下一指令地址计算模块

模块定义：

module NPC (

input wire [31:0] pc,

input wire [31:0] bimm,

input wire aequb,

input wire agtb,

input wire [25:0] jimm,

input wire [31:0] ra,

input wire [4:0] npcop,

output wire [31:0] pc4,

output wire [31:0] npc

);

表二 NPC信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | PC | I | 32 | 当前指令地址 |
| 2 | Bimm | I | 32 | b类指令已处理的跳转地址 |
| 3 | Aeqb | I | 1 | 两操作数相等信号，用于Beq |
| 4 | Agtb | I | 1 | A操作数大于B操作数信号，用于Bgtz |
| 5 | Jimm | I | 26 | J和Jal指令未处理的跳转地址 |
| 6 | Ra | I | 32 | Jr和Jalr得到的来自寄存器的跳转地址 |
| 7 | NPCOp | I | 5 | NPC选择信号 |
| 8 | PC4 | O | 32 | 当前指令地址加4的数值，指令为jr时将其传入寄存器写入数值端 |
| 9 | NPC | O | 32 | 下一条指令的地址 |

NPCOp信号对应表

|  |  |
| --- | --- |
| NPCOp信号取值 | 对应的指令 |
| 0 | PC+4 |
| 1 | Beq |
| 2 | J Jal |
| 3 | Jr Jalr |
| 4 | Bgtz |
| 5 | 暂未定义 |

NPC内部实现逻辑为计算出所有可能的下一个指令地址值，再通过多路选择器选择出适当的地址进行输出，选择信号为NPCOp。

注：

1. 处理B类指令时，输入的信号(Bimm)为已经对立即数做零扩展，并左移两位的值；
2. 处理J和Jal指令时，输入的信号(Jimm)为直接取自当前指令0至25位的立即数，在NPC内部按照指令集的操作方法对其进行拼接；
3. 处理Jr和Jalr指令时，跳转地址为Ra的值。

#### 3. IM

模块功能：指令存储器

模块定义：

module IM (

input wire[31:0] address,

output wire[31:0] command

);

表三 IM信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | Address | I | 32 | 输入的地址 |
| 2 | Command | O | 32 | 输出地址对应的指令 |

在IM模块中，采用系统任务$readmemh("code.txt",command\_momery);读取code.txt文件中的指令机器码，根据地址的第2~11位读取对应的指令。

#### 4. Controller

模块功能：单周期控制器

模块定义：

module controller (

input wire[31:0] command,

output reg [4:0] npcop,

output reg [1:0] wrsel,

output reg [1:0] wdsel,

output reg rfwe,

output reg [1:0] extop,

output reg [1:0] asel,

output reg [1:0] bsel,

output reg [4:0] aluop,

output reg dmwe,

output reg [1:0] dmtype

);

表四 Controller信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | Command | I | 32 | 指令的机器码 |
| 2 | NPCOp | O | 4 | NPC选择信号，用于选择下一条指令，区分是否跳转(具体见NPC模块) |
| 3 | WRSel | O | 2 | 寄存器写入地址选择[20:16](0),[15:11](1),31号寄存器(2) |
| 4 | WDSel | O | 2 | 寄存器写入数据选择，ALU计算得到的数据(0),DM中取出的数据(1),NPC计算的下一个地址(2) |
| 5 | RFWE | O | 1 | GRF写入使能信号，1写入有效 |
| 6 | EXTOp | O | 2 | 控制对立即数的扩展方式：零扩展(0),符号扩展(1),部分位零扩展(2) |
| 7 | ASel | O | 2 | 控制ALU中B操作数的输入，GRF输出(0), |
| 8 | BSel | O | 2 | 控制ALU中B操作数的输入，GRF输出(0),扩展后的立即数(1) |
| 9 | ALUop | O | 5 | ALU操作选择信号 |
| 10 | DMWE | O | 1 | 内存写入使能信号，1写入有效 |
| 12 | DMtype | O | 2 | 内存操作类型：0(W),1(H),2(B) |

控制器controller的设计过程本质是一个译码的过程，其功能是将每一条机器指令包含的信息转化为恰当的CPU控制信号，并将各个控制信号输出传递到CPU的各个功能单元，通过对功能单元的控制达到控制CPU运行进程的效果。在Controller实现过程中，我采用了宏定义的方法定义了ALU的控制信号和指令对应的机器码信号，并在always@( \* )块中进行指令的判断和控制信号的赋值。

#### 5. RF

模块功能：寄存器堆

模块定义：

module RF (

input wire clk,

input wire reset,

input wire [4:0] a1,

input wire [4:0] a2,

input wire [4:0] a3,

input wire rfwe,

input wire [31:0] pc,

input wire [31:0] wd,

output wire [31:0] rd1,

output wire [31:0] rd2

);

表六 RF信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | Clk | I | 1 | 时钟信号 |
| 2 | Reset | I | 1 | 同步复位信号，信号为1将32个寄存器全部清零 |
| 3 | A1 | I | 5 | 地址输入信号，指定32个寄存器中的某一个，将其中存储的数据读入到RD1 |
| 4 | A2 | I | 5 | 地址输入信号，指定32个寄存器中的某一个，将其中存储的数据读入到RD2 |
| 5 | A3 | I | 5 | 地址输入信号，指定32个寄存器中的某一个作为写入的目标寄存器 |
| 6 | RFWE | I | 1 | GRF写入使能信号，1写入有效 |
| 7 | WD | I | 32 | 32位数据输入信号 |
| 8 | PC | I | 32 | 当前指令地址，评测需要 |
| 9 | RD1 | O | 32 | 输出A1指定的寄存器中的32位数据 |
| 10 | RD2 | O | 32 | 输出A2指定的寄存器中的32位数据 |

#### 6. EXT

模块功能：位扩展器

模块定义：

module EXT (

input wire [15:0] immoffset,

input wire [1:0] extop,

output wire [31:0] out

);

表七 位扩展器信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | ImmOffset | I | 16 | 待扩展数据 |
| 2 | EXTOp | I | 2 | 扩展方式选择信号：零扩展(0),符号扩展(1),部分位零扩展(2) |
| 3 | Out | O | 32 | 扩展后的数据 |

#### 7. ALU

模块功能：算术运算单元

模块定义：

module ALU (

input wire [31:0] a,

input wire [31:0] b,

input wire [4:0] aluop,

output wire aequb,

output wire agtb,

output wire [31:0] res

);

表八 ALU信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | A | I | 32 | 操作数1 |
| 2 | B | I | 32 | 操作数2 |
| 3 | ALUop | I | 5 | ALU操作类型选择信号 |
| 4 | Aequb | O | 1 | 操作数一和操作数二相等时为一 |
| 5 | Agtb | O | 1 | 操作数一大于操作数二是为一 |
| 6 | Res | O | 32 | 运算结果 |

ALU运算定义表

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 输入信号 | 宏定义 | 执行的操作 |
| 1. | 0 | `None or `AaddB | 两个操作数做加法（对于不涉及ALU操作的指令，统一将ALUOp设为0，对应宏`None） |
| 2. | 1 | `AsubB | 两个操作数做减法 |
| 3. | 2 | `AorB | 两个操作数按位或 |
| 4. | 3 | `luiB | 第二个操作数的低16位作为输出的高16位，后面补零 |
| 5. | 4 | `A | 直接输出A操作数 |
| 6. | 5 | `BsllA | B逻辑左移A位 |
| 7. | 6 | `AltB | A小于B |
| 8. | 7 |  | 以下接口预留未定义 |
| 9. | 8 |  |  |
| 10. | 9 |  |  |
| 11. | 10 |  |  |
| 12. | 11 |  |  |
| 13. | 12 |  |  |
| 14. | 13 |  |  |
| 15. | 14 |  |  |
| 16. | 15 |  |  |
| 17. | 16 |  |  |

#### 8. DM

模块功能：数据存储器

模块定义：

module DM (

input wire clk,

input wire reset,

input wire dmwe,

input wire [31:0] address,

input wire [31:0] wd,

input wire [1:0] dmtype,

input wire [31:0] pc,

output wire [31:0] nrd

);

表九 数据存储器信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | Clk | I | 1 | 时钟信号 |
| 2 | Reset | I | 1 | 同步复位信号 |
| 3 | DMWE | I | 1 | 内存写入使能信号，1写入有效 |
| 4 | Address | I | 32 | 32位地址输入信号 |
| 5 | WD | I | 32 | 32位数据输入信号 |
| 6 | DMtype | I | 2 | 内存操作类型：0(W),1(H),2(B) |
| 7 | PC | I | 32 | 当前指令地址，评测需要 |
| 7 | NRD | O | 32 | 输出Address指定地址对应的32位数据 |

### （三）控制模块设计

控制器的设计本质是译码的过程，下面类比搭建Logisim CPU的方法，列出了和逻辑和或逻辑真值表，并根据真值表进行指令信号的解码和控制信号的选取和输出。

#### 和逻辑真值表

R型指令（addu subu）：前六位均为0，可以通过后五位编码

I J型指令（ori lw sw beq lui）：可以直接通过前六位编码

Nop指令：全零

在Verilog实现的过程中，我们可以先根据command[31:26]是否为零，将指令分为R型和IJ型两类，再根据具体的OPCode和Funct进行编码。

PART1 OPCode == 0

`define nop 6'b000000 // op == 0

`define addu 6'b100001 // op == 0

`define subu 6'b100011 // op == 0

`define jr 6'b001000 // op == 0

`define sll 6'b000000 // op == 0

`define slt 6'b101010 // op == 0

PART2 OPCode != 0

`define ori 6'b001101 // op != 0

`define lw 6'b100011 // op != 0

`define sw 6'b101011 // op != 0

`define beq 6'b000100 // op != 0

`define lui 6'b001111 // op != 0

`define jal 6'b000011 // op != 0

`define j 6'b000010 // op != 0

`define sh 6'b101001 // op != 0

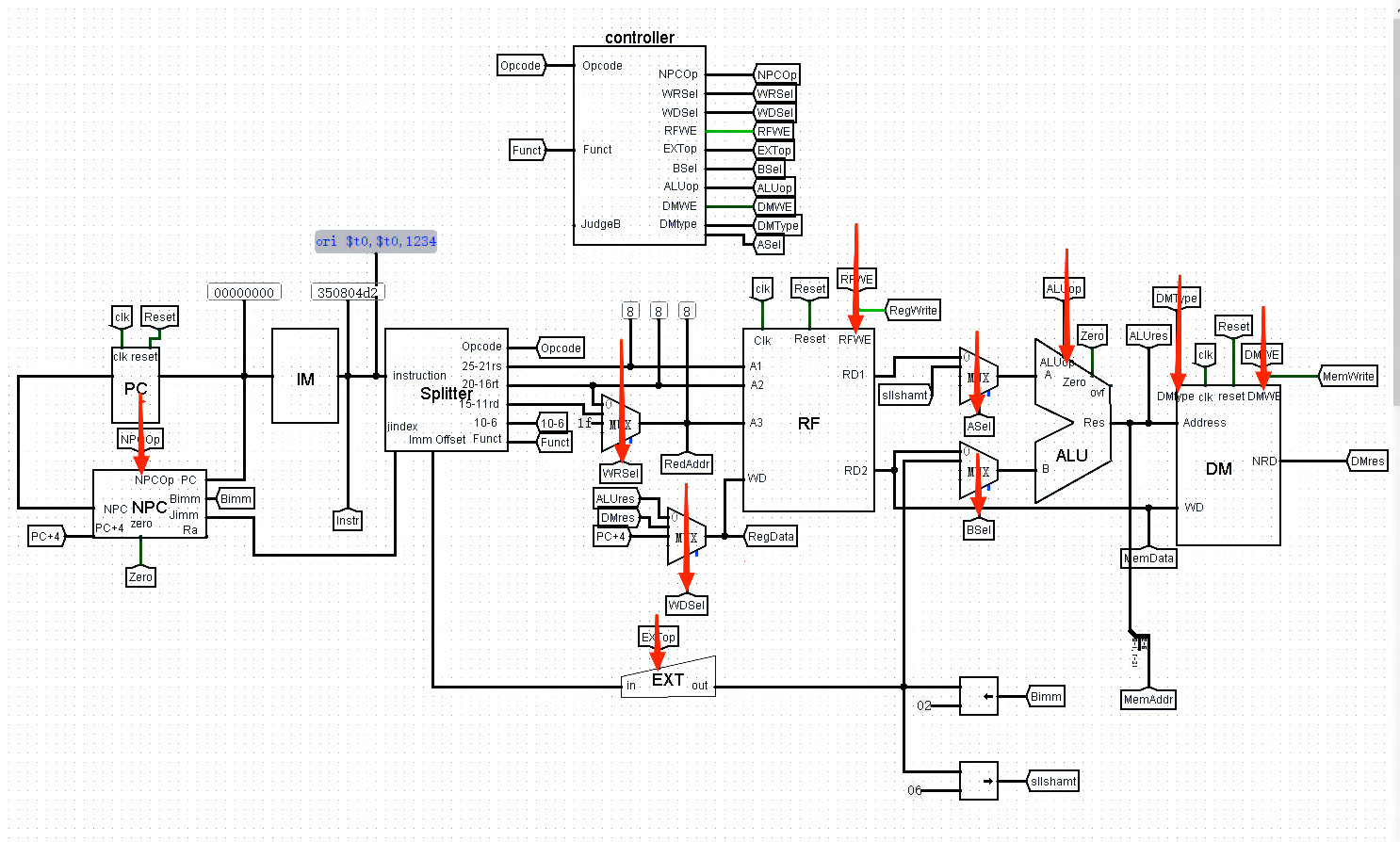
`define lb 6'b100000 // op != 0

`define bgtz 6'b000111 // op != 0

#### 与逻辑真值表

表十 与逻辑真值表

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | NPCOp | WRSel | WDSel | RFWE | EXTOp | ASel | BSel | ALUop | DMWE | DMtype |
| Addu | 0 | 1 | 0 | 1 | x | 0 | 0 | 0 | 0 | x |
| Subu | 0 | 1 | 0 | 1 | x | 0 | 0 | 1 | 0 | x |
| Ori | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 2 | 0 | x |
| Lw | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| Sw | 0 | x | x | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| Beq | 1 | x | x | 0 | 1 | 0 | x | x | 0 | x |
| Lui | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 3 | 0 | x |
| Jal | 2 | 2 | 2 | 1 | x | 0 | x | x | x | x |
| Jr | 3 | x | x | 0 | x | 0 | x | 4 | 0 | x |
| J | 2 | x | x | 0 | x | x | x | x | 0 | x |
| Nop | 0 | x | x | 0 | x | 0 | x | x | 0 | x |
| Sll | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 5 | 0 | x |
| Slt | 0 | 1 | 0 | 1 | x | 0 | 0 | 6 | 0 | x |
| Sh | 0 | x | x | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| Lb | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 2 |
| Bgtz | 4 | x | x | 0 | 1 | 0 | 0 | 7 | 0 | x |
| Jalr | 3 | 1 | 2 | 1 | 0 | 0 | 0 | 4 | 0 | 0 |



图一 控制信号图

### （四）重要机制实现方法

#### 1. 跳转

采用了控制器分析输入指令，产生跳转信号；NPC执行跳转信号，计算下一条指令所在位置的实现方法。针对不同的跳转指令，执行方法略有差异。对于B类指令，本次实现将EXT扩展为32位的立即数左移两位处理完成后再输入到NPC的实现方法，NPC中无需再次进行输入的立即数信号的处理；对于J指令和Jal指令，则将imm立即数直接输入到NPC中，在NPC中对立即数做进一步处理；对于Jr和Jalr指令，则将31号寄存器的数值GPR[rs]直接输入到NPC中。

## 二、测试方案

### （一）典型测试样例

在CPU的测试过程中，根据指令的类型及顺序以及操作数的正负大小不同等，我人为构造了一组测试样例（附录一）。同时，我根据讨论区同学的思路，编写了基于Python的自动代码生成器，每次产生mips指令约500行，对编写的CPU进行强测。通过使用对拍程序对比修改版MARS输出结果和ISim控制台输出结果来判断编写的CPU的正确性。

### （二）自动测试工具

#### 1. 测试样例生成器

import random

code = open("mipscode.txt", "w")

data1 = [] # 记录顺序执行的代码

data2 = [] # 记录跳转代码

interval = [] # 记录中间插入部分代码的行数

helper = [] # 记录sw和lw之前ori指令的相关内容

acnt = 0 # index of helper

class Instruction:

def \_\_init\_\_(self, type, lines=0, rs=None, rt=None, rd=None, oriindex=None):

self.type = type

self.lines = lines

self.rs = rs

self.rt = rt

self.rd = rd

self.oriindex = oriindex

def pre\_fill():

for i in range(2, 28):

code.write("lui ${},{}\n".format(i, random.randint(0, 65535)))

code.write("ori ${},${},{}\n".format(i, i, random.randint(0, 65535)))

for i in range(101):

code.write("sw ${},{}(${})\n".format(random.randint(0, 27), i \* 4, 0))

def print\_instr(x):

if data1[x].type == 1:

code.write("addu ${},${},${}\n".format(data1[x].rs, data1[x].rt, data1[x].rd))

elif data1[x].type == 2:

code.write("subu ${},${},${}\n".format(data1[x].rs, data1[x].rt, data1[x].rd))

elif data1[x].type == 3:

code.write("ori ${},${},{}\n".format(data1[x].rs, data1[x].rt, data1[x].rd))

elif data1[x].type == 4:

code.write("lui ${},{}\n".format(data1[x].rs, data1[x].rt))

elif data1[x].type == 5:

code.write("nop\n")

elif data1[x].type == 6:

y = data1[x].oriindex

code.write("ori ${},$0,{}\n".format(helper[y].rs, helper[y].rt))

code.write("lw ${},{}(${})\n".format(data1[x].rs, data1[x].rd, data1[x].rt))

elif data1[x].type == 7:

y = data1[x].oriindex

code.write("ori ${},$0,{}\n".format(helper[y].rs, helper[y].rt))

code.write("sw ${},{}(${})\n".format(data1[x].rs, data1[x].rd, data1[x].rt))

if \_\_name\_\_ == "\_\_main\_\_":

pre\_fill()

total\_blocks = random.randint(1, 15) # 代码块数

total\_lines = 0 # 代码行数

# 构建过程

for i in range(total\_blocks): # 代码块头尾记录在data2中

data2.append(Instruction(random.randint(0, 1), random.randint(1, 20)))

total\_lines += data2[i].lines

for i in range(total\_blocks + 1): # 两个interval之间夹一个代码块

interval.append(random.randint(1, 20))

total\_lines += interval[i]

for i in range(0, total\_lines): # 随机生成顺序执行代码，保存在data1中

data1.append(Instruction(random.randint(1, 7))) # 5->nop

if data1[i].type == 1 or data1[i].type == 2: # addu subu

data1[i].rs = random.randint(2, 27)

data1[i].rt = random.randint(2, 27)

data1[i].rd = random.randint(2, 27)

elif data1[i].type == 3: # ori

data1[i].rs = random.randint(2, 27)

data1[i].rt = random.randint(2, 27)

data1[i].rd = random.randint(0, 65535)

elif data1[i].type == 4: # lui

data1[i].rs = random.randint(2, 27)

data1[i].rt = random.randint(0, 65535)

elif data1[i].type == 6 or data1[i].type == 7: # 6->lw 7->sw

data1[i].oriindex = acnt

acnt += 1

helper.append(Instruction(3)) # 构建ori指令，保证位置是4的倍数

helper[acnt - 1].rs = random.randint(2, 27)

helper[acnt - 1].rt = 4 \* random.randint(0, 100)

data1[i].rs = random.randint(2, 27)

data1[i].rt = helper[acnt - 1].rs

if helper[acnt - 1].rt >= 50:

data1[i].rd = -4 \* random.randint(0, 12) # offset是4的倍数

else:

data1[i].rd = 4 \* random.randint(0, 12)

for i in range(total\_blocks):

if data2[i].type == 0: # data2[i]==0-->beq data2[i]==1-->jal

coin = random.randint(0, 2)

if coin != 0:

data2[i].rs = data2[i].rt = random.randint(0, 27) # coin !=0 保证rs=rt跳转

else:

data2[i].rs = random.randint(0, 27)

data2[i].rt = random.randint(0, 27)

code.write("ori $31,$0,0x00003F00\n") # set $ra

code.write("ori $29,$0,0x00002F00\n") # set $sp

# 输出过程

k = 0

for i in range(total\_blocks):

for j in range(interval[i]): # 输出interval

print\_instr(k)

k += 1

if data2[i].type == 0: # 输出data2[i]

code.write("beq ${},${},branch{}\n".format(data2[i].rs, data2[i].rt, i))

for j in range(data2[i].lines):

print\_instr(k)

k += 1

code.write("branch{}:\n".format(i))

else:

code.write("ori $2,$0,4\n")

code.write("subu $sp,$sp,$2\n")

code.write("sw $ra,4($sp)\n")

code.write("jal func{}\n".format(i)) # 调用jal时候先压栈，返回时再弹栈

code.write("ori $2,$0,4\n")

code.write("lw $ra,4($sp)\n")

code.write("addu $sp,$sp,$2\n")

code.write("jr $ra\n")

for i in range(total\_blocks):

if data2[i].type == 1:

code.write("func{}:\n".format(i))

for j in range(data2[i].lines):

k += 1

print\_instr(k)

code.write("jr $ra\n")

#### 2. 自动执行脚本

import os

import re

# 生成随机代码+命令行导入Mars并导出机器码

os.system(r"python create.py >mips8.asm")

os.system("java -jar Mars4\_5.jar mips8.asm nc mc CompactTextAtZero a dump .text HexText code.txt")

# 仿真需要的两个文件

tclFile = open("test.tcl", "w")

tclFile.write("run 1000s;\nexit")

prjFile = open("test.prj", "w")

for root, dirs, files in os.walk(r"C:\Users\霖\Desktop\Debug\_Program"):

for fileName in files:

if re.match(r"[\w]\*\.v", fileName):

prjFile.write("Verilog work " + root + "\\" + fileName + "\n")

tclFile.close()

prjFile.close()

os.environ['XILINX'] = r"D:\ISESetup\14.7\ISE\_DS\ISE"

# 命令行运行ISim

os.system(

r"D:\ISESetup\14.7\ISE\_DS\ISE\bin\nt64\fuse --nodebug --prj test.prj -o testmips.exe mips\_tb >CompileLog.txt")

os.system("testmips.exe -nolog -tclbatch test.tcl >myoutput.txt")

#### 3. 正确性判定脚本

num = input()

# diff.txt 为存储不同内容文件

# 以读取方式打开两个txt文件

file1 = open('output' + num + '.txt', "r")

file2 = open('myoutput' + num + '.txt', "r")

# 读取两个txt文件

text1 = file1.read()

text2 = file2.read()

file1.close()

file2.close()

# 将两个文件中内容按行分隔开

line1 = text1.strip('\n').split('\n')

line2 = text2.strip('\n').split('\n')

# 以读取方式打开 diff.txt 文件

outfile = open("diff" + num + ".txt", "w")

judge = True

if len(line1) != len(line2):

outfile.write("diff in length.\n")

outfile.write("len(output" + num + ") is {}.\n".format(len(line1)))

outfile.write("len(myoutput" + num + ") is {}.\n".format(len(line2)))

judge = False

i = 0

while i < min(len(line1), len(line2)):

if line1[i] != line2[i]:

if ''.join(line1[i].split()) != ''.join(line2[i].split()):

outfile.write("diff in len(" + str(i + 1) + ").\n")

judge = False

i += 1

print("核对结束")

if judge:

print("完全相同")

## 三、思考题

### （一）

根据你的理解，在下面给出的 DM 的输入示例中，地址信号 addr 位数为什么是 [11:2] 而不是 [9:0]？这个 addr 信号又是从哪里来的？

在我们设计的CPU中，DM时32bit\*1024字，也就是说我们需要10位的地址信号来确定某个存储数据的存储位置。

在MIPS中，数字时按字存储的，两个相邻数字的地址之差为4字节，这要求我们在存取地址时指令必须是字对齐的，即必须是4的倍数；然而当我们用Verilog语言实现CPU时，我们是申请了1024个reg[31:0]的空间，数字在这1024个空间中是紧密排列存储的，相邻两块空间的地址之差为1，所以我们要将地址信号除以4，也就是略去最后两位，这样才能使得Verilog设计的CPU中数据存储地址和MARS中数据的存储地址形成一一对应关系。

特别地，对于sh、lh、sb、lb等存取半字或者一个字节指令，我们也可以同样的将地址信号左移两位，先一次性取出RAM中4字节中的数据，再根据MIPS指令集中的方法，取byte=Addr1或byte=Addr1……0，根据byte的值进行数据的拼接或者扩展，再将拼接或扩展后的32位数据一次性存储到RAM中或者输出。

### （二）

思考 Verilog 语言设计控制器的译码方式，给出代码示例，并尝试对比各方式的优劣。

代码举例：

从指令出发：

if(command[5:0] == `addu) begin

npcop = 0; wrsel = 1;

wdsel = 0; rfwe = 1;

extop = 0; asel = 0;

bsel = 0; aluop = `AaddB;

dmwe = 0; dmtype = 0;

end

if(command[5:0] == `subu) begin

npcop = 0; wrsel = 1;

wdsel = 0; rfwe = 1;

extop = 0; asel = 0;

bsel = 0; aluop = `AsubB;

dmwe = 0; dmtype = 0;

end

从控制信号出发：

assign wrsel = (command[5:0] == `addu) + (command[5:0] == `subu);

assign rfwe = (command[5:0] == `addu) + (command[5:0] == `subu);

assign dmwe = (command[5:0] == `sw);

assign aluop[0] =

assign aluop[1] =

第一种方式，从指令出发，记录下指令对应的控制信号如何取值。我的CPU实现采用了这种方法，我认为这种方法更具有规范化的特点，即每种指令都需要考虑固定个数的控制信号取值，增加指令时可以参考甚至直接复制已经写好的指令的代码模式，并更改出恰当的控制信号输出即可。同时写出的代码清晰易读。当我们在测试的过程总发现某一条指令的运行结果错误时，我们可以快速找出与该条指令有关的所有控制信号取值，而第二种方法查找起来则比较困难。缺点是当我们发现需要添加某个控制信号时，我们需要在每一个指令中都添加该信号，操作起来比较麻烦。

第二种方式，从控制信号出发，记录下控制信号每种取值所对应的指令。优点在于便于添加控制信号，对于一些只有个别指令需要的控制信号，这种实现方法比较方便。然而在debug某个指令时，需要挨个控制信号查找该指令是否存在，不方便。而且这种实现需要依次对控制信号的每一位进行判断，当控制信号位数较多时（比如本CPU中的aluop信号为5位），我们或者写出五句assign赋值语句依次对aluop的0~4位进行赋值，或者依次判断aluop的每一位再进行拼接，总之需要对信号的每一位依次做出赋值，这种实现远不如第一种方法的直接对多位信号赋值方便。

就我个人而言，我更喜欢第一种实现方法。其一是代码格式化，整洁美观，有固定的模式，易于指令的添加和扩展；其二是对于当前CPU支持的指令不太多的情况下，controller输出的控制信号并不太多，第一种方法增量开发也比较容易，同时也容易寻找出单条指令产生的bug。

### （三）

在相应的部件中，reset 的优先级比其他控制信号（不包括 clk 信号）都要高，且相应的设计都是同步复位。清零信号 reset 所驱动的部件具有什么共同特点？

清零信号reset所驱动的部件（包括PC，GRF，DM）都是具有存储功能的时序逻辑部件，同步复位信号有效且位于时钟上升沿时，三个reset驱动的部件均回复至初始值（PC初始值为0x3000，GRF和DM的初始值为0）。在CPU开始处理指令信号之前，复位信号需要置为1将存储器中的数据均清零，从而保证后续计算结果的正确性。

### （四）

C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理，这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的 Operation 部分 。

通过阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》addi和addiu指令的Operation部分，我们可以看出addi与addiu的唯一区别就是是否判断溢出。当计算结果溢出时，对于addi指令，不向GPR[rt]寄存器写入数值，并产生溢出错误，而addiu指令则对溢出的计算结果进行取模运算，将结果写入GPR[rt]中，不会产生溢出错误。所以在忽略溢出的前提下，addi和addiu时等价的。同理，add和addu也是等价的。

### （五）

根据自己的设计说明单周期处理器的优缺点。

优点：单周期处理器设计简单，仅仅需要较少数量和种类的元件就能达到处理指令信号的功能。同时不涉及指令冲突的问题，相对流水线处理器运行较为稳定。

缺点：处理器执行效率低，由于木桶效应，时钟的最大频率取决于执行时间最长的指令（目前我已知的理论执行时间最长的指令是lw），且一个时钟周期只能执行一条指令，这就使得单周期处理器执行效率低。

附录一 典型测试样例

Mips代码：

ori $at,123

ori $v0,456 # 构造正数

lui $v0,0xffff

ori $v1,$v0,456

lui $a0,0xffff

ori $a1,$a0,795 # 构造负数

addu $a2,$at,$v0 # 正正

addu $a3,$v1,$a1 # 负负

addu $t0,$a2,$a3 # 正负

addu $t1,$a3,$a2 # 负正

subu $t2,$a2,$at # 正正

subu $t3,$a1,$a3 # 负负

subu $t4,$a2,$a3 # 正负

subu $t5,$a3,$t4 # 负正

ori $t6,0

ori $t7,1

nop

sw $t2,0($t6)

sw $t3,8($t6)

sw $t4,12($t6)

sw $t5,4($t6)

ori $k0,0

ori $k1,1

nop

lw $s0,12($t6)

lw $s1,8($t6)

lw $s2,4($t6)

lw $s3,0($t6)

ori $v1,4

ori $k0,4

subu $k1,$k1,$k1

loop:

beq $k0,$k1,end

subu $s1,$s1,$s7

addu $k1,$k0,$k1

j loop

func1:

subu $t8,$t7,$s0

addu $t5,$t9,$s1

addu $k0,$k0,$k1

jr $ra

func2:

addu $t6,$t6,$t7

addu $s4,$s0,$s1

subu $s5,$t0,$s2

addu $s7,$t9,$s3

beq $t6,$t7,func2

sw $ra,0($sp)

addu $sp,$sp,$v1

jal func1

subu $sp,$sp,$v1

lw $ra,0($sp)

jr $ra

end:

jal func2

机器码：

3421007b

344201c8

3c02ffff

344301c8

3c04ffff

3485031b

00223021

00653821

00c74021

00e64821

00c15023

00a75823

00c76023

00ec6823

35ce0000

35ef0001

00000000

adca0000

adcb0008

adcc000c

adcd0004

375a0000

377b0001

00000000

8dd0000c

8dd10008

8dd20004

8dd30000

34630004

375a0004

037bd823

135b0012

02378823

035bd821

08000c1f

01f0c023

03316821

035bd021

03e00008

01cf7021

0211a021

0112a823

0333b821

11cffffb

afbf0000

03a3e821

0c000c23

03a3e823

8fbf0000

03e00008

0c000c27

Isim控制台输出结果：

@00003000: $ 1 <= 0000007b

@00003004: $ 2 <= 000001c8

@00003008: $ 2 <= ffff0000

@0000300c: $ 3 <= ffff01c8

@00003010: $ 4 <= ffff0000

@00003014: $ 5 <= ffff031b

@00003018: $ 6 <= ffff007b

@0000301c: $ 7 <= fffe04e3

@00003020: $ 8 <= fffd055e

@00003024: $ 9 <= fffd055e

@00003028: $10 <= ffff0000

@0000302c: $11 <= 0000fe38

@00003030: $12 <= 0000fb98

@00003034: $13 <= fffd094b

@00003038: $14 <= 00000000

@0000303c: $15 <= 00000001

@00003044: \*00000000 <= ffff0000

@00003048: \*00000008 <= 0000fe38

@0000304c: \*0000000c <= 0000fb98

@00003050: \*00000004 <= fffd094b

@00003054: $26 <= 00000000

@00003058: $27 <= 00000001

@00003060: $16 <= 0000fb98

@00003064: $17 <= 0000fe38

@00003068: $18 <= fffd094b

@0000306c: $19 <= ffff0000

@00003070: $ 3 <= ffff01cc

@00003074: $26 <= 00000004

@00003078: $27 <= 00000000

@00003080: $17 <= 0000fe38

@00003084: $27 <= 00000004

@000030c8: $31 <= 000030cc

@0000309c: $14 <= 00000001

@000030a0: $20 <= 0001f9d0

@000030a4: $21 <= fffffc13

@000030a8: $23 <= ffff0000

@0000309c: $14 <= 00000002

@000030a0: $20 <= 0001f9d0

@000030a4: $21 <= fffffc13

@000030a8: $23 <= ffff0000

@000030b0: \*00000000 <= 000030cc

@000030b4: $29 <= ffff01cc

@000030b8: $31 <= 000030bc

@0000308c: $24 <= ffff0469

@00003090: $13 <= 0000fe38

@00003094: $26 <= 00000008

@000030bc: $29 <= 00000000

@000030c0: $31 <= 000030cc

与MARS输出的标准结果完全相同