计算机组成原理-P5 Verilog 流水线实验报告

一、模块规格

1. IFU

取指令模块，位于 IF 级，在流水线 CPU 中内部仅包含寄存器 PC 以及指令存储器 IM，接受来自 nPC 的输入并更新 PC

端口定义表

信号名	方向	描述
clk	I	时钟信号
reset	I	重置信号
IFU_i_En	I	PC 寄存器使能，用来控制指令冲突时暂停
IFU_i_nPC[31:0]	I	跳转目标地址，针对 JR 等指令
IFU_o_Instr[31:0]	O	当前 PC 所指指令
IFU_o_PC [31:0]	O	当前 PC 的值

功能定义表

序号	功能名称	功能描述
1	重置	reset 置 1 时，PC 寄存器复位至 0x00003000
2	输出指令	Instr 为 PC 所指处指令，PC <= nPC

2. NPC

PC 值更新模块，位于 D 级，接受当前 PC 输入（直接从 IF 级连入）、控制信号输出和 Condition（条件跳转的条件），输出新的 PC 值。

端口定义表

信号名	方向	描述
NPC_i_PC[31:0]	I	当前 PC 值输入
NPC_i_Immediate[25:0]	I	输入的立即数（包括 J 系列的 26 位和 B 系列的 16 位）
NPC_i_RegAddr[31:0]	I	JR 系列指令输入的地址
NPC_i_Mode[2:0]	I	指示 nPC 如何计算，详见 Controller
NPC_i_Condition	I	条件跳转时条件的输入
NPC_o_nPC[31:0]	O	输出 nPC 值

功能：确定 PC 寄存器的下一个值

3. GRF

端口定义表

信号名	方向	描述
clk	I	时钟信号
reset	I	重置信号
GRF_i_WEnable	I	写使能信号
GRF_i_RAddr1[4:0]	I	读功能所读寄存器地址 1
GRF_i_RAddr2[4:0]	I	读功能所读寄存器地址 2
GRF_i_WAddr[4:0]	I	写功能所写寄存器地址
GRF_i_WData[31:0]	I	写功能写入数据
GRF_o_RData1[31:0]	O	读寄存器数据 1
GRF_o_RData2[31:0]	O	读寄存器数据 2

功能定义表

序号	功能名称	描述
1	重置	reset == 1 时重置 GRF，所有寄存器归零
2	读取寄存器数据	clk 接时钟信号时，GRF_o_RDdata1/2 输出寄存器编号为 GRF_i_RAddr1/2 的寄存器的值
3	写入寄存器	GRF_i_WEnable == 1 时，将 GRF_i_WData 处数据在下一个时钟上升沿存入编号为 GRF_i_WAddr 的寄存器

4. ALU

端口定义表

信号名	方向	描述
ALU_i_Operand1[31:0]	I	操作数 1
ALU_i_Operand2[31:0]	I	操作数 2（具体由 ALUOp2Datapath 元件确定）
ALU_i_Operation[3:0]	I	具体操作，0：加， 1：减， 2: 或，3:lui 操作，其他待定
ALU_o_Result[31:0]	O	运算结果
ALU_o_isOverflow	O	ADD 和 SUB 的结果是否溢出

功能定义表

序号	功能名称	描述
1	运算	根据 ALU_Operator 指定的操作进行运算，结果输出至 Result。具体操作：0：加， 1：减， 2: 或，3：lui 操作，其他待定
2	判断结果是否溢出	运算结果为 0 时，ALU_Overflow 置 1

5. **DM**

端口定义表

信号名	方向	描述
clk	I	时钟信号
reset	I	重置数据，清零 RAM
DM_i_Addr[31:0]	I	操作地址
DM_i_WData[31:0]	I	待写入数据
DM_i_WEnable	I	是否写入数据
DM_i_WMode[1:0]	I	操作位宽，0 --- word, 1 --- half, 3 --- byte
DM_o_Rdata[31:0]	O	操作地址处的数据

功能定义表

序号	功能名称	描述
1	读取	以指定位宽读取指定位置数据
2	写入	以指定位宽在指定位置写入数据

注：对于不是以字为宽度的 IO 操作，输出采用符号延拓

6. **EXT**

端口定义表

信号名	方向	描述
EXT_i_Input[15:0]	I	输入数据
EXT_i_Mode[1:0]	I	操作模式： 0 - 无符号延拓 32， 1 - 有符号延拓 32
EXT_o_Output[31:0]	I	输出数据

功能定义表

序号	功能名称	描述
1	无符号延拓 32 位	EXT_Mode == 0 时，将 EXT_Input 做 32 位无符号延拓
2	有符号延拓 32 位	EXT_Mode == 1 时，将 EXT_Input 做 32 位有符号延拓

二、控制器设计

1. 概述

该部分介绍 Controller 及各 MUX 的详细信息。控制器设计仍然分为两部分：对指令的识别以及指令被识别后控制信号的生成。两过程由一个 31 位的 instr_type 变量连接，每种指令（功能完全一致，数据可以不同）对应一个唯一的值。内体总体端口定义如下，具体信号功能请参见下文描述：

信号名	方向	描述
Instr[31:0]	I	指令
instr_type[31:0]	O	当前指令为何种指令（用于冒险控制）
NPC_Mode	O	nPC 计算方法
GRF_WEnable	O	寄存器写使能信号
ALU_Operation[3:0]	O	ALU 操作信号
DM_WEnable	O	内存写使能
DM_Mode[1:0]	O	操作位宽
EXT_Mode[1:0]	O	Extender 模式
MUX_RegWAddr_Sel[2:0]	O	寄存器写地址选择
MUX_RegWData_Sel[2:0]	O	寄存器写数据选择
MUX_ALUOp2_Sel[2:0]	O	ALU 操作数 2 选择信号

2. 相关各 MUX 控制信号说明

a) MUX_RegWAddr: 根据 MUX_RegWAddr_sel 的选择输出对应数据

端口定义表

信号名	方向	描述
MUX_RegWAddr_i_Sel[2:0]	I	选择信号：0 --- rd， 1 --- rt，2 --- $ra，其他待定
MUX_RegWAddr _i_rdIn[4:0]	I	rd 输入
MUX_RegWAddr _i_rtIn[4:0]	I	rt 输入
MUX_RegWAddr _o_Output[4:0]	O	输出

b) RegWData：根据 RegWData_Sel 的选择输出对应数据

端口定义表

信号名	方向	描述
MUX_RegWData_i_Sel[2:0]	I	选择信号：0 - alu 结果， 1 - mem 数据，2 - PC+4，其他待定
MUX_RegWData_i_ALUIn[31:0]	I	ALU 结果
MUX_RegWData_i_memIn[31:0]	I	MEM 结果
MUX_RegWData_i_linkIn[31:0]	I	PC+4 输入
MUX_RegWData_o_Output[31:0]	O	输出

c) MUX_ALUOp2: 根据 MUX_ALUOp2Data_Sel 的选择输出对应数据

端口定义表

信号名	方向	描述
MUX_ALUOp2_i_Sel[1:0]	I	选择信号，0 --- RegData2，1 --- EXTender，2,3 --- 待定
MUX_ALUOp2_i_regIn[31:0]	I	reg 数据输入
MUX_ALUOp2_i_EXTIn[31:0]	I	ext 数据输入
MUX_ALUOp2_o_Output[31:0]	O	输出

3. **指令识别**

指令识别由一个 always 块描述的组合逻辑实现。在 case(Instr[31:26])和 case(Instr[5:0])内，为 instr_type 赋对应当前指令的值，真值表如下：

功能：译码 --- 当检测到某指令时 instr_type 输出其对应信号

OpCode	Funct	Instr_type	
000000	000000	INSTR_NOP	32'd0
000000	100001	INSTR_ADDU	32'd1
000000	100011	INSTR_SUBU	32'd2
001101	xxxxxx	INSTR_ORI	32'd3
100011	xxxxxx	INSTR_LW	32'd4
101011	xxxxxx	INSTR_SW	32'd5
000100	xxxxxx	INSTR_BEQ	32'd6
001111	xxxxxx	INSTR_LUI	32'd7
000011	xxxxxx	INSTR_JAL	32'd8
000000	001000	INSTR_JR	32’d9
000010	xxxxxx	INSTR_J	32’d10

4. **ControlSignal**

端口定义表及信号介绍

信号名	方向	描述
instr_type	O	当前指令类型，见指令识别部分
NPC_Mode	O	nPC 控制, 0 – 顺序，1 – 条件分支，2 – 跳转， 3 – 跳至寄存器
GRF_WEnable	O	寄存器写使能信号，0：GRF 不写入，1：GRF 写入
ALU_Operation[3:0]	O	ALU 操作信号，0：加，1：减，2：或，3：lui（op2 左移 16）
DM_WEnable	O	内存写使能，0：DM 不写入，1：DM 写入
DM_Mode[1:0]	O	操作位宽，0：字，1：半字，2：字节
EXT_Mode	O	Extender 模式，0：无符号拓展至 32 位，1：有符号拓展至 32 位
MUX_RegWAddr_Sel[1:0]	O	寄存器写地址选择，0：rd，1：rt，2：$ra
MUX_RegWData_Sel[1:0]	O	寄存器写数据选择，0：ALU，1：Mem，2：PC+4
MUX_ALUOp2_Sel[1:0]	O	ALU 操作数 2 选择信号，0：rt，1：EXT

功能：根据输入指令信号输出对应控制信号

信号	nop	addu	subu	ori	lw	sw	beq	lui	jal	jr	j
instr_type	0	1	2	3	4	5	6	7	8	9	10
IFUCG_Mode	0	0	0	0	0	0	1	0	2	3	2
GRF_WEnable	0	1	1	1	1	0	0	1	1	0	0
ALU_Operation[3:0]	x	0	1	2	0	1	1	2	x	x	x
DM_WEnable	0	0	0	0	0	1	0	0	0	0	0
DM_Mode[1:0]	x	x	x	x	0	0	x	x	x	x	x
EXT_Mode	x	x	x	0	1	1	x	2	x	x	x
MUX_RegWAddr_Sel[2:0]	x	0	0	1	1	x	x	1	2	x	x
MUX_RegWData_Sel[2:0]	x	0	0	0	1	x	x	0	2	x	x
MUX_ALUOp2_Sel[2:0]	x	0	0	1	1	1	0	1	x	x	x

四、流水线结构及转发、阻塞

1. 流水线总体结构

流水线分为 5 个阶段，分别为 IF、ID、EX、MEM、WB 阶段，分别执行取指令、取 GRF 及立即数拓展、ALU 运算、写内存和数据写回寄存器的功能。流水线各阶段之间由流水线寄存器（PReg）连接，具体数据域如下：

数据域	说明
Instr[31:0]	指令
PC[31:0]	PC 值
rsData[31:0]	rs 域指明的寄存器所存放的数据
rtData[31:0]	rt 域指明的寄存器所存放的数据
extData[31:0]	EXT 输出的数据
ALUResult[31:0]	ALU 运算结果
memData[31:0]	读内存读出的数据
RegWData[31:0]	若指令回写寄存器，则回写值产生时将被存放至此处

除此之外，还有数个控制信号，如下：

控制信号	说明
clk	时钟信号
reset/clear	清空 PReg
PReg_i_Enable	PReg 寄存器写使能，为 1 时才更新

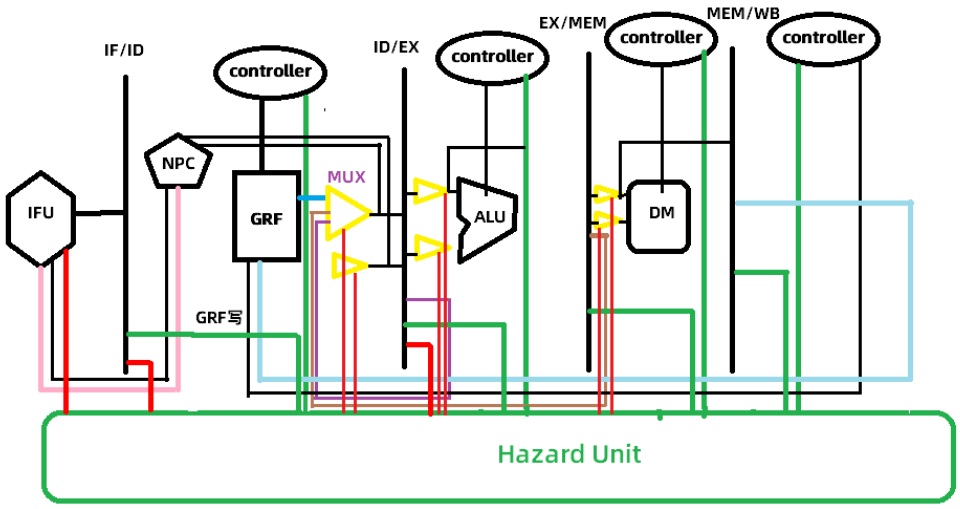
除了某些极特殊特殊情况（为了方便，ID 级 NPC 部件接的是 IF 级 PC）以及转发时的数据，各流水线阶段只需要读取 PReg 内容。

每个流水线阶段都有一个 Controller，4 个 Controller 完全一致，由同一个元件例化而来，只连接本阶段所需要的线。

流水线的冒险由一个专门的冒险单元（Hazard Unit，HU）控制，这个单元接受完成冒险控制功能所需的流水线各阶段输入，输出转发相关 MUX（FMUX）的控制信号以及与阻塞有关的控制信号。具体接口如下：

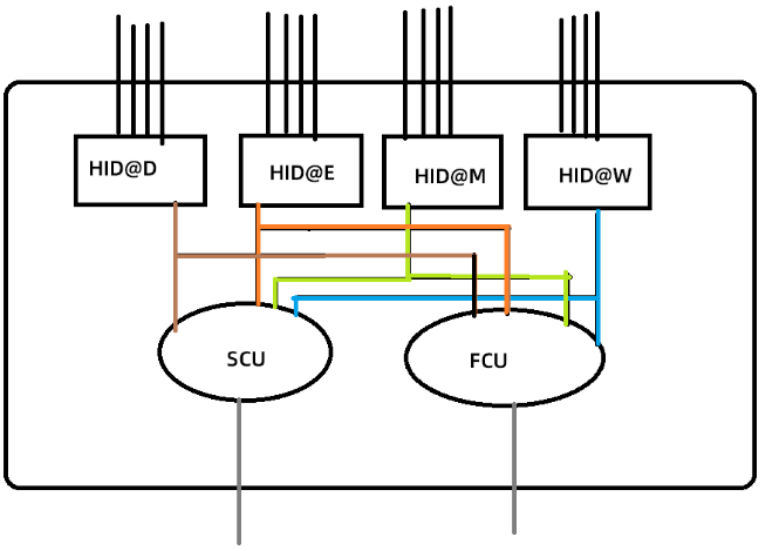
信号名	方向	功能
HU_i_D_Instr[31:0]	I	D 级指令
HU_i_D_Instr_Type[31:0]	I	D 级指令类型
HU_i_D_RegWAddrSel[4:0]	I	D 级写寄存器地址
HU_i_D_GRFWEnable	I	D 级指令寄存器写使能
HU_i_E_Instr[31:0]	I	E 级指令
HU_i_E_Instr_Type[31:0]	I	E 级指令类型
HU_i_E_RegWAddrSel[4:0]	I	E 级写寄存器地址
HU_i_E_GRFWEnable	I	E 级指令寄存器写使能
HU_i_M_Instr[31:0]	I	M 级指令
HU_i_M_Instr_Type[31:0]	I	M 级指令类型
HU_i_M_RegWAddrSel[4:0]	I	M 级写寄存器地址
HU_i_M_GRFWEnable	I	M 级指令寄存器写使能
HU_i_W_Instr[31:0]	I	W 级指令
HU_i_W_Instr_Type[31:0]	I	W 级指令类型
HU_i_W_RegWAddrSel[4:0]	I	W 级写寄存器地址
HU_i_W_GRFWEnable	I	W 级指令寄存器写使能
HU_o_F_IFUEnable	O	IFU 使能，用于阻塞
HU_o_D_PRegEnable	O	IF/ID 级 PReg 使能信号，用于阻塞
HU_o_E_PRegClear	O	ID/EX 级 PReg 复位信号，用于阻塞
HU_o_FwdD_Rs[1:0]	O	ID 级 rs 转发控制信号
HU_o_FwdD_Rt[1:0]	O	ID 级 rt 转发控制信号
HU_o_FwdE_Rs[1:0]	O	EX 级 rs 转发控制信号
HU_o_FwdE_Rt[1:0]	O	EX 级 rt 转发控制信号
HU_o_FwdM_Rs[1:0]	O	MEM 级 rs 转发控制信号
HU_o_FwdM_Rt[1:0]	O	MEM 级 rt 转发控制信号

以下为总体结构示意图：



2. 流水线冒险单元总体结构

流水线冒险单元接受各级输入后，将每一级的输入分别连接至一个 HID（Hazard Instruction Decoder）的实例，由 HID 负责解析有关信息。HID 的输出经过处理（把 Tnew@D 处理成真正的 Tnew），接入 SCU（Stall Control Unit）和 FCU（Forward Control Unit），输出各转发控制信号以及暂停信号。总体结构如下图：



3. 冒险指令解码单元

HID 的输出只用于对冒险的处理，具体接口如下：

信号	方向	说明
HID_i_Instr	I	指令
HID_i_Instr_Type	I	指令类型
HID_i_RegWAddrSel	I	写寄存器地址选择
HID_o_Rs	O	rs
HID_o_Rt	O	rt
HID_o_TuseRs	O	rs 的 Tuse
HID_o_TuseRt	O	rt 的 Tuse
HID_o_TnewD	O	指令处在 D 级时的 Tnew
HID_o_RegWAddr	O	指令所写寄存器的地址（若有）

3. 暂停控制单元

接口定义表

信号	方向	说明
SCU_i_D_Rs	I	D 级指令的 rs
SCU_i_D_Rt	I	D 级指令的 rt
SCU_i_D_TuseRs	I	D 级指令 rs 的 Tuse
SCU_i_D_TuseRt	O	D 级指令 rt 的 Tuse
SCU_i_E_Tnew	O	E 级指令的 Tnew
SCU_i_E_WAddr	O	E 级指令要写的寄存器值
SCU_i_M_Tnew	O	M 级指令的 Tnew
SCU_i_M_WAddr	O	M 级指令要写的寄存器值
SCU_o_Stall	O	是否需要暂停

设置不写寄存器的指令 Tnew 为 0，就不会无谓的暂停。暂停条件：D 级 rs/rt 为 E/M 级要写的寄存器（W 有 GRF 内部转发），且不是 0 号寄存器，且来不及转发（Tuse < Tnew）

3. 转发控制单元

功能：输出 6 个转发 MUX 所需的控制信号

输入接口：D/E/M/W 流水寄存器中指令的 rs/rt/写 Reg 地址/写使能，输出六个 FMUX 控制信号。只要流水线后一阶段指令生成了结果，且 rs/rt 与前方所写地址相同，就转发。

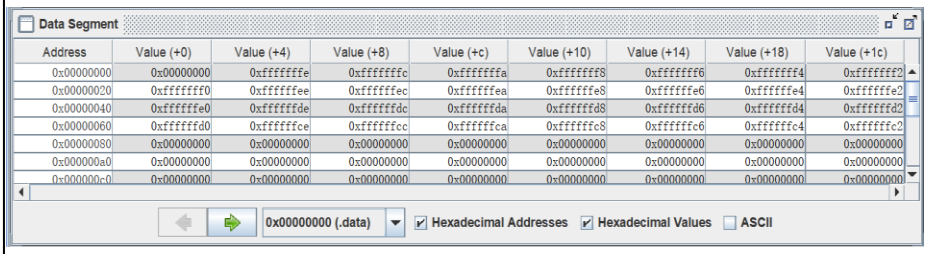
四、测试程序及思考题

首先测试指令基本正确性并尝试一些冒险，基本与 P4 一致但改了寄存器。

```
lui $28, 0
lui $29, 0
# clear, for simulation in mars
ori $1, $0, 0x1234
lui $1, 0x1078
ori $1, $1, 0x1234
addu $0, $1, $1
addu $2, $0, $1
addu $2, $1, $2
# check if $0 stays at 0 and addu
subu $2, $2, $1
# func of subu
ori $3, $0, 5
# check if DM is reset to 0
addu $2, $3, $2
lw $4, 4($0)
sw $2, 3($3)
beq $0, $3, branch
addu $3, $5, $3
addu $5, $5, $3
branch:
addu $5, $5, $3
beq $0, $0, branch2
addu $7, $5, $3
addu $5, $7, $3
branch2:
addu $5, $5, $3
jal jump
addu $6, $5, $3
addu $6, $6, $3
j end
jump:
lui $7, 0
lui $8, 0
lui $9, 0
lui $10, 0
ori $8, 0x0004
ori $9, 0x0002
lui $23, 0
ori $23, 0x80
loop:
sw $10, ($7)
addu $7, $7, $8
subu $10, $10, $9
beq $7, $23, end_loop
addu $3, $5, $3
beq $0, $0, loop
nop
end_loop:
jr $ra
end:
nop
```

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10781234
$v0	2	0x10781239
$v1	3	0x000001e5
$a0	4	0x00000000
$a1	5	0x0000000f
$a2	6	0x000001f9
$a3	7	0x00000000
$t0	8	0x00000004
$t1	9	0x00000002
$t2	10	0xffffffc0
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000080
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00000000
$sp	29	0x00000000
$fp	30	0x00000000
$ra	31	0x0000305c
pc		0x000030a4
hi		0x00000000
lo		0x00000000



其次，使用自动生成程序生成代码，生成程序如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define INSTR_NOP 0
#define INSTR_ADDU 1
#define INSTR_SUBU 2
#define INSTR_ORI 3
#define INSTR_LW 4
#define INSTR_SW 5
#define INSTR_BEQ 6
#define INSTR_LUI 7
#define INSTR_JAL 8
#define INSTR_JR 9
#define INSTR_J 10
// no JR so far, hard to do, lol
const int maxInstructionCount = 1000;

int instrRatioRand(void){
    int randVal = rand() % 120;
    if(randVal == 0)
        return INSTR_NOP;
    else if(randVal < 14)
        return INSTR_ADDU;
    else if(randVal < 32)
        return INSTR_SUBU;
    else if(randVal < 50)
        return INSTR_ORI;
    else if(randVal < 65)
        return INSTR_LW;
    else if(randVal < 75)
        return INSTR_SW;
    else if(randVal < 90)
        return INSTR_BEQ;
    else if(randVal < 105)
        return INSTR_LUI;
    else if(randVal < 110)
        return INSTR_JAL;
    /* else if(randVal < 115)
        return INSTR_JR; */
    else
        return INSTR_J;
}
```

```
// 0 - 3, 31 , 5 registers, more hazard
int rand0Up(void){
    int randVal = rand() % 8;
    if(randVal < 4)
        return 0;
    else if(randVal == 7)
        return 31;
    else
        return randVal - 3;
}

int rand0Down(void){
    int randVal = rand() % 13;
    if(randVal == 0)
        return 0;
    else if(randVal >= 10 && randVal < 13 )
        return 31;
    else
        return (randVal + 2)/3;
}

// Instructions here
void nop(int total){
    printf("L%d: nop\n", total);
}

void three_cal(int total, char * mnemonic){
    int dest = rand0Down();
    int tmp = rand() % 5;
    int src1 = (tmp == 4)?31:tmp;
    tmp = rand() % 5;
    int src2 = (tmp == 4)?31:tmp;
    printf("L%d: %s $%d, $%d, $%d\n", total, mnemonic, dest, src1, src2);
}

void addu(int total){
    three_cal(total, "addu");
}


void subu(int total){
    three_cal(total, "subu");
}
```

```
void ori(int total){
    // randMax is 0x7fff
    printf("L%d: ori $%d, $%d, %d\n", total, rand0Down(), rand0Down(), (ra
nd() + rand())%65536);
}

void lui(int total){
    // randMax is 0x7fff
    printf("L%d: lui $%d, %d\n", total, rand0Down(), (rand() + rand())%655
36);
}

void lw(int total){
    int rawRand = rand();
    int randWord = (rawRand % 1536 < 512) ? 0 : ((rawRand % 1536) - 512);
    int randAddr = randWord << 2;
    printf("L%d: lw $%d, %d($0)\n", total, rand0Down(), randAddr);
}

void sw(int total){
    int rawRand = rand();
    int randWord = ((rawRand) % 1536 < 512) ? 0 : ((rawRand) % 1536 - 512)
;
    int randAddr = randWord << 2;
    printf("L%d: sw $%d, %d($0)\n", total, rand0Down(), randAddr);
}

void beq(int total){
    int randJump = rand() % 64 + 1;
    int label = ((total + randJump) < 999) ? (total + randJump) : 1001;
    printf("L%d: beq $%d, $%d, L%d\n", total, rand0Up(), rand0Up(),label);
}

void jal(int total){
    int randJump = rand() % 64 + 1;
    int label = ((total + randJump) < 999) ? (total + randJump) : 1001;
    printf("L%d: jal L%d\n", total, label);
}

void j(int total){
    int randJump = rand() % 64 + 1;
    int label = ((total + randJump) < 999) ? (total + randJump) : 1001;
    printf("L%d: j L%d\n", total, label);
}
```

```
int main(void){
    srand((unsigned)time(NULL));

    int isLastInstrJump = 0;
    int total = 0;

    do {
        switch(instrRatioRand()){
            case INSTR_NOP:
                nop(total);
                isLastInstrJump = 0;
                break;
            case INSTR_ADDU:
                addu(total);
                isLastInstrJump = 0;
                break;
            case INSTR_SUBU:
                subu(total);
                isLastInstrJump = 0;
                break;
            case INSTR_ORI:
                ori(total);
                isLastInstrJump = 0;
                break;
            case INSTR_LW:
                lw(total);
                isLastInstrJump = 0;
                break;
            case INSTR_SW:
                sw(total);
                isLastInstrJump = 0;
                break;
            case INSTR_BEQ:
                if(isLastInstrJump)
                    total--;
                else
                    beq(total);
                    isLastInstrJump = 1;
                break;
            case INSTR_LUI:
                lui(total);
                isLastInstrJump = 0;
                break;
```

```
            case INSTR_JAL:
                if(isLastInstrJump)
                    total--;
                else
                    jal(total);
                    isLastInstrJump = 1;
                break;
            case INSTR_J:
                if(isLastInstrJump)
                    total--;
                else
                    j(total);
                    isLastInstrJump = 1;
                break;
        }

    } while (total++ < maxInstructionCount);

    printf("L1001: nop\n");
    printf("Exit: nop\n");
}
```

生成的部分代码示例

```
INTI0: lui $gp, 0
INTI1: lui $sp, 0
L0: addu $31, $0, $3
L1: addu $2, $0, $0
L2: beq $3, $2, L30
L3: lui $31, 17681
L4: j L44
L5: sw $2, 3156($0)
L6: subu $31, $3, $2
L7: lui $2, 17031
L8: jal L30
L9: ori $1, $31, 13194
L10: lui $3, 56810
L11: jal L46
L12: ori $1, $2, 20778
L13: sw $3, 1264($0)
L14: beq $0, $2, L62
L15: ori $2, $31, 27822
L16: sw $3, 176($0)
L17: subu $3, $1, $31
L18: lui $31, 53213
```

构造策略：只使用 0，1，2，3，31 号寄存器，减少对 0 号寄存器的写入，提高冒险发生的可能性。跳转指令只向下跳避免死循环，同样 jr 暂不测试留待手动测试。beq 增大 0 号寄存器概率，增大条件成立可能。