Verilog单周期CPU设计文档

一、 CPU设计方案综述

（一） 总体设计概述

使用Verilog开发一个简单的单周期CPU，总体概述如下：

1. 此CPU为32位CPU
2. 此CPU为单周期设计
3. 此CPU支持的指令集为：

   {addu, subu, ori, lw, sw, beq, lui, nop,andi,jal,j,jr,sll,add,sub}
4. nop机器码为0x00000000
5. addu, subu不支持溢出

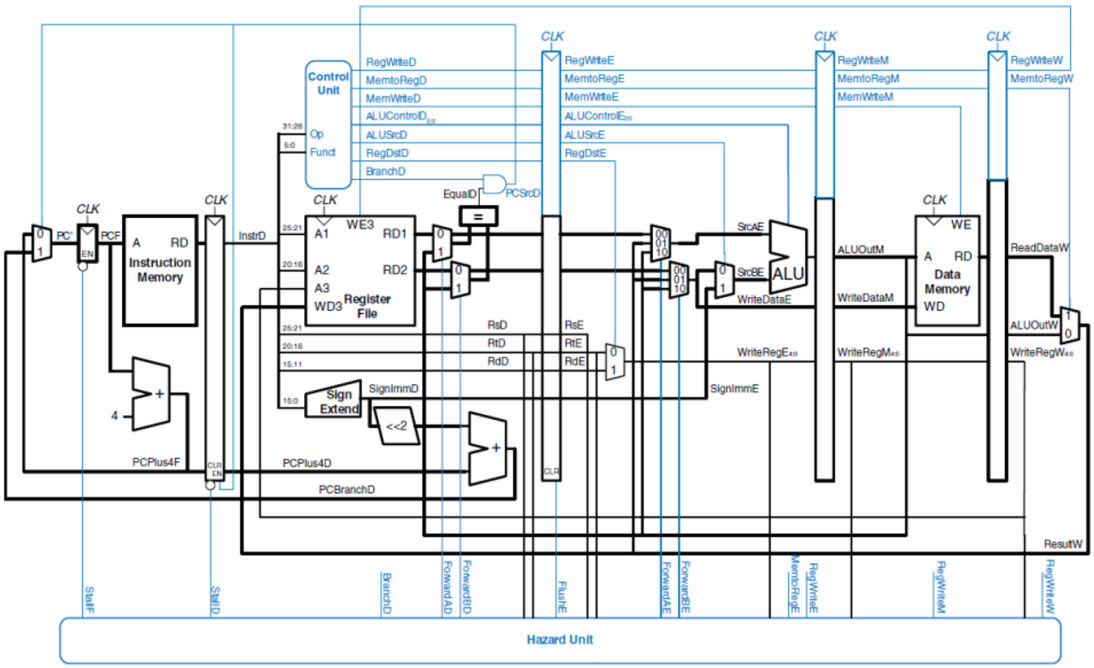

Figure 7.58 Pipelined processor with full hazard handling

（二） 关键模块定义

1. IFU

（1） 端口说明

表1-IFU端口说明

序号	信号名	方向	描述
1	clk	I	时钟信号
2	reset	I	异步复位信号，将PC值置为0x000000000：无效1：复位
3	branch[3:0]	I	判断转移类型 4'b0001:PC+4 4'b0010:beq 4'b0100:26位扩展 4'b1000：jr指令指定寄存器
4	eq	I	A1、A2对应GRF两个寄存器中的值是否相等（是否满足beq的跳转要求） 0：不相等 1：相等
5	imm32[31:0]	I	32位的扩展后的立即数（16位或26位）
6	instr[31:0]	O	将IM中，要执行的指令输出
7	PC4[31:0]	O	满足jal指令中PC+4的存储
8	PC[31:0]	O	当前执行的PC

（2） 功能定义

表2-IFU功能定义

序号	功能	描述
1	复位	reset有效时，PC置为0x00000000
2	更新PC的值	branch为4'b0001：PC = PC + 4 branch为4'b0010： eq为0: PC = PC + 4 +sign_extend(offset||02) eq为1：PC = PC + 4 branch4'b0100: PC=PC31..28||instr_index||02 branch为4'b1000:PC = GPR[rs]
3	输出指令	根据PC的值，取出IM中的指令

```
`timescale 1ns / 1ps
`define PC_Initial 32'h0000_3000
module IFU(
    input [3:0] branch,
    input eq,
    input [31:0] imm32,
     input clk,
     input reset,

     output [31:0] Instr,
    output [31:0] PC4,
     output [31:0] PC
    );
```

```
     reg [31:0] now_PC=`PC_Initial;
     reg [31:0] NPC;
     wire [9:0] Address;

    always @(posedge clk)begin
        if(reset)begin
            now_PC<=`PC_Initial;
        end
        else begin
            now_PC<=NPC;
            //$display("branch:%h,eq:%h,imm:%h",branch,eq,imm32);
        end
    end
    assign Address=now_PC[11:2];
    assign PC4=now_PC+4;
    always @(*)begin
        if(branch==4'b0001)begin
            NPC=PC4;
        end
        else if(branch==4'b0010)begin
            if(eq)begin
                NPC=PC4+{imm32[29:0],{2{1'b0}}};
            end
            else begin
                NPC=PC4;
            end
        end
        else if(branch==4'b0100)begin
            NPC={now_PC[31:28],imm32[25:0],{2{1'b0}}};
        end
        else begin
            NPC=imm32;
        end
    end
    assign PC=now_PC;
    IM imm(.Address(Address),.Instr(Instr));
endmodule

```

```
//内置IM
`timescale 1ns / 1ps
module IM(
    input [9:0] Address,
    output [31:0] Instr
    );
    reg [31:0] im[1023:0];
     integer i=0;
     initial begin
        for(i=0;i<=1023;i=i+1)begin
            im[i]=0;
        end
        $readmemh("code.txt",im);
     end
    assign Instr=im[Address];
endmodule

```

2. GRF

（1） 端口说明

表3-GRF端口说明

序号	信号名	方向	描述
1	clk	I	时钟信号
2	reset	I	异步复位信号，将32个寄存器中全部清零1：清零0：无效
3	WE	I	写使能信号1：可向GRF中写入数据0：不能向GRF中写入数据
4	A1[4:0]	I	5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD1
5	A2[4:0]	I	5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD2
6	A3[4:0]	I	5位地址输入信号，指定32个寄存器中的一个，作为RD的写入地址
7	WD[31:0]	I	32位写入数据
8	WPC[31:0]	I	当前写入GRF的PC（用于测试）
9	RD1[31:0]	O	输出A1指定的寄存器的32位数据
10	RD2[31:0]	O	输出A2指定的寄存器的32位数据

（2） 功能定义

表4-GRF功能定义

序号	功能	描述
1	异步复位	reset为1时，将所有寄存器清零
2	读数据	将A1和A2地址对应的寄存器的值分别通过RD1和RD2读出
3	写数据	当WE为1且时钟上升沿来临时，将WD写入到A3对应的寄存器内部

```
`timescale 1ns / 1ps
module GRF(
    input [4:0] A1,
    input [4:0] A2,
    input [4:0] A3,
    input [31:0] WD,
    input clk,
    input reset,
    input WE,
     input [31:0] WPC,
    output [31:0] RD1,
    output [31:0] RD2
```

```
    );
        reg[31:0] RF[31:0];
        integer i=0;
        initial begin
            for(i=0;i<=31;i=i+1)begin
                RF[i]=0;
            end
        end
        always@(posedge clk)begin
            if(reset)begin
                for(i=0;i<=31;i=i+1)begin
                    RF[i]<=0;
                end
            end
            else begin
                if(WE)begin
                    RF[A3]<=WD;
                    RF[0]<=0;
                    $display("@%h: $%d <= %h", WPC, A3, WD);
                end
                else begin
                    RF[A3]<=RF[A3];
                end
            end
        end
        assign RD1=RF[A1];
        assign RD2=RF[A2];
endmodule

```

3. ALU

(1) 端口说明

表5-ALU端口说明

序号	信号名	方向	描述
1	A[31:0]	I	参与运算的第一个数
2	B[31:0]	I	参与运算的第二个数
3	ALUop[6:0]	I	决定ALU做何种操作 7'b000001：无符号加 7'b000010：无符号减 7'b000100：与 7'b001000：或 7'b010000：左移位运算
4	eq	O	A与B是否相等 0：不相等 1：相等
5	res	O	A与B做运算后的结果

(2) 功能定义

表6-ALU功能定义

序号	功能	描述
1	加运算	res = A + B
2	减运算	res = A - B
3	与运算	res = A & B
4	或运算	res = A | B
5	左移位运算	Res=A<<B

```
`timescale 1ns / 1ps
module ALU(
    input [31:0] A,
    input [31:0] B,
    input [6:0] ALUop,
    output [31:0] res,
    output eq
    );
    assign res= (ALUop==7'b0000001)?A+B:
                    (ALUop==7'b0000010)?A+~B+1:
                    (ALUop==7'b0000100)?A&B:
                    (ALUop==7'b0001000)?A|B:
                    (ALUop==7'b0010000)?A<<B:
                    32'd0;
    assign eq = (A==B);
endmodule

```

4. DM

(1) 端口说明

表7-DM端口说明

序号	信号名	方向	描述
1	clk	I	时钟信号
2	reset	I	异步复位信号 0：无效 1：内存值全部清零
3	WE	I	写使能信号 0：禁止写入 1：允许写入
4	Address[31:0]	I	读取或写入信号地址
5	WD[31:0]	I	32为写入数据
6	pc[31:0]	I	当前输入DM的PC值（用于测试）
6	RD[31:0]	O	32位读出数据

(2) 功能定义

表8-DM功能定义

序号	功能	描述
1	异步复位	当reset为1时，DM中所有数据清零
2	写入数据	当WE有效时，时钟上升沿来临时，WD中数据写入A对应的DM地址中
3	读出数据	RD永远读出A对应的DM地址中的值

```
`timescale 1ns / 1ps
module DM(
    input [31:0] Address,
    input [31:0] WD,
    input clk,
    input reset,
    input WE,
     input [31:0] pc,
    output [31:0] RD
    );
    reg [31:0]dm[1023:0];
    integer i;
    initial begin
        for(i = 0;i < 1024; i = i + 1)begin
            dm[i] = 0;
        end
    end
    always @(posedge clk)begin
        if(reset)begin
            for(i=0;i<=31;i=i+1)begin
                dm[i]<=0;
            end
        end
```

```
        else begin
            if(WE)begin
                dm[Address[11:2]]<=WD;
                $display("@%h: *%h <= %h", pc, Address, WD);
            end
            else begin
                dm[Address[11:2]]<=dm[Address[11:2]];
            end
        end
    end
    assign RD=dm[Address[11:2]];

endmodule

```

5. EXT

(1) 端口说明

表9-EXT端口说明

序号	信号名	方向	描述
1	imm16[15:0]	I	代扩展的16位信号
2	sign	I	无符号或符号扩展选择信号 0：无符号扩展 1：符号扩展
3	imm32[31:0]	O	扩展后的32位的信号

(2) 功能定义

表10-EXT功能定义

序号	功能	描述
1	无符号扩展	当sign为0时，将imm16无符号扩展输出
2	符号扩展	当sign为1时，将imm16符号扩展输出

```
`timescale 1ns / 1ps
module EXT(
    input [15:0] imm16,
    input sign,
    output reg [31:0] imm32
    );
     always @(*)begin
        if(sign==1'b0)begin
            imm32={{16{1'b0}},imm16};
        end
        else begin
            imm32={{16{imm16[15]}},imm16};

```

```
        end
    end
endmodule

```

6. Controller

(1) 端口说明

表11-Controller端口说明

序号	信号名	方向	描述
1	op[5:0]	I	instr[31:26]6位控制信号
2	func[5:0]	I	instr[5:0]6位控制信号
3	AluOp[6:0]	O	ALU的控制信号
4	WeGrf	O	GRF写使能信号 0：禁止写入 1：允许写入
5	WeDm	O	DM的写入信号 0：禁止写入 1：允许写入
6	branch[3:0]	O	PC转移位置选择信号 4'b0001:其他情况 4'b0010:beq 4'b0100:j||jal 4'b1000:jr;
7	AluSrc1[3:0]	O	参与ALU运算的第一个数 4'b0001：RD1 4'b0010：RD2
8	AluSrc2[3:0]	O	参与ALU运算的第二个数，来自GRF还是imm 4'b0001：RD2 4'b0010：imm32 4'b0100: offset
9	WhichtoReg[3:0]	O	将何种数据写入GRF? 4'b0001：ALU计算结果 4'b0010：DM读出信号 4'b0100：upperImm 4'b1000: PC+4
10	RegDst[3:0]	O	GRF写入地址选择信号 4'b0001：Rd 4'b0010：Rt 4'b0100：31号寄存器
11	SignExt	O	是否对imm16进行符号扩展 0：不进行符号扩展 1：进行符号扩展

```
`timescale 1ns / 1ps
module Controller(
    input [5:0] op,
    input [5:0] func,
    output [6:0] ALUop,
    output Wegrf,
    output WeDm,
    output [3:0] branch,
    output [3:0] AluSrc1,
    output [3:0] AluSrc2,
```

```
    output [3:0] WhichtoReg,
    output [3:0] RegDst,
    output SignExt
    );
//指令定义:
     wire addu,subu,ori,lw,sw,lui,beq,andi,jal,j,jr,sll,add,sub;
    //R型
    assign addu=(op==6'b000000)&&(func==6'b100001);
    assign subu=(op==6'b000000)&&(func==6'b100011);
    assign add=(op==6'b000000)&&(func==6'b100000);
    assign sub=(op==6'b000000)&&(func==6'b100010);
    assign sll=(op==6'b000000)&&(func==6'b000000);
    //I型
    assign ori=op==6'b001101;
    assign lw=op==6'b100011;
    assign sw=op==6'b101011;
    assign lui=op==6'b001111;
    assign beq=op==6'b000100;
    assign andi=op==6'b001100;
    //J型
    assign jal=op==6'b000011;
    assign j=op==6'b000010;
    assign jr=(op==6'b000000)&&(func==6'b001000);
//输出定义:
    wire ALUop_6,ALUop_5,ALUop_4,ALUop_3,ALUop_2,ALUop_1,ALUop_0;
    assign
ALUop_0=~ALUop_6&&~ALUop_5&&~ALUop_4&&~ALUop_3&&~ALUop_2&&~ALUop_1;
    assign ALUop_1=subu||sub;
    assign ALUop_2=andi;
    assign ALUop_3=ori;
    assign ALUop_4=sll;
    assign ALUop_5=1'b0;
    assign ALUop_6=1'b0;
    ////////////////////////////////////////
    wire branch_3,branch_2,branch_1,branch_0;
    assign branch_0=~branch_3&&~branch_2&&~branch_1;
    assign branch_1=beq;
    assign branch_2=j||jal;
    assign branch_3=jr;
    ////////////////////////////////////////
    wire AluSrc1_3,AluSrc1_2,AluSrc1_1,AluSrc1_0;
    assign AluSrc1_0=~AluSrc1_3&&~AluSrc1_2&&~AluSrc1_1;
    assign AluSrc1_1=sll;
    assign AluSrc1_2=1'b0;
    assign AluSrc1_3=1'b0;
    ////////////////////////////////////////
    wire AluSrc2_3,AluSrc2_2,AluSrc2_1,AluSrc2_0;
    assign AluSrc2_0=~AluSrc2_3&&~AluSrc2_2&&~AluSrc2_1;
    assign AluSrc2_1=lw||sw||ori||andi;
    assign AluSrc2_2=sll;
    assign AluSrc2_3=1'b0;
    ////////////////////////////////////////
    wire WhichtoReg_3,WhichtoReg_2,WhichtoReg_1,WhichtoReg_0;
    assign WhichtoReg_0=~WhichtoReg_1&&~WhichtoReg_2&&~WhichtoReg_3;
    assign WhichtoReg_1=lw;
    assign WhichtoReg_2=lui;
    assign WhichtoReg_3=jal;
    ////////////////////////////////////////
```

```
    wire RegDst_3,RegDst_2,RegDst_1,RegDst_0;
    assign RegDst_0=~RegDst_1&&~RegDst_2&&~RegDst_3;
    assign RegDst_1=beq||lui||lw||sw||ori||andi;
    assign RegDst_2=jal;
    assign RegDst_3=1'b0;
//输出结果：
    assign ALUop={ALUop_6,ALUop_5,ALUop_4,ALUop_3,ALUop_2,ALUop_1,ALUop_0};
    assign Wegrf=sub||addu||subu||ori||lui||sll||jal||andi||lw||add;
    assign WeDm=sw;
    assign branch={branch_3,branch_2,branch_1,branch_0};
   assign AluSrc1={AluSrc1_3,AluSrc1_2,AluSrc1_1,AluSrc1_0};
    assign AluSrc2={AluSrc2_3,AluSrc2_2,AluSrc2_1,AluSrc2_0};
    assign WhichtoReg={WhichtoReg_3,WhichtoReg_2,WhichtoReg_1,WhichtoReg_0};
   assign RegDst={RegDst_3,RegDst_2,RegDst_1,RegDst_0};
    assign SignExt=lw||sw||beq||andi;

endmodule

```

真值表

端口	addu	subu	ori	lw	sw	lui	beq
op	000000	000000	001101	100011	101011	001111	000100
func	100001	100011					
AluOp	0000001	0000010	0001000	0000000	0000000	0000000	0000000
WeGrf	1	1	1	1	0	1	0
WeDm	0	0	0	0	1	0	0
branch	0001	0001	0001	0001	0001	0001	0010
AluSrc1	0001	0001	0001	0001	0001	0001	0001
AluSrc2	0001	0001	0010	0010	0010	0001	0001
WhichtoReg	0001	0001	0001	0010	0001	0100	0001
RegDst	0001	0001	0010	0010	0010	0010	1010
SignExt	0	0	0	1	1	0	1
端口	**andi**	**jal**	**j**	**jr**	**sll**	**add**	**sub**
op	001100	000011	000010	000000	000000	000000	000000
func				001000	000000	100000	100010
AluOp	0000100	0000000	0000000	0000000	0010000	0000000	0000001
WeGrf	1	1	0	0	1	1	1
WeDm	0	0	0	0	0	0	0
branch	0001	0100	0100	1000	0001	0001	0001
AluSrc1	0001	0001	0001	0001	0010	0001	0001
AluSrc2	0010	0001	0001	0001	0100	0001	0001
WhichtoReg	0001	1000	0001	0001	0001	0001	0001
RegDst	0010	0100	0001	0001	0001	0001	0001
SignExt	1	0	0	0	0	0	0

二、 测试方案

（1） 测试代码：

```
.text

ori $a0,$0,0x100

ori $a1,$a0,0x123

lui $a2,456

lui $a3,0xffff

ori $a3,$a3,0xffff

addu $s0,$a0,$a2
```

```
addu $s1,$a0,$a3

addu $s4,$a3,$a3

subu $s2,$a0,$a2

subu $s3,$a0,$a3

sw $a0,0($0)

sw $a1,4($0)

sw $a2,8($0)

sw $a3,12($0)

sw $s0,16($0)

sw $s1,20($0)

sw $s2,24($0)

sw $s3,44($0)

sw $s4,48($0)

lw $a0,0($0)

lw $a1,12($0)

sw $a0,28($0)

sw $a1,32($0)

ori $a0,$0,1

ori $a1,$0,2

ori $a2,$0,1

beq $a0,$a1,loop1

beq $a0,$a2,loop2

loop1: sw $a0,36($t0)

loop2: sw $a1,40($t0)

jal loop3

jal loop3
```

```
sw $s5,64($t0)

ori $a1,$a1,4

jal loop4

loop3:sw $a1,56($t0)

sw $ra,60($t0)

ori $s5,$s5,5

jr $ra

loop4: sw $a1,68($t0)

sw $ra,72($t0)
```

（2） MARS中运行结果

（3） 该CPU运行结果

```
@00003000: $ 4 <= 00000100
@00003004: $ 5 <= 00000123
@00003008: $ 6 <= 01c80000
@0000300c: $ 7 <= ffff0000
@00003010: $ 7 <= ffffffff
@00003014: $16 <= 01c80100
@00003018: $17 <= 000000ff
@0000301c: $20 <= fffffffe
@00003020: $18 <= fe380100
@00003024: $19 <= 00000101
@00003028: *00000000 <= 00000100
@0000302c: *00000004 <= 00000123
@00003030: *00000008 <= 01c80000
@00003034: *0000000c <= ffffffff
@00003038: *00000010 <= 01c80100
@0000303c: *00000014 <= 000000ff
@00003040: *00000018 <= fe380100
@00003044: *0000002c <= 00000101
@00003048: *00000030 <= fffffffe
@0000304c: $ 4 <= 00000100
@00003050: $ 5 <= ffffffff
@00003054: *0000001c <= 00000100
@00003058: *00000020 <= ffffffff
@0000305c: $ 4 <= 00000001
@00003060: $ 5 <= 00000002
@00003064: $ 6 <= 00000001
@00003074: *00000028 <= 00000002
```

@00003078: $31 <= 0000307c
@0000308c: *00000038 <= 00000002
@00003090: *0000003c <= 0000307c

三、 思考题

（一）阅读下面给出的 DM 的输入示例中（示例 DM 容量为 4KB，即 32bit × 1024字），根据你的理解回答，这个 addr 信号又是从哪里来的？地址信号 addr 位数为什么是 [11:2] 而不是 [9:0]？

文件	模块接口定义

dm.v	dm(clk,reset,MemWrite,addr,din,dout);
	input clk; //clock
	input reset; //reset
	input MemWrite; //memory write enable
	input [11:2] addr; //memory's address for write
	input [31:0] din; //write data
	output [31:0] dout; //read data

MIPS中以字节为单位，我们的DM中，以32位的register为单位。addr是ALU单元的输出端口接过来的，代表的是要读取的DM存储器的地址，由于DM为1024个字，所以地址为10位。

地址信号addr为[11:2]是因为mips默认按字寻址，所以最后两位默认为00，这样也有利于扩大寻址范围。

（二） 思考上述两种控制器设计的译码方式，给出代码示例，并尝试对比各方式的优劣。

指令对应的控制信号如何取值:
对于单独的指令易于debug，且思路清晰，但容易造成代码冗长，可读性差，也会造成相同功能部件重复编写，造成仿真成本较高。
控制信号每种取值所对应的指令:
这种方式在面对少量指令时显得代码较为可读性差，但在大量指令的条件下添加指令相对简单，且代码风格简便。缺点是debug时代码显得不清晰，利于添加但不利于修改。
总结而言，就我个人而言，本人更偏爱第二种控制器设计，且如果用独热码编写控制信号，也可以一定程度增强可读性，使代码更加完善。

（三）在相应的部件中，复位信号的设计都是同步复位，这与 P3 中的设计要求不同。请对比同步复位与异步复位这两种方式的 reset 信号与 clk 信号优先级的关系。

异步复位时clk和reset是相同的优先级，只要两者触发其中之一就可以实现控制信号的变化。

同步复位时clk的优先级高于reset，当clk不生效时，系统会忽略reset的取值，只有两者同时为1才可以实现复位。

（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。

addi与addiu的区别在于，当出现溢出时，addiu忽略溢出，将溢出的最高位舍弃；addi会报告SignalException(IntegerOverflow)。故在忽略溢出的大前提下，二者等价。

四.自动化测试模块

```python
import random
# 指令集
R_type = ['add', 'sub', 'addu', 'subu']
# R型指令
I_type = ['ori', 'andi', 'lui', 'lw', 'sw', 'beq']
# I型指令
J_type = ['jal', 'j']
# J型指令
filename = "D:\\coding_file\\study\\Lesson\\co_lesson\\lesson\\p4\\test.asm"
# 输出文件位置
label = [0]
# 输出label的编号范围，事先存入0防止在第一次输出标签前出现跳转指令
cnt = 0
# 可执行代码的行数
flag = 1
# 当前所标出过的编号号码
jal = []
# 使用过的jal对应标签编号
R_num = len(R_type)
I_num = len(I_type)
J_num = len(J_type)
num = R_num+I_num+J_num


class get_Code:
    def __init__(self):
        # 对应指令生成随机数
        self.rs = random.randint(0, 31)
        self.rt = random.randint(0, 31)
        self.rd = random.randint(0, 31)
        self.imm16 = random.randint(0, (1 << 16)-1)
        self.imm26 = random.randint(0, (1 << 26)-1)
        self.mem = random.randint(0, 3071)
        # 存储指令类型
        self.list = []
        # get函数
        self.get_R()
        self.get_I()
        self.get_J()
        self.get_Label()
        self.main()

    def get_R(self):
```

```
        random1 = random.randint(0, R_num - 1)
        type1 = R_type[random1]
        self.list.append(type1)

    def get_I(self):
        random2 = random.randint(0, I_num - 1)
        type2 = I_type[random2]
        self.list.append(type2)

    def get_J(self):
        random3 = random.randint(0, J_num - 1)
        type3 = J_type[random3]
        self.list.append(type3)

    def get_Label(self):
        random4 = random.randint(0, len(label)-1)
        ran = label[random4]
        return ran

    def main(self):
        sel = random.randint(0, num)
        # 控制参数类型
        if sel in range(0, R_num):
            # 通过控制随机数的范围来决定输出各种指令的频率,并用各种指令的数目保证各指令出现概率基本相同
            self.code = self.list[0] + ' ' + '$' + \
                str(self.rd) + ' ' + ',' + '$' + \
                str(self.rs) + ' ' + ',' + '$' + str(self.rt) + '\n'
        elif sel in range(R_num, R_num+I_num):
            if self.list[1] == 'lw' or self.list[1] == 'sw':
                self.code = self.list[1] + ' ' + '$' + \
                    str(self.rt) + ',' + str(self.imm16 << 2) + \
                    '(' + '$' + '0'+')'+'\n'
            elif self.list[1] == 'lui':
                self.code = self.list[1] + ' ' + \
                    '$' + str(self.rt) + ',' + str(self.imm16) + '\n'
            elif self.list[1] == 'beq':
                self.code = self.list[1] + ' ' + '$' + \
                    str(self.rt) + ',' + '$' + \
                    str(self.rs) + ',' + 'label_' + \
                    str(self.get_Label()) + '\n'
            else:
                self.code = self.list[1] + ' ' + '$' + \
                    str(self.rt) + ',' + '$' + \
                    str(self.rs) + ',' + str(self.imm16) + '\n'
        elif sel in range(R_num+I_num, num+1):
            if self.list[2] == 'jal':
                node = self.get_Label()
                self.code = self.list[2] + ' ' + 'label_' + str(node) + '\n'
                jal.append(node)
            elif self.list[2] == 'j':
                self.code = self.list[2] + ' ' + \
                    'label_' + str(self.get_Label()) + '\n'


with open(filename, 'w+') as f:
    f.write('label_0' + ':' + '\n')
```

```
    for cnt in range(0, 30):
        a = get_Code()
        f.write(a.code)
        if random.randint(0, 3) == 1 and label != []:
          # 通过控制random范围来决定标签和jr出现的概率
            f.write('label_' + str(flag) + ':' + '\n')
            label.append(flag)
            flag = flag+1
        if random.randint(0, num+1) == 1 and jal != []:
            f.write('jr $ra' + '\n')
            jal.pop(random.randint(0, len(jal) - 1))
        if random.randint(0, num+1) == 1:
            f.write('nop' + '\n')
    if jal != []:
        ran = random.randint(0, len(jal) - 1)
        f.write('jr $ra' + '\n')
        jal.pop(ran)
    f.close()
```