# 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，共包含F、D、E、M和W五级，支持的指令集包含{add、addu、sub、subu、slt、sltu、and、or、nor、xor、sllv、srlv、srav、sll、srl、sra、jr、jalr、mfhi、mflo、mthi、mtlo、mult、multu、div、divu、addi、addiu、slti、sltiu、andi、ori、xori、lui、sw、sh、sb、lw、lh、lhu、lb、lbu、beq、bne、bgtz、blez、bgez、bltz、j、jal、mfc0、mtc0、exet}共53条。为了实现这些功能，CPU主要包含了PC、FlowReg\_D、RF、NPC、CMP、EXT、controller、FlowReg\_E、ALU、multdiv、CP0、errorjudger、FlowReg\_M、FlowReg\_W、RiskSolveUnit，共15个模块，外设包括IM、DM、timecounter0、timecounter1、systembridge，共5个模块，在顶层模块文件mips.v中连接CPU和外设。

### （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 pcenable,

input wire [2:0] npcop,

input wire [31:0] pc\_D\_B,

input wire [31:0] pc\_D\_JJal,

input wire [31:0] pc\_D\_Jr,

input wire branch,

input wire jumpto4180,

output reg [31:0] currentpc\_F

);

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 | pcenable | I | 1 | 使能信号，为0时PC寄存器保持原有值不变，用于阻塞。 |
| 4 | npcop | I | 3 | 下一条指令计算方法选择信号 |
| 5 | PC\_D\_B | I | 32 | B类指令的地址 |
| 6 | PC\_D\_JJal | I | 32 | J和Jal指令的跳转地址 |
| 7 | PC\_D\_Jr | I | 32 | Jr指令的跳转地址 |
| 8 | branch | I | 1 | B类指令满足跳转条件信号，满足条件时信号为1 |
| 9 | jumpto4180 | I | 1 | 地址跳转到0x4180，高电平时F级指令地址跳转为0x4180，优先级最高 |
| 10 | currentpc\_F | O | 32 | 输入到指令寄存器和D级流水线寄存器的地址信号 |

npcop选择信号对应表

|  |  |
| --- | --- |
| npcop | 对应的npc计算方法 |
| 3’d0 | PC+4 |
| 3’d1 | PC\_B |
| 3’d2 | PC\_JJal |
| 3’d3 | PC\_Jr |
| 3’d4 | PC\_eret |

#### 2. FlowReg\_D

模块功能：D级流水线寄存器

模块定义：

module FlowReg\_D (

input wire clk,

input wire En\_D,

input wire reset,

input wire clear\_D,

input wire jumpto4180,

input wire wrongpc\_F,

input wire [31:0] command\_F,

input wire [31:0] commandAddr\_F, pc\_eret,

output reg wrongpc\_D,

output reg [31:0] command\_D,

output reg [31:0] commandAddr\_D

);

表二 D级流水线寄存器信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 | En\_D | I | 1 | D级寄存器使能 |
| 3 | reset | I | 1 | D级寄存器复位信号 |
| 4 | clear\_D | I | 1 | D级寄存器复位信号，用于清空延迟槽 |
| 5 | jumpto4180 | I | 1 | 跳转至0x4180信号 |
| 6 | wrongpc\_F | I | 1 | F级PC异常信号 |
| 7 | command\_F | I | 32 | F级指令信号 |
| 8 | commandAddr\_F | I | 32 | F级当前指令地址 |
| 9 | pc\_eret | I | 32 | CP0 EPC寄存器中保存的指令地址 |
| 10 | wrongpc\_D | I | 1 | D级PC异常信号 |
| 11 | Command\_D | O | 32 | D级指令信号 |
| 12 | commandAddr\_D | O | 32 | D级当前指令地址 |

#### 3. 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] rfwd,

input wire [31:0] commandAddr\_W,

output wire [31:0] rfrd1,

output wire [31:0] rfrd2

);

表三 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rfrd1 |
| 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rfrd2 |
| 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 | rfwd | I | 32 | 32位数据输入信号 |
| 8 | commandAddr\_W | I | 32 | W级当前指令地址，评测需要 |
| 9 | rfrd1 | O | 32 | 输出A1指定的寄存器中的32位数据 |
| 10 | rfrd2 | O | 32 | 输出A2指定的寄存器中的32位数据 |

注 RF采用内部转发，W级写入RF的数据对D级可见。

实现方法为：

assign rfrd1 = (a1 == a3 && a3 != 5'd0 && rfwe == 1'd1) ? rfwd : register[a1];

assign rfrd2 = (a2 == a3 && a3 != 5'd0 && rfwe == 1'd1) ? rfwd : register[a2];

#### 4. 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] commandAddr\_D,

input wire [31:0] command\_D,

input wire [31:0] rfrd1\_D,

output wire [31:0] pc\_D\_B,

output wire [31:0] pc\_D\_JJal,

output wire [31:0] pc\_D\_Jr,

output wire [31:0] pc8\_D

);

表四 NPC信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | commandAddr\_D | I | 32 | D级当前指令地址 |
| 2 | command\_D | I | 32 | D级当前指令 |
| 3 | rfrd1\_D | I | 32 | D级RF的RD1信号通路中数据，Jr指令的跳转地址 |
| 4 | pc\_D\_B | O | 32 | 计算出的B类指令NPC |
| 5 | pc\_D\_JJal | O | 32 | 计算出的Jal/J指令NPC |
| 6 | pc\_D\_Jr | O | 32 | 计算出的Jr指令NPC |
| 7 | pc8\_D | O | 32 | 当前指令+8后的地址，Jal指令向$ra寄存器存储该地址 |

#### 5. CMP

模块功能：比较器

模块定义：

module CMP (

input wire [31:0] cmpa,

input wire [31:0] cmpb,

input wire [3:0] cmpop,

output reg branch\_true

);

表五 CMP信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | cmpa | I | 32 | 进行比较的第一个操作数 |
| 2 | cmpb | I | 32 | 进行比较的第二个操作数 |
| 3 | cmpop | I | 4 | 比较操作选择信号 |
| 4 | branch\_true | O | 1 | 是否进行B类跳转 |

#### 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] extres

);

表六 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)，立即数10-6位零扩展 |
| 3 | EXTRes | O | 32 | 扩展后的数据 |

#### 7. FlowReg\_E

模块功能：E级流水线寄存器

模块定义：

module FlowReg\_E (

input wire clk, CLR\_E, rfwe\_D, dmwe\_D, reset, startmd\_D, unknowinstr\_D, wrongpc\_D, jumpto4180,

input wire [1:0] rfwdsel\_D, rfwrsel\_D, asel\_D, bsel\_D,

input wire [2:0] tnew\_D, dmtype\_D,

input wire [3:0] multdivop\_D,

input wire [4:0] aluop\_D, commandtype\_D, read1addr\_D, read2addr\_D,

input wire [31:0] rfrd1\_D, rfrd2\_D, pc8\_D, extres\_D, command\_D, commandAddr\_D, pc\_eret,

output reg rfwe\_E, dmwe\_E, startmd\_E, unknowinstr\_E, wrongpc\_E,

output reg [1:0] rfwdsel\_E, rfwrsel\_E, asel\_E, bsel\_E,

output reg [2:0] tnew\_E, dmtype\_E,

output reg [3:0] multdivop\_E,

output reg [4:0] aluop\_E, commandtype\_E, read1addr\_E, read2addr\_E,

output reg [31:0] rfrd1\_E, rfrd2\_E, pc8\_E, extres\_E, command\_E, commandAddr\_E

);

表七 E级流水线寄存器信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 | CLR\_E | I | 1 | E级寄存器清零信号 |
| 3 | reset | I | 1 | E级寄存器复位信号 |
| 4 | rfwe\_D | I | 1 | D级指令RF写使能信号 |
| 5 | dmwe\_D | I | 1 | D级指令DM写使能信号 |
| 6 | startmd\_D | I | 1 | D级指令开始进行乘除运算信号 |
| 7 | unknowinstr\_D |  |  | D级指令未知信号 |
| 8 | wrongpc\_D |  |  | D级指令地址异常信号 |
| 9 | jumpto4180 |  |  | 跳转至0x4180信号 |
| 10 | rfwesel\_D | I | 2 | D级指令RF写入数据选择信号 |
| 11 | rfwrsel\_D | I | 2 | D级指令RF写入地址选择信号 |
| 12 | asel\_D | I | 2 | D级指令ALU A操作数选择信号 |
| 13 | bsel\_D | I | 2 | D级指令ALU B操作数选择信号 |
| 14 | dmtype\_D | I | 3 | D级指令DM写入类型选择信号 |
| 15 | tnew\_D |  |  | D级指令新数据产生时间 |
| 16 | multdivop\_D | I | 4 | D级指令乘除运算单元操作选择信号 |
| 17 | aluop\_D | I | 5 | D级指令 ALU操作选择信号 |
| 18 | commandtype\_D | I | 5 | D级指令类型 |
| 19 | read1addr\_D | I | 5 | D级指令 RF读取地址1 |
| 20 | read2addr\_D | I | 5 | D级指令 RF读取地址2 |
| 21 | rfrd1\_D | I | 32 | D级指令 RF读取数据1 |
| 22 | rfrd2\_D | I | 32 | D级指令 RF读取数据2 |
| 23 | pc8\_D | I | 32 | D级指令的地址+8 |
| 24 | extres\_D | I | 32 | D级指令立即数扩展结果 |
| 25 | command\_D | I | 32 | D级指令 |
| 26 | commandAddr\_D | I | 32 | D级指令的地址 |
| 27 | pc\_eret |  |  | CP0 EPC寄存器保存的异常指令地址 |
| 28 | rfwe\_E | O | 1 | E级指令RF写使能信号 |
| 29 | dmwe\_E | O | 1 | E级指令DM写使能信号 |
| 30 | startmd\_E |  |  | E级指令开始进行乘除运算信号 |
| 31 | unknowinstr\_E |  |  | E级指令未知信号 |
| 32 | wrongpc\_E |  |  | E级指令地址异常信号 |
| 33 | rfwesel\_E | O | 2 | E级指令RF写入数据选择信号 |
| 34 | rfwrsel\_E | O | 2 | E级指令RF写入地址选择信号 |
| 35 | asel\_E | O | 2 | E级指令ALU A操作数选择信号 |
| 36 | bsel\_E | O | 2 | E级指令ALU B操作数选择信号 |
| 37 | dmtype\_E | O | 3 | E级指令DM写入类型选择信号 |
| 38 | tnew\_E | O | 3 | E级指令Tnew信号 |
| 39 | multdivop\_E |  |  | E级指令乘除运算单元操作选择信号 |
| 40 | aluop\_E | O | 5 | E级指令 ALU操作选择信号 |
| 41 | commandtype\_E |  |  | E级指令乘除运算单元操作选择信号 |
| 42 | read1addr\_E | O | 5 | E级指令 RF读取地址1 |
| 43 | read2addr\_E | O | 5 | E级指令 RF读取地址2 |
| 44 | rfrd1\_E | O | 32 | E级指令 RF读取数据1 |
| 45 | rfrd2\_E | O | 32 | E级指令 RF读取数据2 |
| 46 | pc8\_E | O | 32 | E级指令的地址+8 |
| 47 | extres\_E | O | 32 | E级指令立即数扩展结果 |
| 48 | command\_E | O | 32 | E级指令 |
| 49 | commandAddr\_E | O | 32 | E级指令的地址 |

#### 8. 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 overflow,

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 | overflow | O | 1 | ALU计算结果溢出 |
| 5 | res | O | 32 | 运算结果 |

ALU运算定义表

|  |  |  |
| --- | --- | --- |
| 输入信号 | 宏定义 | 执行的操作 |
| 1 | `uAaddB | 两个操作数做加法(不检测溢出) |
| 2 | `uAsubB | 两个操作数做减法(不检测溢出) |
| 3 | `AaddB | 两个操作数做加法(检测溢出) |
| 4 | `AsubB | 两个操作数做减法(检测溢出) |
| 5 | `AltB | 第一个操作数小于第二个操作数（有符号） |
| 6 | `uAltB | 第一个操作数小于第二个操作数（无符号） |
| 7 | `AandB | 两个操作数做与运算 |
| 8 | `AorB | 两个操作数做或运算 |
| 9 | `AnorB | 两个操作数做或非运算 |
| 10 | `AxorB | 两个操作数做异或运算 |
| 11 | `BsllA | B逻辑左移A[4:0]位 |
| 12 | `BsrlA | B逻辑右移A[4:0]位 |
| 13 | `BsraA | B算术右移A[4:0]位 |
| 14 | `luiB | 将B的低16位加载到高16位，后面补零 |

#### 9. multdiv

模块功能：乘除法运算单元

模块定义：

module multdiv (

input wire clk, reset, start, jumpto4180,

input wire [3:0] multdivop,

input wire [31:0] a, b,

output wire busy,

output reg [31:0] registerhi,

output reg [31:0] registerlo

);

表九 multdiv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 | start | I | 1 | 开始进行乘除运算信号 |
| 4 | jumpto4180 | I | 1 | 跳转至0x4180信号 |
| 5 | multdivop | I | 4 | 乘除运算方法选择信号 |
| 6 | a | I | 32 | a操作数 |
| 7 | b | I | 32 | b操作数 |
| 8 | busy | O | 1 | 乘除运算单元繁忙信号 |
| 9 | registerhi | O | 32 | hi寄存器中数值 |
| 10 | registerlo | O | 32 | lo寄存器中数值 |

#### 10. CP0

模块功能：协处理器

模块定义：

module CP0 (

input wire clk, reset, isDelay,

input wire [4:0] errorcode,

input wire [5:0] interruptrequest,

input wire [31:0] command\_E, write\_data, pc\_E,

output wire jumpto4180, interruptout,

output wire [31:0] pc\_eret,

output wire [31:0] readfromcp

);

表十 CP0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 | isDelay | I | 1 | 当前指令是否为延迟槽指令 |
| 4 | errorcode | I | 5 | 当前指令的异常码 |
| 5 | interruptrequest | I | 6 | 六位中断信号 |
| 6 | command\_E | I | 32 | E级指令信号 |
| 7 | write\_data | I | 32 | 向CP0中寄存器写入的数据 |
| 8 | pc\_E | I | 32 | E级指令地址 |
| 9 | jumpto4180 | O | 1 | 跳转到0x4180 |
| 10 | interruptout | O | 1 | 响应外部中断信号时置1 |
| 11 | pc\_eret | O | 32 | EPC中保存的异常指令地址 |
| 12 | readfromcp | O | 32 | 从CP0模块中读到的数据 |

#### 11. errorjudger

模块功能：指令异常判断器

模块定义：

module errorjudger (

input wire unknowinstr, overflow, wrongpc\_E,

input wire [4:0] commandtype\_E,

input wire [31:0] command\_E, commandAddr\_E, alures,

output reg [4:0] errorcode

);

表十一 errorjudger信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
|  | unknowinstr | I | 1 | 未知指令信号 |
|  | overflow | I | 1 | 算术溢出信号 |
|  | wrongpc\_E | I | 1 | 地址异常信号 |
|  | commandtype\_E | I | 5 | E级指令类型 |
|  | command\_E | I | 32 | E级指令 |
|  | commandAddr\_E | I | 32 | E级指令地址 |
|  | alures | I | 32 | ALU计算指令 |
|  | errorcode | O | 5 | 异常码 |

#### 12. FlowReg\_M

模块功能：M级流水线寄存器

模块定义：

module FlowReg\_M (

input wire clk, rfwe\_E, dmwe\_E, reset,

input wire [1:0] rfwdsel\_E, rfwrsel\_E,

input wire [2:0] tnew\_E, dmtype\_E,

input wire [4:0] read2addr\_E, commandtype\_E,

input wire [31:0] ALUres\_E, registerhi\_E, registerlo\_E, wddm\_E, pc8\_E, command\_E, commandAddr\_E,

output reg rfwe\_M, dmwe\_M,

output reg [1:0] rfwdsel\_M, rfwrsel\_M,

output reg [2:0] tnew\_M, dmtype\_M,

output reg [4:0] read2addr\_M, commandtype\_M,

output reg [31:0] ALUres\_M, registerhi\_M, registerlo\_M, wddm\_M, pc8\_M, command\_M, commandAddr\_M

);

表十二 M级流水线寄存器信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | clk | I | 1 | M级寄存器时钟信号 |
| 2 | reset | I | 1 | M级寄存器复位信号 |
| 3 | rfwe\_E | I | 1 | E级指令RF写使能信号 |
| 4 | dmwe\_E | I | 1 | E级指令DM写使能信号 |
| 5 | rfwdsel\_E | I | 2 | E级指令RF写入数据选择信号 |
| 6 | rfwrsel\_E | I | 2 | E级指令RF写入地址选择信号 |
| 7 | dmtype\_E | I | 3 | E级指令DM写入类型选择信号 |
| 8 | tnew\_E | I | 3 | E级指令Tnew信号 |
| 9 | read2addr\_E | I | 5 | E级指令 RF读取地址2 |
| 10 | commandtype\_E | I | 5 | E级指令类型 |
| 11 | ALUres\_E | I | 32 | E级指令ALU运算结果 |
| 12 | registerhi\_E | I | 32 | E级指令hi寄存器数值 |
| 13 | registerlo\_E | I | 32 | E级指令lo寄存器数值 |
| 14 | wddm\_E | I | 32 | E级指令DM输入数据 |
| 15 | pc8\_E | I | 32 | E级指令地址+8 |
| 16 | command\_E | I | 32 | E级指令 |
| 17 | CommandAddr\_E | I | 32 | E级指令的地址 |
| 18 | rfwe\_M | O | 1 | M级指令RF写使能信号 |
| 19 | dmwe\_M | O | 1 | M级指令DM写使能信号 |
| 20 | rfwdsel\_M | O | 2 | M级指令RF写入数据选择信号 |
| 21 | rfwrsel\_M | O | 2 | M级指令RF写入地址选择信号 |
| 22 | dmtype\_M | O | 3 | M级指令DM写入类型选择信号 |
| 23 | tnew\_M | O | 3 | M级指令Tnew信号 |
| 24 | read2addr\_M | O | 5 | M级指令 RF读取地址2 |
| 25 | commandtype\_M | O | 5 | M级指令类型 |
| 26 | ALUres\_M | O | 32 | M级指令ALU运算结果 |
| 27 | registerhi\_M | O | 32 | M级指令hi寄存器数值 |
| 28 | registerlo\_M | O | 32 | M级指令lo寄存器数值 |
| 29 | wddm\_M | O | 32 | M级指令DM输入数据 |
| 30 | pc8\_M | O | 32 | M级指令地址+8 |
| 31 | command\_M | O | 32 | M级指令 |
| 32 | CommandAddr\_M | O | 32 | M级指令的地址 |

#### 13. FlowReg\_W

模块功能：W级流水线寄存器

模块定义：

module FlowReg\_W (

input wire clk, rfwe\_M, reset,

input wire [1:0] rfwdsel\_M, rfwrsel\_M,

input wire [4:0] commandtype\_M,

input wire [31:0] ALUres\_M, registerhi\_M, registerlo\_M, DMRes\_M, pc8\_M, commandAddr\_M, command\_M,

output reg rfwe\_W,

output reg [1:0] rfwdsel\_W, rfwrsel\_W,

output reg [4:0] commandtype\_W,

output reg [31:0] ALUres\_W, registerhi\_W, registerlo\_W, DMRes\_W, pc8\_W, commandAddr\_W, command\_W

);

表十三 E级流水线寄存器信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | clk | I | 1 | W级寄存器时钟信号 |
| 2 | reset | I | 1 | W级寄存器复位信号 |
| 3 | rfwe\_M | I | 1 | M级指令RF写使能信号 |
| 4 | rfwdsel\_M | I | 2 | M级指令RF写入数据选择信号 |
| 5 | rfwrsel\_M | I | 2 | M级指令RF写入地址选择信号 |
| 6 | commandtype\_M | I | 5 | M级指令类型 |
| 7 | ALUres\_M | I | 32 | M级指令ALU运算结果 |
| 8 | registerhi\_M | I | 32 | M级指令hi寄存器数值 |
| 9 | registerlo\_M | I | 32 | M级指令lo寄存器数值 |
| 10 | DMRes\_M | I | 32 | M级指令DM输出数据 |
| 11 | pc8\_M | I | 32 | M级指令地址+8 |
| 12 | commandAddr\_M | I | 32 | M级指令地址 |
| 13 | command\_M | I | 32 | M级指令 |
| 14 | rfwe\_W | O | 1 | W级指令RF写使能信号 |
| 15 | rfwdsel\_W | O | 2 | W级指令RF写入数据选择信号 |
| 16 | rfwrsel\_W | O | 2 | W级指令RF写入地址选择信号 |
| 17 | commandtype\_W | O | 5 | W级指令类型 |
| 18 | ALUres\_W | O | 32 | W级指令ALU运算结果 |
| 19 | registerhi\_W | O | 32 | W级指令hi寄存器数值 |
| 20 | registerlo\_W | O | 32 | W级指令lo寄存器数值 |
| 21 | DWRes\_W | O | 32 | W级指令DW输出数据 |
| 22 | pc8\_W | O | 32 | W级指令地址+8 |
| 23 | commandAddr\_W | O | 32 | W级指令地址 |
| 24 | command\_W | O | 32 | W级指令 |

#### 14. Controller

模块功能：控制器

模块定义：

module controller (

input wire [31:0] command,

output wire startmd\_D,

output wire [4:0] commandtype\_D,

output reg rfwe\_D, dmwe\_D, unknowinstr,

output reg [1:0] wrsel\_D, wdsel\_D, extop, asel\_D, bsel\_D,

output reg [2:0] npcop, tuse\_Drs, tuse\_Drt, tnew\_D, dmtype\_D,

output reg [3:0] cmpop, multdivop\_D,

output reg [4:0] aluop\_D

);

表十四 控制器信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 | startmd\_D | O | 1 | D级指令开始进行乘除运算信号 |
| 3 | rfwe\_D | O | 1 | D级指令RF写使能信号 |
| 4 | dmwe\_D | O | 1 | D级指令DM写使能信号 |
| 5 | wrsel\_D | O | 2 | D级指令RF写入地址选择信号 |
| 6 | wdsel\_D | O | 2 | D级指令RF写入数据选择信号 |
| 7 | extop | O | 2 | D级指令位扩展方法 |
| 8 | asel\_D | O | 2 | D级指令ALU A操作数选择信号 |
| 9 | bsel\_D | O | 2 | D级指令ALU B操作数选择信号 |
| 10 | dmtype\_D | O | 3 | D级指令 DM写入类型选择信号 |
| 11 | unknowinstr | O | 1 | D级指令为未知指令时置1 |
| 12 | npcop | O | 3 | D级指令的下一条指令地址计算操作类型 |
| 13 | tuse\_Drs | O | 3 | D级指令使用RF的rs地址对应的数据的时间 |
| 14 | tuse\_Drt | O | 3 | D级指令使用RF的rt地址对应的数据的时间 |
| 15 | tnew\_D | O | 3 | D级指令产生新的写入RF的时间 |
| 16 | cmpop | O | 4 | D级指令cmp操作类型选择信号 |
| 17 | multdivop\_D | O | 4 | D级指令乘除运算单元操作选择信号 |
| 18 | aluop\_D | O | 5 | D级指令ALU操作类型选择信号 |
| 19 | commandtype\_D | O | 5 | D级指令类型 |

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。

#### 15. RiskSolveUnit

模块功能：冲突化解单元

模块定义：

module RiskSolveUnit (

input wire rfwe\_E, rfwe\_M, rfwe\_W, busy\_E, startmd\_E,

input wire [4:0] read1addr\_D, read2addr\_D, read1addr\_E, read2addr\_E, read2addr\_M, writeaddr\_E, writeaddr\_M, writeaddr\_W,

input wire [4:0] commandtype\_D, commandtype\_E, commandtype\_M, commandtype\_W,

output reg [1:0] forward\_CMPa\_D, forward\_CMPb\_D, forward\_ALUa\_E, forward\_ALUb\_E, forward\_DM\_M,

input wire [2:0] tuse\_Drs, tuse\_Drt, tnew\_E , tnew\_M,

output reg stall\_F, stall\_D, clear\_D, flush\_E,

input wire [31:0] command\_D, command\_E, command\_M, command\_W

);

表十五 冲突化解单元信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | rfwe\_E | I | 1 | E级指令RF写使能信号 |
| 2 | rfwe\_M | I | 1 | M级指令RF写使能信号 |
| 3 | rfwe\_W | I | 1 | W级指令RF写使能信号 |
| 4 | busy\_E | I | 1 | E级乘除单元繁忙指令 |
| 5 | startmd\_E | I | 1 | E级乘除单元开始运算指令 |
| 6 | read1addr\_D | I | 5 | 即将参与D级CMP比较的A操作数对应的寄存器地址 |
| 7 | read2addr\_D | I | 5 | 即将参与D级CMP比较的B操作数对应的寄存器地址 |
| 8 | read1addr\_E | I | 5 | 即将参与E级ALU运算的A操作数对应的寄存器地址 |
| 9 | read2addr\_E | I | 5 | 即将参与E级ALU运算的B操作数对应的寄存器地址 |
| 10 | read2addr\_M | I | 5 | 即将参与M级DM写入的数据对应的寄存器地址 |
| 11 | writeaddr\_E | I | 5 | E级结果写入的RF地址 |
| 12 | writeaddr\_M | I | 5 | M级结果写入的RF地址 |
| 13 | writeaddr\_W | I | 5 | W级结果写入的RF地址 |
| 14 | commandtype\_D | I | 5 | D级指令类型 |
| 15 | commandtype\_E | I | 5 | E级指令类型 |
| 16 | commandtype\_M | I | 5 | M级指令类型 |
| 17 | commandtype\_W | I | 5 | W级指令类型 |
| 18 | forward\_CMPa\_D | O | 2 | D级CMP A操作数转发选择信号 |
| 19 | forward\_CMPb\_D | O | 2 | D级CMP B操作数转发选择信号 |
| 20 | forward\_ALUa\_E | O | 2 | E级ALU A操作数转发选择信号 |
| 21 | forward\_ALUb\_E | O | 2 | E级ALU B操作数转发选择信号 |
| 22 | forward\_DM\_M | O | 2 | M级 DM写入数据转发选择信号 |
| 上为判断转发的信号，下为判断阻塞的信号 | | | | |
| 23 | tuse\_Drs | I | 3 | D级指令GRF[rs]的使用时间 |
| 24 | tuse\_Drt | I | 3 | D级指令GRF[rt]的使用时间 |
| 25 | tnew\_E | I | 3 | E级指令得到新的向RF写入的数据的时间 |
| 26 | tnew\_M | I | 3 | M级指令得到新的向RF写入的数据的时间 |
| 27 | stall\_F | O | 1 | PC寄存器冻结信号 |
| 28 | stall\_D | O | 1 | D级流水线寄存器冻结信号 |
| 29 | flush\_E | O | 1 | E级流水线寄存器清除信号 |
| 30 | command\_D | I | 32 | D级指令信号 |
| 31 | command\_E | I | 32 | E级指令信号 |
| 32 | command\_M | I | 32 | M级指令信号 |
| 33 | command\_W | I | 32 | W级指令信号 |

#### 16. systembridge

模块功能：系统桥

模块定义：

module systembridge (

input wire we\_cpu,

input wire [31:0] addr\_cpu,

input wire [31:0] data\_dm, data\_timer0, data\_timer1,

output wire we\_dm, we\_timer0, we\_timer1,

output wire [31:0] addr\_dm, addr\_timer0, addr\_timer1,

output wire [31:0] data\_to\_cpu

);

表十六 冲突化解单元信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名称 | 数据方向 | 位数 | 功能描述 |
| 1 | we\_cpu | I | 1 | CPU传来的写使能信号 |
| 2 | addr\_cpu | I | 32 | CPU传来的写地址信号 |
| 3 | data\_dm | I | 32 | DM传来的数据 |
| 4 | data\_time0 | I | 32 | timer0传来的数据 |
| 5 | data\_time1 | I | 32 | timer1传来的数据 |
| 6 | we\_dm | O | 1 | DM写使能信号 |
| 7 | we\_time0 | O | 1 | timer0写使能信号 |
| 8 | we\_time1 | O | 1 | timer1写使能信号 |
| 9 | addr\_dm | O | 32 | DM写入地址 |
| 10 | addr\_time0 | O | 32 | timer0写入地址 |
| 11 | addr\_time1 | O | 32 | timer写入地址 |
| 12 | data\_to\_cpu | O | 32 | 传给CPU的数据 |

#### 17. timecounter

模块功能：计时器

模块定义：

module timecounter(

input clk,

input reset,

input [31:2] addr,

input we,

input [31:0] Din,

output [31:0] Dout,

output IRQ

);

表十七 计时器信号定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 | addr | I | 30 | 写入地址信号 |
| 4 | we | I | 1 | 写使能信号 |
| 5 | Din | I | 32 | 输入的数据 |
| 6 | Dout | O | 32 | 读出的数据 |
| 7 | IRQ | O | 1 | 计时器中断信号 |

### （三）控制模块设计

控制器的设计本质是译码的过程，根据指令信号定位到具体的指令，并根据指令的操作得到对应的控制信号。

P7搭建的CPU能够处理53条指令，与之前搭建的CPU相比，支持的指令数量有显著增加。为了简化控制单元设计，我采用了指令分类的方法，根本目的是将控制信号大部分相同的指令的控制信号在一个if块中输出。

#### 指令信号解码

R型指令（add addu sub subu slt sltu and or nor xor sllv srav sll srl sra jr jalr mfhi mflo mthi mtlo mthi mtlo mult multu div divu）：前六位均为0，可以通过后五位编码

I J型指令（addi addiu slti sltiu andi ori xori lui sw sh sb lw lh lhu lb lbu beq bne bgtz blez j jal ）：可以直接通过前六位编码

Nop指令：全零

特殊指令：bgez、bltz op = 6’b000001 根据指令的20-16位编码

mtc0 11'b01000000100

mfc0 11'b01000000000

eret 32'b01000010000000000000000000011000

PART1 OPCode == 0

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

// double register

`define add 6'b100000 // op == 0

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

`define sub 6'b100010 // op == 0

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

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

`define sltu 6'b101011 // op == 0

`define and 6'b100100 // op == 0

`define or 6'b100101 // op == 0

`define nor 6'b100111 // op == 0

`define xor 6'b100110 // op == 0

`define sllv 6'b000100 // op == 0

`define srlv 6'b000110 // op == 0

`define srav 6'b000111 // op == 0

// shift

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

`define srl 6'b000010 // op == 0

`define sra 6'b000011 // op == 0

// jump register

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

// jump register and link

`define jalr 6'b001001 // op == 0

// mult and div

`define mfhi 6'b010000 // op == 0

`define mflo 6'b010010 // op == 0

`define mthi 6'b010001 // op == 0

`define mtlo 6'b010011 // op == 0

`define mult 6'b011000 // op == 0

`define multu 6'b011001 // op == 0

`define div 6'b011010 // op == 0

`define divu 6'b011011 // op == 0

PART2 OPCode != 0

// register and imm

`define addi 6'b001000 // op != 0

`define addiu 6'b001001 // op != 0

`define slti 6'b001010 // op != 0

`define sltiu 6'b001011 // op != 0

`define andi 6'b001100 // op != 0

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

`define xori 6'b001110 // op != 0

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

// save and load

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

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

`define sb 6'b101000 // op != 0

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

`define lh 6'b100001 // op != 0

`define lhu 6'b100101 // op != 0

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

`define lbu 6'b100100 // op != 0

// double register jump

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

`define bne 6'b000101 // op != 0

// single register jump

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

`define blez 6'b000110 // op != 0

// jump

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

// jump and link

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

PART3 special command

OPCode = 6’b000001

`define bgez 5'b00001

`define bltz 5'b00000

`define mtc0 11'b01000000100

`define mfc0 11'b01000000000

`define eret 32'b01000010000000000000000000011000

#### 指令分类

根据指令的操作类型，我将指令分为一下几类：

**a.** jumpandlink(跳转并链接类)

此类指令特点是向寄存器写入PC+8，在RiskSolveUnit判断转发回写数据时会用到。

包含：jal、jalr

b. calmudv（乘除法计算类）

此类指令的特点是需要在乘除运算单元做运算，且会使得乘除运算单元busy一定的时钟周期

包含：mult、multu、div、divu

c. readmudv(读取乘除运算结果类)

此类指令的特点是会读取hi或lo寄存器中的数值，但不会使得乘除运算单元busy

包含：mflo、mfhi

d. setmudv(对hi或lo寄存器赋值类)

此类指令的特点是会对hi或lo寄存器赋特定的值，需要在乘除运算单元进行运算，但不会使乘除运算单元busy

包含：mtlo、mthi

e. caldoubleregister(双寄存器运算类)

此类指令的特点是两个寄存器中的数值进行运算，并将结果写入到第三个寄存器中。

包含指令：add、addu、sub、subu、slt、sltu、and、or、nor、xor、sllv、srlv、srav

f. shift(移位(立即数位）类)

包含指令：sll、srl、sra

g. calregisterimmsign(寄存器和符号扩展立即数计算类)

包含指令：addi、addiu、slti、sltiu

h. calregisterimmzero(寄存器和零扩展立即数计算类)

包含指令：andi、ori、xori、lui

i. save(存储类)

包含指令：sw、sh、sb

j. load(载入类)

包含指令：lw、lh、lhu、lb、lbu

k. aboutc0 (存取c0类)

包含指令：mfc0、mtc0

l. 未分类指令

nop、j、jr、（beq、bne）、（bgez、bltz）、（bgtz、blez）

#### 控制信号输出

这部分是各指令对应的控制信号真值表，与前几个Project不同的是，P6采用的是根据指令类输出控制信号，代码实现举例：

if(commandtype\_D == `caldoubleregister)begin

npcop = 0; wrsel\_D = 1;

wdsel\_D = 0; rfwe\_D = 1;

extop = 0; cmpop = `None;

asel\_D = 0; bsel\_D = 0;

dmwe\_D = 0; dmtype\_D = `None;

tnew\_D = 2; tuse\_Drs = 1;

tuse\_Drt = 1;

case(command[5:0])

`add : begin aluop\_D = `AaddB; end

`addu : begin aluop\_D = `AaddB; end

`sub : begin aluop\_D = `AsubB; end

`subu : begin aluop\_D = `AsubB; end

`slt : begin aluop\_D = `AltB; end

`sltu : begin aluop\_D = `uAltB; end

`and : begin aluop\_D = `AandB; end

`or : begin aluop\_D = `AorB; end

`nor : begin aluop\_D = `AnorB; end

`xor : begin aluop\_D = `AxorB; end

`sllv: begin aluop\_D = `BsllA; end

`srlv : begin aluop\_D = `BsrlA; end

`srav: begin aluop\_D = `BsraA; end

default : begin aluop\_D = 5'd0; end

endcase

end

下面列出各个指令的控制信号真值表：

表十八 控制信号逻辑真值表（一）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | npcop | wrsel\_D | wdsel\_D | rfwe\_D | extop\_D | asel\_D | bsel\_D |
| add | 0 | 1 | 0 | 1 | x | 0 | 0 |
| addu | 0 | 1 | 0 | 1 | x | 0 | 0 |
| sub | 0 | 1 | 0 | 1 | x | 0 | 0 |
| subu | 0 | 1 | 0 | 1 | x | 0 | 0 |
| slt | 0 | 1 | 0 | 1 | x | 0 | 0 |
| sltu | 0 | 1 | 0 | 1 | x | 0 | 0 |
| and | 0 | 1 | 0 | 1 | x | 0 | 0 |
| or | 0 | 1 | 0 | 1 | x | 0 | 0 |
| nor | 0 | 1 | 0 | 1 | x | 0 | 0 |
| xor | 0 | 1 | 0 | 1 | x | 0 | 0 |
| sllv | 0 | 1 | 0 | 1 | x | 0 | 0 |
| srlv | 0 | 1 | 0 | 1 | x | 0 | 0 |
| srav | 0 | 1 | 0 | 1 | x | 0 | 0 |
| sll | 0 | 1 | 0 | 1 | 2 | 1 | 0 |
| slr | 0 | 1 | 0 | 1 | 2 | 1 | 0 |
| sra | 0 | 1 | 0 | 1 | 2 | 1 | 0 |
| jr | 3 | x | x | 0 | 0 | 0 | 0 |
| jalr | 3 | 1 | 2 | 1 | 0 | 0 | 0 |
| mfhi | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| mflo | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| mthi | 0 | x | x | 0 | x | 0 | 0 |
| mtlo | 0 | x | x | 0 | x | 0 | 0 |
| mult | 0 | x | x | 0 | x | 0 | 0 |
| multu | 0 | x | x | 0 | x | 0 | 0 |
| div | 0 | x | x | 0 | x | 0 | 0 |
| divu | 0 | x | x | 0 | x | 0 | 0 |
| addi | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| addiu | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| slti | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| sltiu | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| andi | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| ori | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| xori | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| lui | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| sw | 0 | x | x | 0 | 1 | 0 | 1 |
| sh | 0 | x | x | 0 | 1 | 0 | 1 |
| sb | 0 | x | x | 0 | 1 | 0 | 1 |
| lw | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| lh | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| lhu | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| lb | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| lbu | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| beq | 1 | x | x | 0 | 1 | 0 | 0 |
| bne | 1 | x | x | 0 | 1 | 0 | 0 |
| bgtz | 1 | x | x | 0 | 1 | 0 | 0 |
| blez | 1 | x | x | 0 | 1 | 0 | 0 |
| j | 2 | x | x | 0 | x | 0 | 0 |
| jal | 2 | 2 | 2 | 1 | x | 0 | 0 |
| bgtz | 1 | x | x | 0 | 1 | 0 | 0 |
| bltz | 1 | x | x | 0 | 1 | 0 | 0 |
| nop | 0 | x | x | 0 | x | x | x |
| madd | 0 | x | x | 0 | x | x | x |
| maddu | 0 | x | x | 0 | x | x | x |
| msub | 0 | x | x | 0 | x | x | x |
| msubu | 0 | x | x | 0 | x | x | x |
| mfc0 | 0 | x | x | 1 | x | x | x |
| mtc0 | 0 | x | x | 0 | x | x | x |
| eret | 4 | x | x | 0 | x | x | x |

表十九 控制信号逻辑真值表（二）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | aluop\_D | dmwe\_D | dmtype\_D | tnew\_D | tuse\_Drs | tuse\_Drt | cmp\_op |
| add | 1 | 0 | x | 2 | 1 | 1 | x |
| addu | 1 | 0 | x | 2 | 1 | 1 | x |
| sub | 2 | 0 | x | 2 | 1 | 1 | x |
| subu | 2 | 0 | x | 2 | 1 | 1 | x |
| slt | 3 | 0 | x | 2 | 1 | 1 | x |
| sltu | 4 | 0 | x | 2 | 1 | 1 | x |
| and | 5 | 0 | x | 2 | 1 | 1 | x |
| or | 6 | 0 | x | 2 | 1 | 1 | x |
| nor | 7 | 0 | x | 2 | 1 | 1 | x |
| xor | 8 | 0 | x | 2 | 1 | 1 | x |
| sllv | 9 | 0 | x | 2 | 1 | 1 | x |
| srlv | 10 | 0 | x | 2 | 1 | 1 | x |
| srav | 11 | 0 | x | 2 | 1 | 1 | x |
| sll | 9 | 0 | x | 2 | 5 | 1 | x |
| slr | 10 | 0 | x | 2 | 5 | 1 | x |
| sra | 11 | 0 | x | 2 | 5 | 1 | x |
| jr | 0 | 0 | x | 0 | 0 | 5 | x |
| jalr | 0 | 0 | x | 2 | 0 | 5 | x |
| mfhi | 0 | 0 | x | 2 | 5 | 5 | x |
| mflo | 0 | 0 | x | 2 | 5 | 5 | x |
| mthi | 0 | 0 | x | 0 | 1 | 5 | x |
| mtlo | 0 | 0 | x | 0 | 1 | 5 | x |
| mult | 0 | 0 | x | 0 | 1 | 1 | x |
| multu | 0 | 0 | x | 0 | 1 | 1 | x |
| div | 0 | 0 | x | 0 | 1 | 1 | x |
| divu | 0 | 0 | x | 0 | 1 | 1 | x |
| addi | 1 | 0 | x | 2 | 1 | 5 | x |
| addiu | 1 | 0 | x | 2 | 1 | 5 | x |
| slti | 3 | 0 | x | 2 | 1 | 5 | x |
| sltiu | 4 | 0 | x | 2 | 1 | 5 | x |
| andi | 5 | 0 | x | 2 | 1 | 5 | x |
| ori | 6 | 0 | x | 2 | 1 | 5 | x |
| xori | 8 | 0 | x | 2 | 1 | 5 | x |
| lui | 12 | 0 | x | 2 | 1 | 5 | x |
| sw | 1 | 1 | 1 | 0 | 1 | 2 | x |
| sh | 1 | 1 | 2 | 0 | 1 | 2 | x |
| sb | 1 | 1 | 4 | 0 | 1 | 2 | x |
| lw | 1 | 0 | 1 | 3 | 1 | 5 | x |
| lh | 1 | 0 | 2 | 3 | 1 | 5 | x |
| lhu | 1 | 0 | 3 | 3 | 1 | 5 | x |
| lb | 1 | 0 | 4 | 3 | 1 | 5 | x |
| lbu | 1 | 0 | 5 | 3 | 1 | 5 | x |
| beq | 0 | 0 | x | 0 | 0 | 0 | 1 |
| bne | 0 | 0 | x | 0 | 0 | 0 | 2 |
| bgtz | 0 | 0 | x | 0 | 0 | 5 | 3 |
| blez | 0 | 0 | x | 0 | 0 | 5 | 6 |
| j | 0 | 0 | x | 0 | 5 | 5 | x |
| jal | 0 | 0 | x | 2 | 5 | 5 | x |
| bgez | 0 | 0 | x | 0 | 0 | 5 | 4 |
| bltz | 0 | 0 | x | 0 | 0 | 5 | 5 |
| nop | 0 | x | 0 | 0 | 5 | 5 | x |
| madd | 0 | 0 | x | 0 | 1 | 1 | x |
| maddu | 0 | 0 | x | 0 | 1 | 1 | x |
| msub | 0 | 0 | x | 0 | 1 | 1 | x |
| msubu | 0 | 0 | x | 0 | 1 | 1 | x |
| mfc0 | 0 | 0 | x | 2 | 5 | 5 | x |
| mtc0 | 0 | 0 | x | 0 | 5 | 1 | x |
| eret | 0 | 0 | x | 0 | 5 | 5 | x |

### （四）CP0模块设计

CP0模块在CPU中的功能是记录CPU当前运行的状态并根据CPU当前状态和中断信号以及errorjudge模块传来的error信号，判断CPU在下一个时钟周期是否要进入中断/异常处理状态。

在CP0中，我定义了state\_register, cause\_register, epc\_register, prid\_register这四个寄存器，并通过组合逻辑根据当前E级指令信号（我的CP0模块位于CPU的E级）以及CPU当前的state来判断是否要响应异常中断或是eret指令从异常处理返回用户态。同时在下一个时钟上升沿改变CP0中对应寄存器中的数值，以记录当前状态。

在实现过程中，CP0还要接收当前指令是否为延迟槽指令（存储EPC时需要存PC-4）并传出是否响应了外部中断信号（在下一个下降沿要确保向7f20写入数据从而终止外部中断）。我根据M级指令类型，判断E级指令是否为延迟槽指令：E级指令为延迟槽指令当且仅当M级指令为跳转类指令；是否响应了外部中断信号通过HWInt[4]、IM[4]和IE通过组合逻辑给出。在CP0得到已响应外部中断信号后的时钟上升沿，将一个reg类型的中间变量置1，目的是保存当前已经响应外部中断，确保在接下来的时钟下降沿时中间变量仍然可以处于高电平。当中间变量处于高电平时，我们向0x7f20写入值，这样我们就可以确保在响应外部中断后的时钟下降沿终止外部中断信号，外部中断信号只能存在一个时钟周期。

### （五）Bridge和IO设计

Bridge在我的CPU中的功能仅具有多路分配器的功能，根据当前M级的指令地址判断CPU当前的操作对象是DM，Timer0，Timer1中的哪一个，以及CPU将从三者中哪一个部件读入数据。

IO端口是在mips\_tb文件中的，可以在其中设定中断次数以及外部中断对应的宏观PC。

reg [31:0] need\_interrupt = 100;

always @(negedge clk) begin

if (~reset) begin

if (need\_interrupt == 32'd100 && fixed\_macroscopic\_pc == 32'h3010) begin # 实现在0x3010处中断

interrupt <= 1;

need\_interrupt <= need\_interrupt - 1;

end

# 此处可添加中断信号

end

end

### （六）重要机制实现方法

#### 1. 跳转

NPC模块利用D级指令地址，计算D级指令的下一条指令所有可能的跳转地址（不区分D级指令的类型），得到pc\_D\_B, pc\_D\_JJal, pc\_D\_Jr, 并将所有计算出的、可能的地址传输到PC中。PC模块计算F级指令地址+4的结果，并接收由controller传递的下一条指令运算方法信号（npcop），利用mux元件对得到的所有可能的地址进行选择，得到正确的地址，储存到PC模块的寄存器中。

#### 2. 流水线延迟槽

流水线延迟槽即当前指令为分支类指令时，不论是否发生跳转，CPU都执行下一条指令，这样我们就能够通过编译调度，将适当的指令移到延迟槽中，从而充分发挥流水线CPU的性能。具体到CPU设计实现的过程，我们只需要在D级进行完跳转指令的判断后的下一个时钟周期，无论结果如何、是否跳转，都不清除D级流水线寄存器中的指令，这样分支指令的下一条指令无论如何都将被执行，也就是达到了延迟槽的功能。

#### 3. 转发

转发是要解决当一条指令的执行需要某个寄存器中的数值，但是该指令之前的指令已经改变了该寄存器的值且改变后的值还没有写入到寄存器中的情况。为了得到当前寄存器中的正确的值，我们需要将已经得到的、改变的寄存器编号与本条指令当前需要的寄存器编号相同的、存储于后续流水级的数据转发到之前的流水级中。

转发的本质是RF涉及到流水线寄存器的两个级（D级和W级），造成的RF读和写阶段的分离使得我们需要通过转发来解决数据冲突的问题。通过分析需要用到RF数据的流水级（D级、E级和M级），以及能够保存新的RF寄存器中数据的流水级（M级和W级）不难得知转发的位点有且只有三处：

（1） D级比较器的两个输入端（此处转发的结果也作为D级RF寄存器中读取的数据传递到E级）

转发数据来源：

a. M级PC+8（pc8\_M）

b. M级ALU的结果（ALUres\_M）

c. RF寄存器读出的结果（注：CPU设计时RF采用了内部转发，此处无需转发W级回写数据）

（2） E级ALU的两个输入端

转发数据来源：

a. M级PC+8（pc8\_M）

b. M级ALU的结果（ALUres\_M）

c. W级回写寄存器的数据（res\_W）

d 沿流水线寄存器传递到E级ALU输入端的数据（rfrd\_E）

（3） M级DM的数据输入端

转发数据来源：

a. W级回写寄存器的数据（res\_W）

b 沿流水线寄存器传递到M级DM数据输入端的数据（wddm\_M）

CPU设计时，在RiskSolveUnit模块产生上述转发的多路选择器选择信号，大致的思想是“有转发数据则用转发数据，多个转发数据采用新的转发数据”，这里的新的指的就是出现时间较晚的指令，也就是靠前的流水级。在RiskSolveUnit模块，转发选择信号的生成规则是：只要**当前位点的读取寄存器地址和某转发输入来源的写入寄存器地址相等且不为0**，而转发信号的优先级可以通过对if-else指令排列的顺序来确定，对优先级高的信号优先判断即可。例如E级ALU的A输入端的转发数据选择信号的生成方法为：

if(rfwe\_M == 1 && read1addr\_E == writeaddr\_M && read1addr\_E == 5'd31 && command\_M[31:26] == `jal)begin

forward\_ALUa\_E = 2'd3;

end

else if(rfwe\_M == 1 && read1addr\_E == writeaddr\_M && read1addr\_E != 0) begin

forward\_ALUa\_E = 2'd2;

end

else if(rfwe\_W == 1 && read1addr\_E == writeaddr\_W && read1addr\_E != 0) begin

forward\_ALUa\_E = 2'd1;

end

else begin

forward\_ALUa\_E = 2'd0;

end

这样我们就能够在RiskSolveUnit模块中得到正确的选择信号，将其输入到各个转发部件的选择信号端口，即可转发恰当的数据。

#### 4. 暂停

阻塞是要解决当一条指令的执行需要某个寄存器中的数值，但是该指令之前的指令即将改变该寄存器的值且得到改变数值的时间要大于需求该寄存器数值的时间，也就是说之前的指令无法在该指令需要该寄存器值之前得到更新该寄存器的值，这时候我们就不得不阻塞流水线CPU的运行，直到之前的指令能够在该指令需求前得到更新后的寄存器数值。

阻塞的实现采用的是教程中的供给时间模型。

对于某一个指令的某一个数据需求，我们定义需求时间Tuse为：这条指令位于D级的时候，再经过多少个时钟周期就必须要使用相应的数据。

对于某个指令的数据产出，我们定义供给时间Tnew为：位于某个流水级的某个指令，它经过多少个时钟周期可以算出结果并且存储到流水级寄存器里。

在设计过程中，我在controller模块进行指令译码的同时，赋予指令Tnew\_D、Tuse\_Drs以及Tuse\_Drt的具体数值，其中Tnew为动态数值，在不同流水级中的数值不同，需要随流水线向下传递，其计算方法为Tnew\_n = max(Tnew\_n-1,0) n为流水线级数。通过分析得知，我们只需要在指令刚刚进入流水线（处于D级）时做出判断：通过比对D级指令的Tuse\_Drs、Tuse\_Drt和E、M级指令的Tnew，当Tnew>Tuse需要暂停流水线，其余情况均可通过转发避免数据冲突。（注： 对于不产生新数据的指令，将Tnew设置成0；对于不需要使用数据的指令，将Tuse设置成5，这样能够确保不产生新数据或不需要新数据的指令不会引发暂停）。当我们得知需要对流水线做暂停操作时，即在RiskSolveUnit输出PC寄存器的冻结信号，D级流水线寄存器的冻结信号以及E级流水线的清除信号，这样就相当于我们在D级指令之前插入了一个nop指令。如此，就完成了暂停机制的构建。

#### 5. 异常

由于所有异常指令都可以在E级判断完毕，我在E级中加入了errorjudge模块，用于判断指令的异常行为，并生成异常指令对应的异常码传递至CP0模块。在CP0模块中，如果接收到异常码且当前未处于中断或异常处理状态，则会进入异常处理状态，产生异常信号（命名为jumpto4180），该信号具有的功能包括清除DEM级流水线寄存器以及更改F级当前PC至0x4080。同时在CP0模块中也会将EXL设为1，用于标记当前已进入异常处理状态，并记录中断产生的指令地址以及是否为延迟槽指令等信息。

#### 6. 中断

中断处理则是将外部中断信号以及两个计时器的中断信号传递至CP0模块中，CP0模块会根据CPU当前转态以及是否中断使能来确定是否响应此中断，若响应此中断，则会向cause寄存器中写入当前六个中断位的中断状态，其余操作与异常处理类似。

在处理异常和中断时值得注意的是，由于我的宏观PC选取是E级PC，当中断或者异常发生，清除DE流水线寄存器时需要将指令地址更改为中断或异常跳转的目标地址。即如果响应中断或异常，则在清除DE流水线寄存器时将指令地址更改为0x4180；如果eret返回用户态，则在清除DE流水线寄存器时将指令地址更改为pc\_eret。这样我们才能确保E级的macro\_pc始终为CPU处理完毕的下一条指令地址，不至于E级macro\_pc出现为0的状态。

## 二、测试方案

### （一）典型测试样例

#### 1. 顺序执行测试

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

subu $a3,$v1,$a1 # 负负

slt $t0,$a2,$a3 # 正负

sltu $t1,$a3,$a2 # 负正

and $t2,$a2,$at # 正正

or $t3,$a1,$a3 # 负负

nor $t4,$a2,$a3 # 正负

xor $t5,$a3,$t4 # 负正

sllv $t6,$a1,$a2

srlv $t7,$a2,$a3

srav $s0,$t1,$s2

mult $t1,$t2

mfhi $t0

mflo $t1

mthi $s0

divu $t1,$t3

mflo $t5

ori $t6,0

ori $t7,1

nop

sw $t2,0($t6)

sh $t3,6($t6)

sb $t4,7($t6)

sw $t5,4($t6)

ori $k0,0

ori $k1,1

nop

lw $s0,12($t6)

lh $s1,10($t6)

lb $s2,4($t6)

lw $s3,0($t6)

ori $v1,4

ori $k0,4

subu $k1,$k1,$k1

#### 2. 转发测试

# ALU+ALU

lui $t0,0xffff

ori $t0,$t0,0xffff

lui $t1,10

ori $t1,$t1,100

mthi $t1

# ALU（包括乘除类）+save类

addu $t3,$t1,$t0

sllv $t4,$t0,$t3

mult $t3,$t4

mflo $t5

sw $t5,0($0)

# ALU（包括乘除类）+save类

ori $t6,$t6,100

sw $6,4($0)

lui $t7,100

sh $t7,8($0)

Jal or jalr+save类

jal f1

sb $ra,12($0)

Jal or jalr+ALU（包括乘除类）

f1:

jal f2

srav $s0,$t7,$ra

Jal or jalr+save类

f2:

jal f3

sw $ra,24($0)

addu $k0,$ra,5

f3:

sw $s2,16($0)

ori $s3,100

ori $s4,100

bne $s3,$s4,f4

ori $s5,$s5,235

ori $s6,$s6,794

f4:

sw $s6,20($0)

#### 3. 暂停测试

# ALU（包括乘除类）+B类

ori $t0,$t0,4

ori $t1,$t1,4

bne $t0,$t1,f1

nop

ori $s0,10

mflo $t0

bgtz $t0,f2

nop

f2:

# Jal+B类

f1:

ori $s0,100

jal f2

ori $t2,$s0,320

f2:

beq $ra,$t2,f3

nop

f3:

sw $t0,0($0)

sw $t1,4($0)

sw $s0,8($0)

sw $t2,12($0)

# load类+B类（暂停两拍）

lw $t3,0($0)

lw $t4,4($0)

beq $t3,$t4,f4

nop

# load类+ ALU（包括乘除类）

f4:

lw $s5,8($0)

lw $s6,12($0)

addu $k0,$s5,$s6

lbu $t0,3($0)

mthi $t0

# load类+save类

lw $s5,0($0)

sw $s6,0($s5)

# load+save（不暂停）

lw $t0,4($0)

lw $t1,8($0)

sw $t1,0($t0)

# 乘除法+乘除法

mult $t0,$t1

mflo $t2

mult $s0,$s1

divu $t1,$t2

# 针对P7新增指令

mtc0 $t0,$14

eret

#### 4. 跳转及延迟槽测试

ori $t0,$t0,4

ori $t1,$t1,5

ori $t2,$t2,4

# B类跳转

bne $t0,$t1,f1

subu $s0,$t0,$t2

sllv $s1,$t1,$t3

f1:

subu $t4,$t1,$t0

# jal or jalr跳转

jal f2

lui $t5,100

ori $t5,300

la $s1,f3

jalr $31,$s1

nop

nop

f3:

# B类跳转

beqz $t0,end

f2:

addu $t6,$ra,$t2

ori $k0,$k0,10

ori $k1,$k1,20

# B类跳转

beq $k0,$k1,f4

addu $t6,$t5,$k0

addu $k0,$k0,$k0

bgtz $k0,f5

nop

# j跳转

j f3

subu $t7,$t6,$k1

ori $a0,$a0,10

f4:

# jr跳转

jr $ra

sw $ra,0($0)

lw $ra,4($0)

end:

sw $t0,0($0)

#### 4. 中断和异常测试

构造教程中提到的异常类型

AdEL

j 0x3002

j 0x7f20

ori $t0,$t0,1

lw $t1,0($t0)

lbu $t2,0($t0)

lui $t1,0x7fff

lh $t2,0x7fff($t1)

AdES

sw $t0,2($0)

sh $t0,1($0)

sb $t0,0x7f03($0)

lui $t0,0x7fff

sh $t2,0x7fff($t0)

sw $t2,0x7f08($0)

Ov

lui $t0,0x7fff

lui $t1,0x7fff

lui $t2,0x8000

add $t3,$t0,$t1

addi $t4,$t0,0x7fff

sub $t5,$t2,$t0

中断信号可以通过外部中断和Timer计时器产生，在测试过程中可以调节handler中对两个timer设置的初始值和更改mips\_tb中的外部中断产生对应的指令来改变中断信号产生的时间。在这里我构造了中断异常同时产生，eret返回用户态紧跟异常或中断，连续中断以及连续异常等边界条件，进行中断和异常的测试。

例如连续外部中断：

if (need\_interrupt == 32'd3 && fixed\_macroscopic\_pc == 32'h3010) begin

interrupt <= 1;

need\_interrupt <= need\_interrupt - 1;

end

if (need\_interrupt == 32'd2 && fixed\_macroscopic\_pc == 32'h3014) begin

interrupt <= 1;

need\_interrupt <= need\_interrupt - 1;

end

if (need\_interrupt == 32'd1 && fixed\_macroscopic\_pc == 32'h3018) begin

interrupt <= 1;

need\_interrupt <= need\_interrupt - 1;

end

在进行P7测试时，我发现的值得注意的点包括：

1. 对于指令序列mtc0 eret，eret需要阻塞一个周期（理论上转发也可）

2. 由于我的宏观PC选择的是E级PC，所以在响应中断和异常以及从异常返回用户态清除流水线寄存器时，都要注意将流水线寄存器中的地址更改为恰当的数值

3. 要注意当mtc0指令和乘除运算指令产生异常或中断时，不应改变E级CP0中寄存器以及register\_hi和register\_lo寄存器中保存的数值。

### （二）自动测试工具（python环境）

#### 1. 测试样例生成器

# 测试样例生成器，包含P7的全部指令

import random

data1 = [] # 记录顺序执行的代码

data2 = [] # 记录跳转代码 # beq bne blez bgtz bltz bgez j jal jalr jr

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

# 九个大指令类

calmudv = ['mult', 'multu', 'div', 'divu']

readmudv = ['mflo', 'mfhi']

setmudv = ['mtlo', 'mthi']

caldoubleregister = ['add', 'addu', 'sub', 'subu', 'slt', 'sltu', 'and', 'or', 'nor', 'xor', 'sllv', 'srlv', 'srav']

shift = ['sll', 'srl', 'sra']

calregisterimm = ['addi', 'addiu', 'andi', 'ori', 'xori']

saveload = ['sw', 'sh', 'sb', 'lw', 'lh', 'lhu', 'lb', 'lbu']

shifti = ['slti', 'sltiu']

lui = ['lui']

aboutc0 = ['mfc0', 'mtc0']

# 跳转类

btypetwo = ['beq', 'bne']

btypeone = ['blez', 'bgtz', 'bltz', 'bgez']

jforward = ['jal', 'jalr']

jback = ['jr']

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):

print("lui ${},{}".format(i, random.randint(0, 65535)))

print("ori ${},${},{}".format(i, i, random.randint(0, 65535)))

for i in range(30):

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

def print\_instr(x):

if data1[x].type == 1:

print("{} ${},${}".format(random.choice(calmudv), data1[x].rs, data1[x].rt, data1[x].rd))

elif data1[x].type == 2:

print("{} ${}".format(random.choice(readmudv), data1[i].rs))

elif data1[x].type == 3:

print("{} ${}".format(random.choice(setmudv), data1[i].rs))

elif data1[x].type == 4:

print("{} ${},${},${}".format(random.choice(caldoubleregister), data1[i].rs, data1[i].rt, data1[i].rd))

elif data1[x].type == 5:

print("{} ${},${},{}".format(random.choice(shift), data1[i].rs, data1[i].rt, random.randint(0, 31)))

elif data1[x].type == 6:

print(

"{} ${},${},{}".format(random.choice(calregisterimm), data1[i].rs, data1[i].rt, random.randint(0, 65535)))

elif data1[x].type == 7:

print("{} ${},{}($0)".format(random.choice(saveload), data1[i].rs, random.randrange(0, 8192, 4)))

elif data1[x].type == 8:

print("{} ${},${},{}".format(random.choice(shifti), data1[i].rs, data1[i].rt, random.randint(0, 31)))

elif data1[x].type == 9:

print("{} ${},{}".format(random.choice(lui), data1[i].rt, random.randint(0, 65535)))

elif data1[x].type == 10:

print("{} ${}, ${}".format(random.choice(aboutc0), data1[i].rs, random.randint(12, 14)))

else:

print("nop")

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

pre\_fill()

total\_blocks = random.randint(1, 25) # 代码块数

total\_lines = 0 # 代码行数

cnt = 0

# 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, 3), random.randint(1, 25)))

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, 30))

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, 11)))

data1[i].rs = random.randint(2, 31)

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

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

for i in range(total\_blocks):

coin = random.randint(0, 3)

if coin == 0:

data2[i].rs = data2[i].rt = random.randint(0, 31)

else:

data2[i].rs = random.randint(0, 31)

data2[i].rt = random.randint(0, 31)

print("ori $29,$0,0x00002F00")

print("jal funct")

print("ori $3,$3,100") # 测试延迟槽

print("ori $4,$4,200")

print("funct:") # 初始化31号寄存器

# 输出过程

k = 0

for i in range(total\_blocks):

print("interval{}:".format(cnt))

cnt += 1

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

print\_instr(k)

k += 1

if data2[i].type == 0: # 输出data2[i]

print("{} ${},${},branch{}".format(random.choice(btypetwo), data2[i].rs, data2[i].rt, i))

print("ori $3,$3,100") # 测试延迟槽

print("ori $4,$4,200")

print("interval{}:".format(cnt))

cnt += 1

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

print\_instr(k)

k += 1

print("branch{}:".format(i))

elif data2[i].type == 1:

print("{} ${} branch{}".format(random.choice(btypeone), data2[i].rs, i))

print("ori $3,$3,100") # 测试延迟槽

print("ori $4,$4,200")

print("interval{}:".format(cnt))

cnt += 1

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

print\_instr(k)

k += 1

print("branch{}:".format(i))

elif data2[i].type == 2:

print("j branch{}".format(i))

print("ori $3,$3,100") # 测试延迟槽

print("ori $4,$4,200")

print("interval{}:".format(cnt))

cnt += 1

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

print\_instr(k)

k += 1

print("branch{}:".format(i))

elif data2[i].type == 3:

coin = random.randint(0, 1)

if coin == 0: # jal

print("jal funct{}".format(i))

# else:

# print("jalr ${}, ${}")

print("ori $3,$3,100") # 测试延迟槽

print("ori $4,$4,200")

print("interval{}:".format(cnt))

cnt += 1

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

print\_instr(k)

k += 1

print("beq $0,$0,interval{}".format(cnt))

print("ori $3,$3,100") # 测试延迟槽

print("ori $4,$4,200")

print("funct{}:".format(i))

print("jr $ra")

print("ori $3,$3,100") # 测试延迟槽

print("ori $4,$4,200")

print("interval{}:".format(cnt))

for j in range(interval[-1]): # 输出interval

print\_instr(k)

k += 1

# print("end:")

# print("beq $0, $0, end")

# print("nop")

#### 2. 自动执行脚本

import os

import re

# 生成仿真需要的两个文件

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

tclFile.write("run 10us;\nexit")

prjFile1 = open("testmine.prj", "w")

for root, dirs, files in os.walk(r"E:\1-Project\P7\testprogram\modulemine"):

for fileName in files:

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

prjFile1.write("Verilog work " + root + "\\" + fileName + "\n")

prjFile2 = open("testzqy.prj", "w")

for root, dirs, files in os.walk(r"E:\1-Project\P7\testprogram\modulezqy"):

for fileName in files:

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

prjFile2.write("Verilog work " + root + "\\" + fileName + "\n")

tclFile.close()

prjFile1.close()

prjFile2.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 testmine.prj -o testmipsmine.exe mips\_txt >CompileLogmine.txt")

os.system("testmipsmine.exe -nolog -tclbatch test.tcl >" + num + "-myoutput.txt")

os.system(

r"D:\ISESetup\14.7\ISE\_DS\ISE\bin\nt64\fuse --nodebug --prj testzqy.prj -o testmipszqy.exe mips\_txt >CompileLogzqy.txt")

os.system("testmipszqy.exe -nolog -tclbatch test.tcl >" + num + "-zqyoutput.txt")

os.system(r"python compare.py " + num) # 调用compare.py比对输出

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

for i in range(20): # 可设定执行次数

gen(str(i))

#### 3. 正确性判定脚本

import os

import re

import sys

def cmp(num):

# 读取三个txt文件

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

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

text1 = file1.read()

text2 = file2.read()

file1.close()

file2.close()

# 由于不同程序的结果输出数据的时间可能有差异，且Mars输出的结果不带有时间，所以采用正则表达式匹配，并构造(指令地址，写入对象，写入数据三元组)

obj = re.compile(r".\*?@(?P<pc>.\*?): (?P<place>.\*?) <= (?P<number>.\*?)\n", re.S)

result1 = obj.finditer(text1)

result2 = obj.finditer(text2)

operation1 = []

operation2 = []

for it in result1:

operation1.append((it.group('pc'), it.group('place'), it.group('number')))

for it in result2:

operation2.append((it.group('pc'), it.group('place'), it.group('number')))

res = open(num + '-diff.txt', 'w')

if not (len(operation1) == len(operation2)):

flag = False

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

res.write("len(res1) = {}\n".format(len(operation1)))

res.write("len(res2) = {}\n".format(len(operation2)))

for i in range(min(len(operation1), len(operation2))):

if not (operation1[i] == operation2[i]):

flag = False

res.write("diff in len({})\n".format(i + 1))

res.write("res1 = {}\n".format(operation1[i]))

res.write("res2 = {}\n".format(operation2[i]))

res.close()

if flag:

print("完全正确")

os.remove(num + '-diff.txt')

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

for i in range(1, len(sys.argv)):

strs = sys.argv[i]

cmp(strs)

## 三、思考题

### （一）

我们计组课程一本参考书目标题中有“硬件/软件接口”接口字样，那么到底什么是“硬件/软件接口”？（Tips：什么是接口？和我们到现在为止所学的有什么联系？）

硬件接口指的是两个硬件设备之间的连接方式，硬件之间可以通过接口传递信息。硬件接口既包括物理上的接口，还包括逻辑上的数据传送协议。软件接口指程序中具体负责在不同模块之间传输或接受数据的并做处理的类或者函数，能够赋予开发人员无需理解软件程序源码就可以了解和访问软件内部功能和工作机制的能力，如各种软件API和人与软件的交互界面等等。

在我们目前设计的CPU中，不同计算机硬件之间通过硬件接口相互连接。包括CPU与IM的接口、CPU与系统桥的接口等等。硬件接口允许两个硬件之间进行数据的传输和沟通，能够将所有硬件的功能协调和整合起来。同时，两个相互连接的接口应该是高度匹配的，需要我们在构建时加以注意。

### （二）

BE部件对所有的外设都是必要的吗？

Bridge的功能类似于多路分配器的功能，是通过分析CPU传入的地址信号，将得到的数据传输到恰当的CPU外设，我们设计BE部件的根本原因是DM、Timer1和Timer2共用一个地址信号，需要用BE模块区分操作对象。但是对于CPU产生的仅涉及单一外设的沟通信号时，BE信号不是必要的，我们可以直接将CPU和单一外设相连，就如我们设计的CPU可以直接与IM相连。因此BE部件不是必要的。

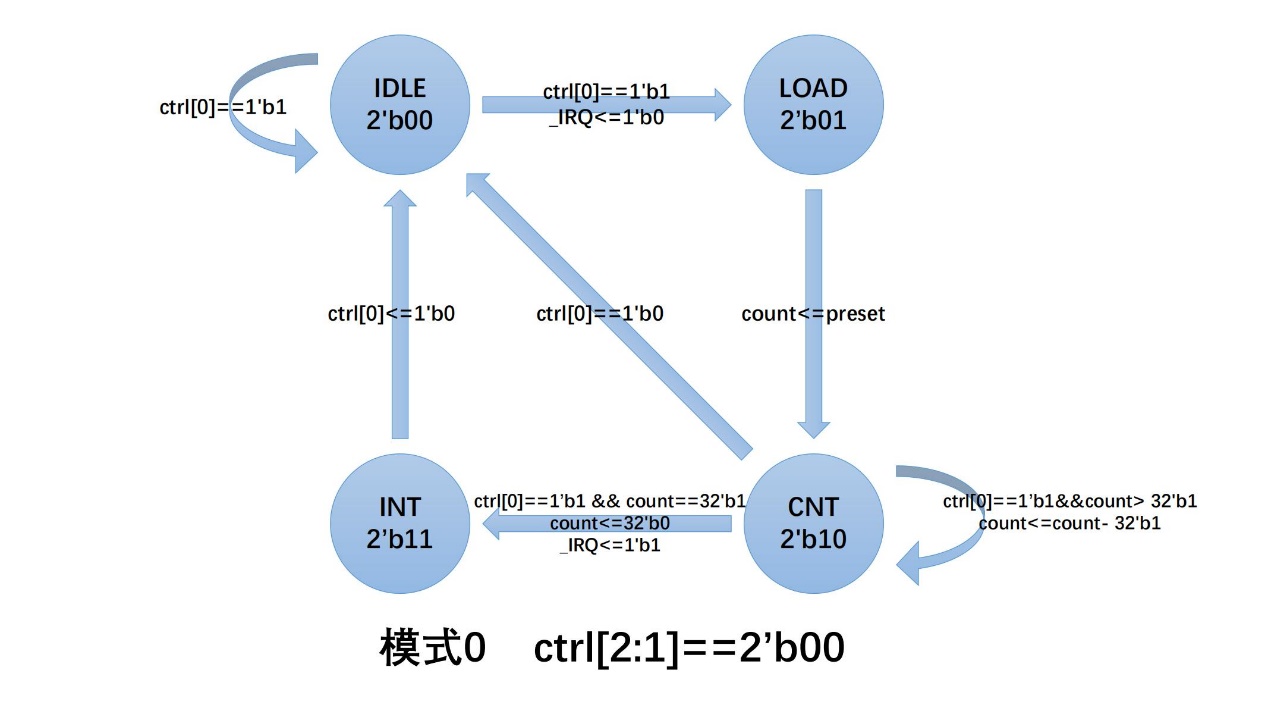
### （三）

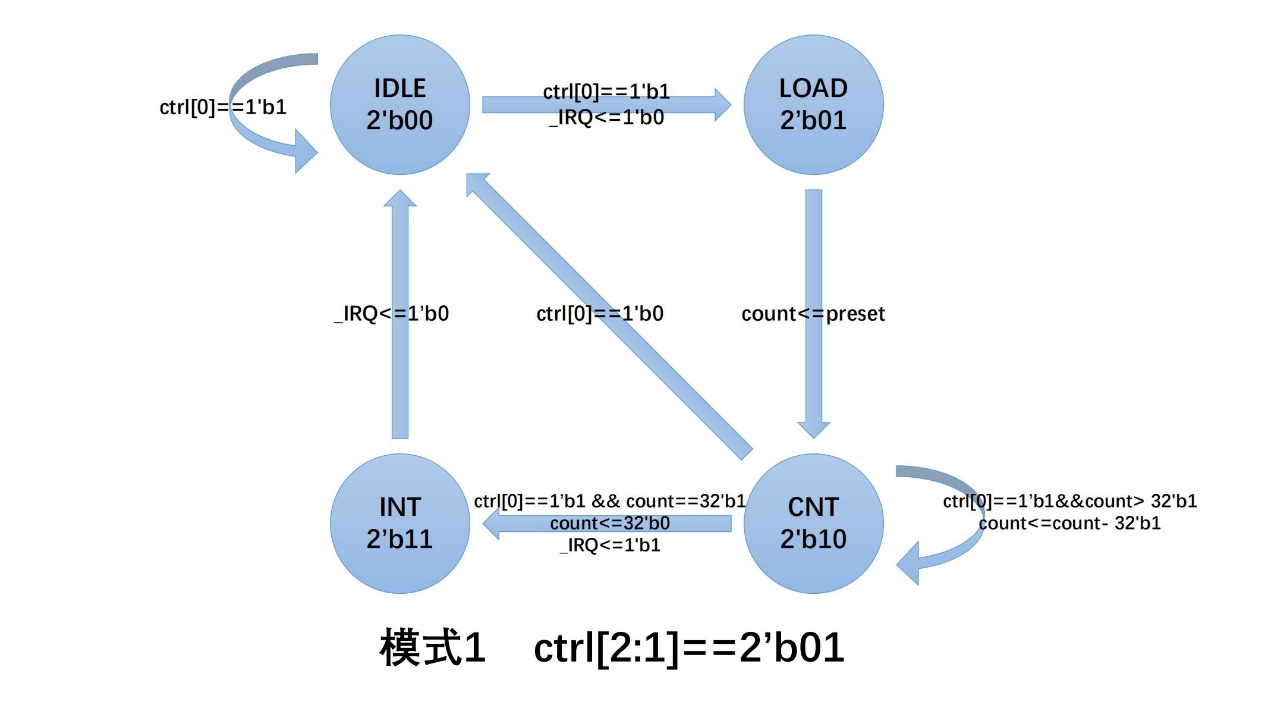
请阅读官方提供的定时器源代码，阐述两种中断模式的异同，并分别针对每一种模式绘制状态移图。

两种中断模式的相同点是都能够以初值寄存器的32位数值为初始值开始计数，每个时钟周期减一，当计数器为0时产生中断信号。

两种中断模式的不同点是计数器倒计数为0时，两个处理模式的处理方法不同。在模式0的条件下，计数器倒计数为0后，控制寄存器的使能端将自动变为0，当使能端变为1后，初值寄存器值才再次被加载至计数器，计数器重新启动倒计数，模式0通常用于产生定时中断；在模式1的条件下，当计数器倒计数为0后，初值寄存器值被自动加载至计数器，计数器继续倒计数。模式1通常用于产生周期性脉冲。不同于模式0，模式1下计数器每次计数循环中只产生一周期的中断信号

状态转移图：





### （四）

请开发一个主程序以及定时器的 exception handler。整个系统完成如下功能：

（1）定时器在主程序中被初始化为模式0；

（2）定时器倒计数至0产生中断；

（3）handler设置使能 Enable 为 1 从而再次启动定时器的计数器。(2) 及 (3) 被无限重复。

（4）主程序在初始化时将定时器初始化为模式0，设定初值寄存器的初值为某个值，如100或1000。（注意，主程序可能需要涉及对CP0.SR的编程，推荐阅读过后文后再进行。）

.ktext 0x4180

ori $t2, $0, 9

sw $t2, 0x7f00($0)

sw $t2, 0x7f10($0) # 重新设定ctrl

mfc0 $t0, $14

addi $t0, $t0, 4

mtc0 $t0, $14 # 对EPC加4

eret

.text

ori $t0, $0, 1000

sw $t0, 0x7f04($0) # 初始化preset in timer0为1000

ori $t1, $0, 100

sw $t1, 0x7f14($0) # 初始化preset in timer1为100

ori $t2, $0, 9

sw $t2, 0x7f00($0)

sw $t2, 0x7f10($0) # 设定ctrl为32'b1001 允许产生中断\_模式0\_计数器使能

### （五）

请查阅相关资料，说明鼠标和键盘的输入信号是如何被CPU知晓的？

键盘、鼠标这类的低速设备是通过中断请求的方式进行IO操作的。即当键盘上按下一个按键的时候，键盘会发出一个中断信号，中断信号经过中断控制器传到CPU，然后CPU根据不同的中断号执行不同的中断响应程序，然后进行相应的IO操作，把按下的按键编码读到寄存器（或者鼠标的操作），最后放入内存中。