北京郵電大學

Beijing University of Posts and Telecommunications

实　验　报　告

课程名称_____计算机组成原理_____

实验名称__虚拟机设计_

___计算机__学院_2018211319_班

姓名_王天乐_　学号_2018210259_

教师___杨旭东_____　　成绩_____

__2020__年__6__月__5__日

目录

一、 实验目的

通过设计仿真模拟器，深入了解计算机指令格式、操作数类型、指令和操作数寻址方式、各指令和相应寄存器功能，并实现**同步和异步中断**功能。

二、 运行环境

Win10 64 位操作系统 PC 机，基于 Visual Studio 开发环境，模拟实现 MIPS 仿真模拟器。

三、 实验设计

3.1 实验说明

在本实验中，我们选择对 MIPS 精简指令集系统进行仿真模拟，设计编程结构为字：32bit，字节：8bit，存储器大小为 4GB，根据 MIPS 三种不同的指令格式对 31 条指令使用 C++面向对象中类继承、静态成员等方法进行了模拟。仿真器可对汇编语言和机器语言进行输入处理，并打印出所有指令的微操作。

3.2 指令格式

3.2.1 R 型指令

助记符	指令格式						寻址方式
Bit #	31~26	25~21	20~16	15~11	10~6	5~0	
R-type	op	rs	rt	rd	shamt	func	
add	000000	rs	rt	rd	00000	100000	寄存器寻址
addu	000000	rs	rt	rd	00000	100001	寄存器寻址
sub	000000	rs	rt	rd	00000	100010	寄存器寻址
subu	000000	rs	rt	rd	00000	100011	寄存器寻址
and	000000	rs	rt	rd	00000	100100	寄存器寻址
or	000000	rs	rt	rd	00000	100101	寄存器寻址
xor	000000	rs	rt	rd	00000	100110	寄存器寻址
nor	000000	rs	rt	rd	00000	100111	寄存器寻址
slt	000000	rs	rt	rd	00000	101010	寄存器寻址
sltu	000000	rs	rt	rd	00000	101011	寄存器寻址
sll	000000	00000	rt	rd	shamt	000000	寄存器寻址
srl	000000	00000	rt	rd	shamt	000010	寄存器寻址
sra	000000	00000	rt	rd	shamt	000011	寄存器寻址
sllv	000000	rs	rt	rd	00000	000100	寄存器寻址
srlv	000000	rs	rt	rd	00000	000110	寄存器寻址
srav	000000	rs	rt	rd	00000	000111	寄存器寻址
jr	000000	rs	00000	00000	00000	001000	寄存器寻址

其中：

- R_s, R_t 分别为第一、二源操作数；R_d 为目标操作数；
- OP：指令的基本操作，为操作码；在 R 型指令中为 0
- $shamt$：位移量
- $funct$：功能码

3.2.2 I 型指令

助记符	指令格式				寻址方式
Bit #	31~26	25~21	20~16	15~0	
I-type	op	rs	rt	immediate	
addi	001000	rs	rt	immediate	立即数寻址
addiu	001001	rs	rt	immediate	立即数寻址
andi	001100	rs	rt	immediate	立即数寻址
ori	001101	rs	rt	immediate	立即数寻址
xori	001110	rs	rt	immediate	立即数寻址
lui	001111	00000	rt	immediate	立即数寻址
lw	100011	rs	rt	immediate	基址寻址
sw	101011	rs	rt	immediate	基址寻址
beq	000100	rs	rt	immediate	PC 相对寻址
bne	000101	rs	rt	immediate	PC 相对寻址
slti	001010	rs	rt	immediate	立即数寻址
sltiu	001011	rs	rt	immediate	立即数寻址

其中：

- 双目、$Load/Store$：R_s 和立即数是源操作数，R_t 为目标操作数；
- 条件转移：R_s, R_t 均为源操作数；

3.2.3 J 型指令

助记符	指令格式		寻址方式
Bit #	31~26	25~0	
J-type	op	address	
j	000010	address	伪直接寻址
jal	000011	address	伪直接寻址
inti	110100	address	中断向量

其中：

- 26 位立即数作为跳转目标地址的部分地址

3.2.4 其他指令

助记符	指令格式			寻址方式
Bit #	31~26	25~21	20~0	
I-type	op	rs	\	
read	110000	rs	\	寄存器寻址
write	110001	rs	\	寄存器寻址
disp	110010	\	imm	直接寻址
ei	110011	\	\	\

3.3 寄存器

3.3.1 通用寄存器

寄存器名	寄存器编号	用途说明
$s0	0	保存固定的常数 0
$at	1	汇编器的临时变量
$v0～$v1	2～3	子函数调用返回结果
$a0～$a3	4～7	函数调用参数 1～3
$t0～$t7	8～15	临时变量，函数调用时不需要保存和恢复
$s0～$s7	16～23	函数调用时需要保存和恢复的寄存器变量
$t8～$t9	24～25	临时变量，函数调用时不需要保存和恢复
$k0～$k1	26～27	中断、异常处理程序使用
$gp	28	全局指针变量(Global Pointer)
$sp	29	堆栈指针变量(Stack Pointer)
$fp	30	帧指针变量(Frame Pointer)
$ra	31	返回地址(Return Address)

操作数为 5bits，因此寄存器个数有 $2^5 = 32$ 个。

3.3.2 乘除寄存器

2 个 32 位乘、商寄存器 H_i 和 L_0；乘法分别存放 64 位乘积的高、低 32 位；除法时分别存放余数和商。

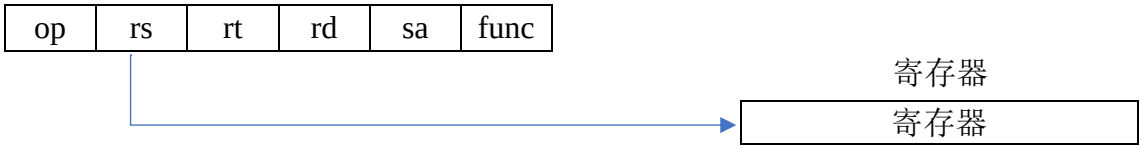
3.3.3 精度寄存器

32 个 32 位单精度浮点寄存器$f_0 - f_{31}$

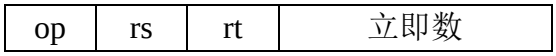
3.4 寻址方式

MIPS 只有 5 种寻址方式，并且每种指令只有 1 种寻址方式

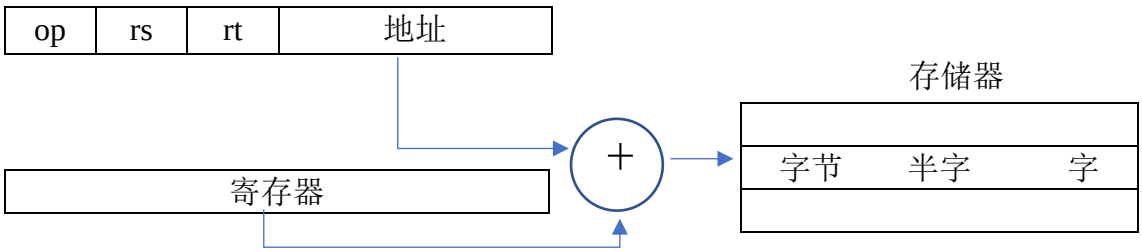
- 寄存器直接寻址



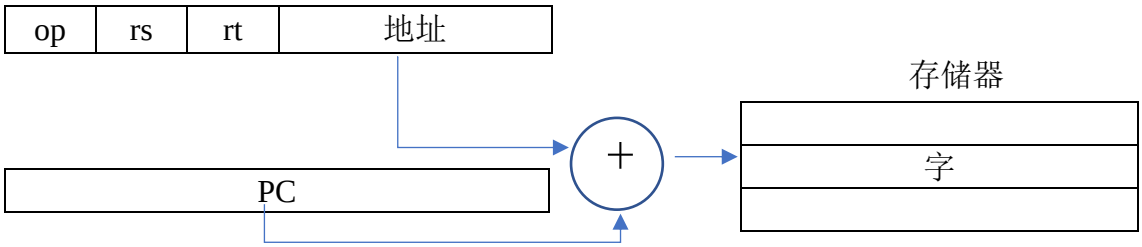
- 立即数寻址



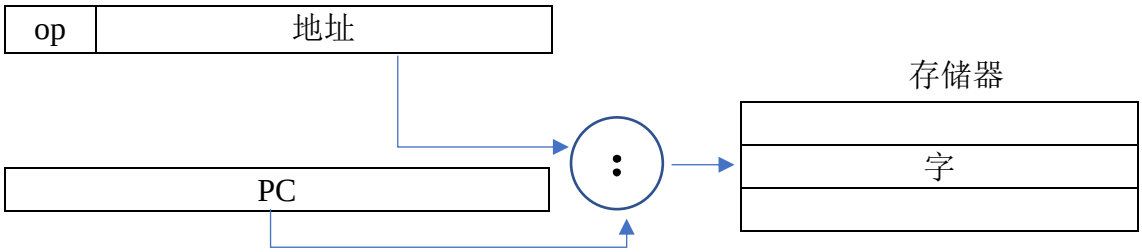
- 基址偏移量寻址



- PC 相对寻址

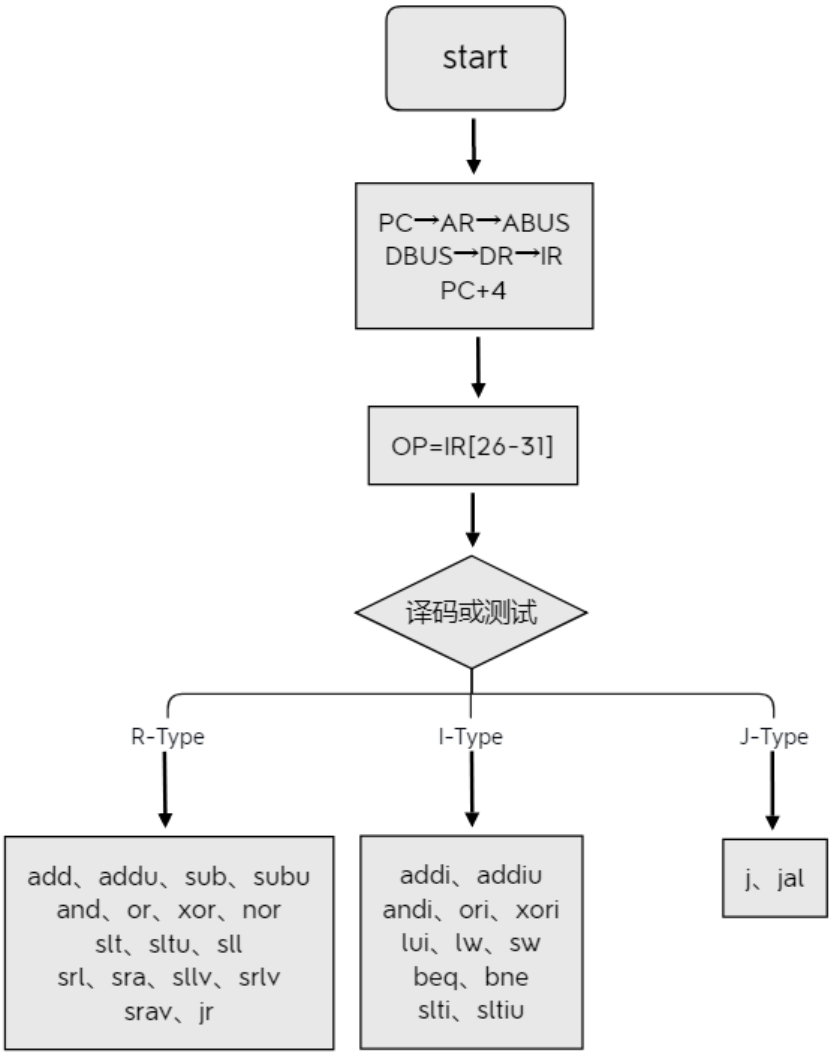


- 伪直接寻址（页面寻址）



3.5 指令流程图

指令执行主流程图：



3.5.1 R 型指令

- 算数类指令

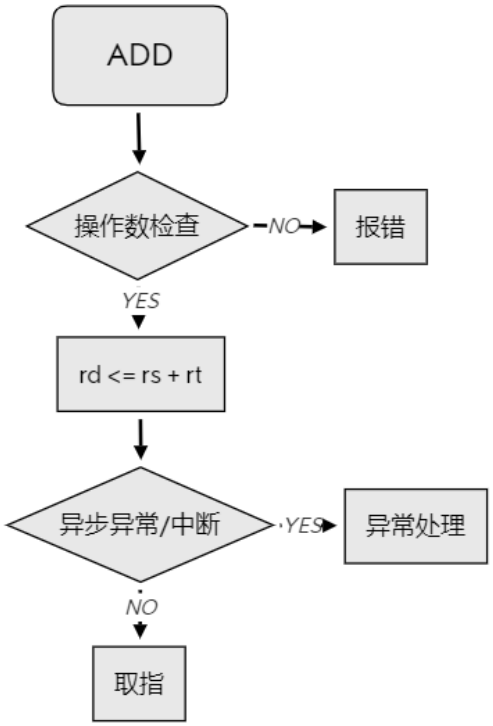
ADD

描述：有符号“加”算数运算

格式：add rd, rs, rt

功能：rd <= rs + rt

备注：rs, rt, rd 均使用寄存器寻址方式



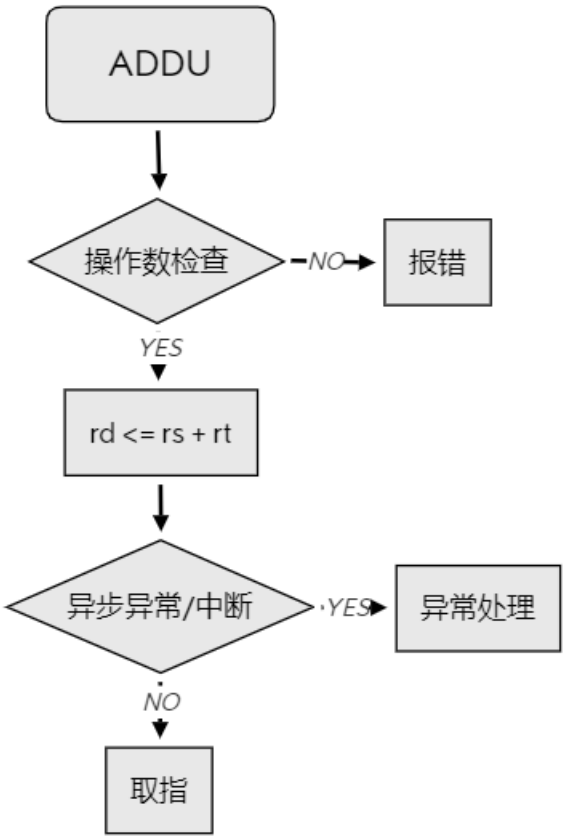
ADDU

描述：无符号“加”算数运算

格式：addu rd, rs, rt

功能：rd <= rs + rt

备注：rs, rt, rd 均为无符号



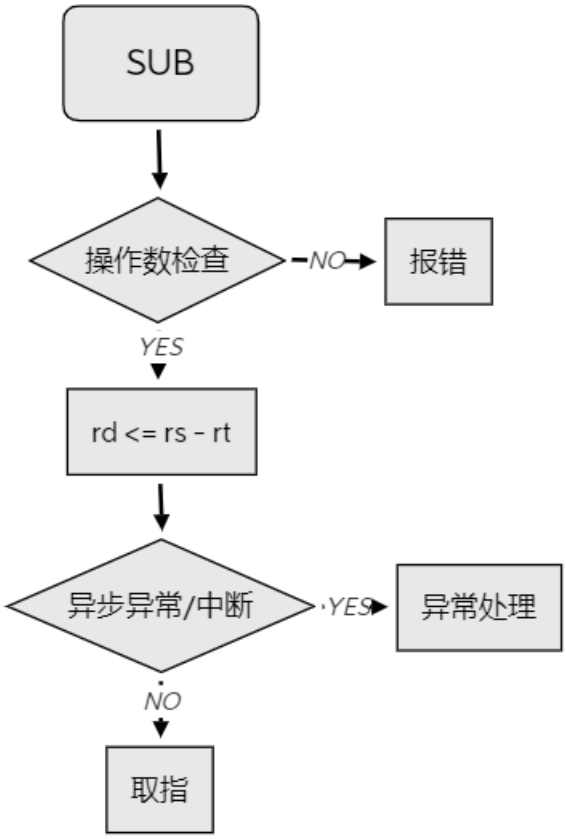
SUB

描述：有符号“减”算数运算

格式：sub rd, rs, rt

功能：rd <= rs – rt

备注：rs, rt, rd 均使用寄存器寻址方式



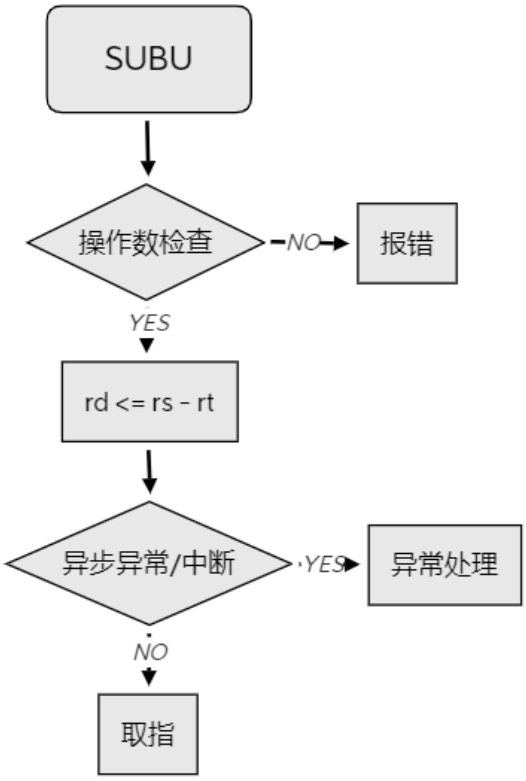
SUBU

描述：无符号“减”算数运算

格式：subu rd, rs, rt

功能：rd <= rs – rt

备注：rs, rt, rd 均为无符号数，使用寄存器寻址方式



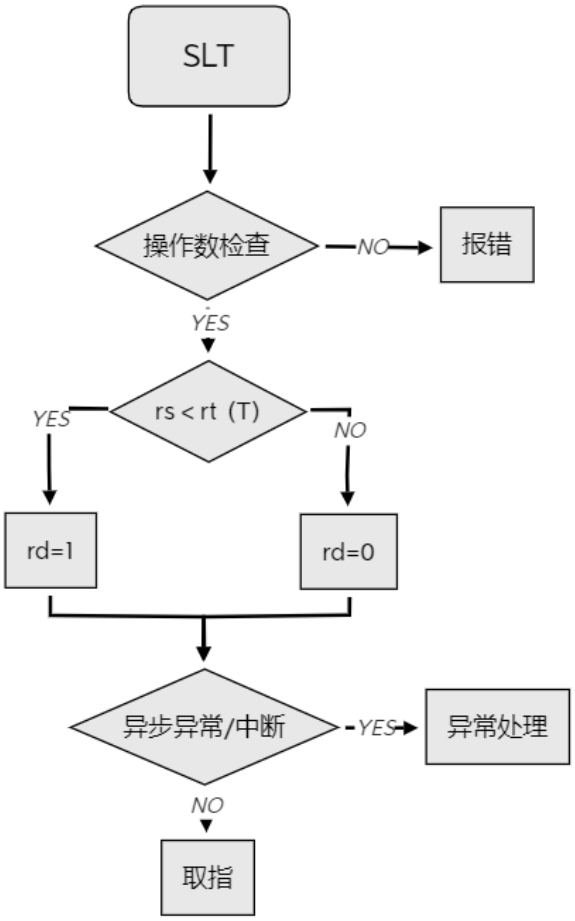
SLT

描述：有符号数比较大小

格式：slt rd, rs, rt

功能：if (rs < rt) rd=1，else rd=0

备注：rs，rt，rd 均使用寄存器寻址



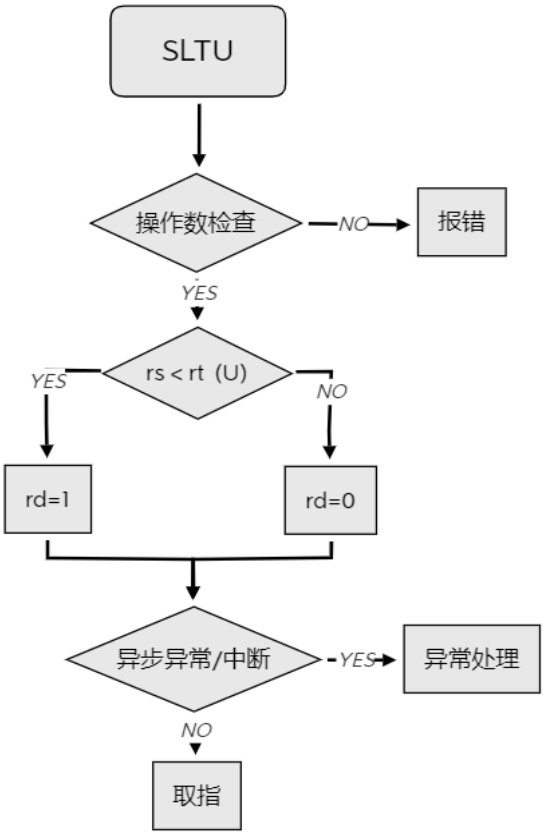
SLTU

描述：无符号数比较大小

格式：slt rd, rs, rt

功能：if (rs < rt) rd=1，else rd=0

备注：rs，rt，rd 均使用寄存器寻址



- 逻辑类指令

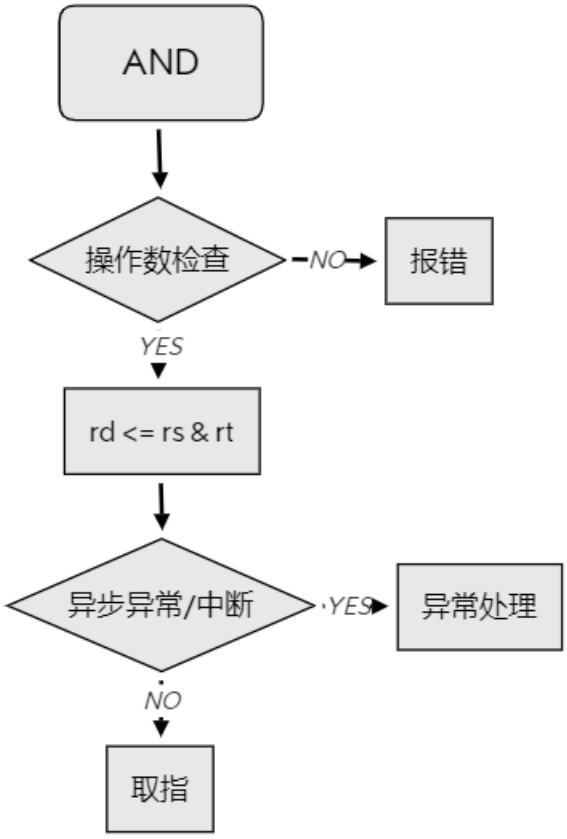
AND

描述：“与”逻辑运算

格式：and rd, rs, rt

功能：rd <= rs & rt

备注：rs, rt, rd 均使用寄存器寻址方式



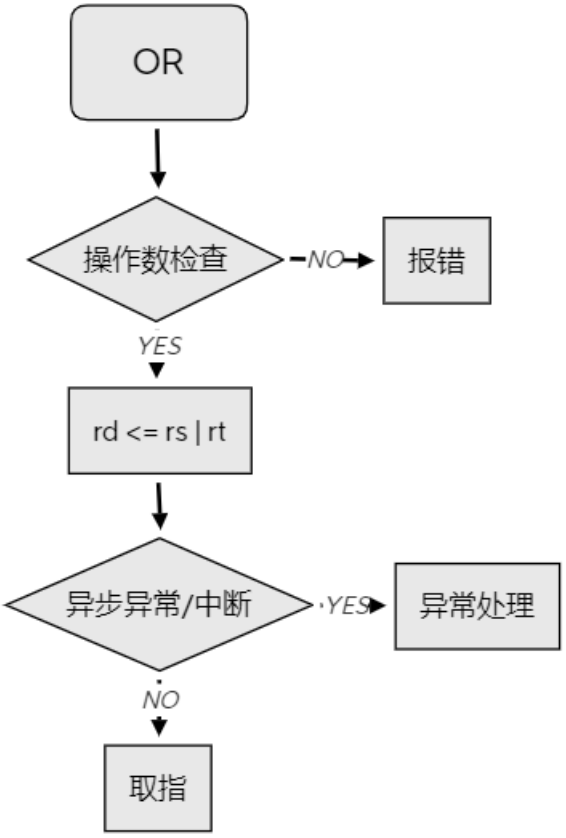
OR

描述：“或”逻辑运算

格式：or rd, rs, rt

功能：rd <= rs | rt

备注：rs, rt, rd 均使用寄存器寻址方式



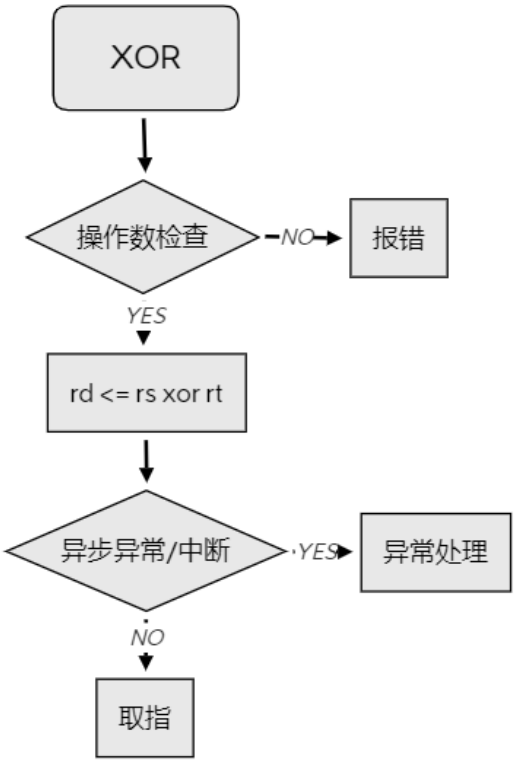
XOR

描述：“异或”逻辑运算

格式：xor rd, rs, rt

功能：rd <= rs xor rt

备注：rs, rt, rd 均使用寄存器寻址方式



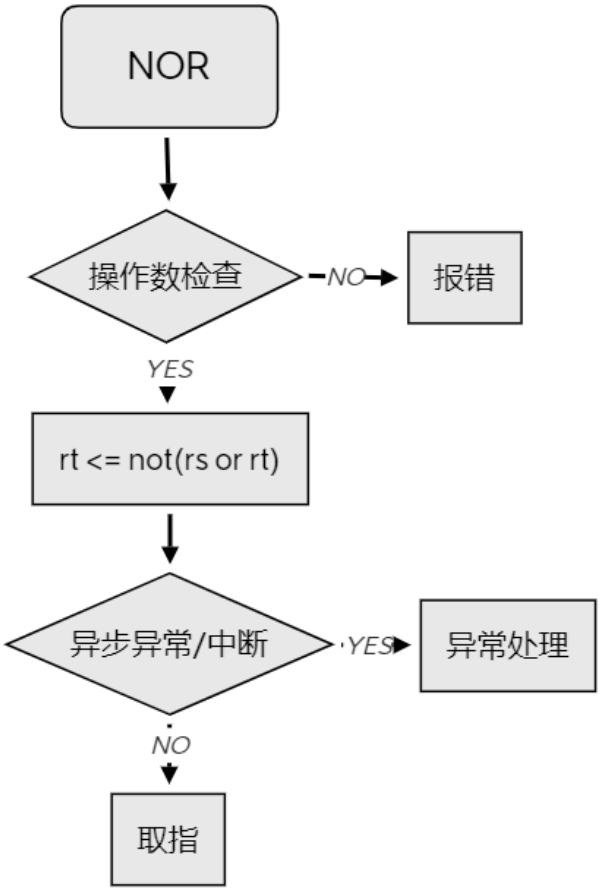
NOR

描述：“或非”逻辑运算

格式：nor rd, rs, rt

功能：rt <= not(rs or rt)

备注：rs，rt，rd 均使用寄存器寻址



- 位移类指令

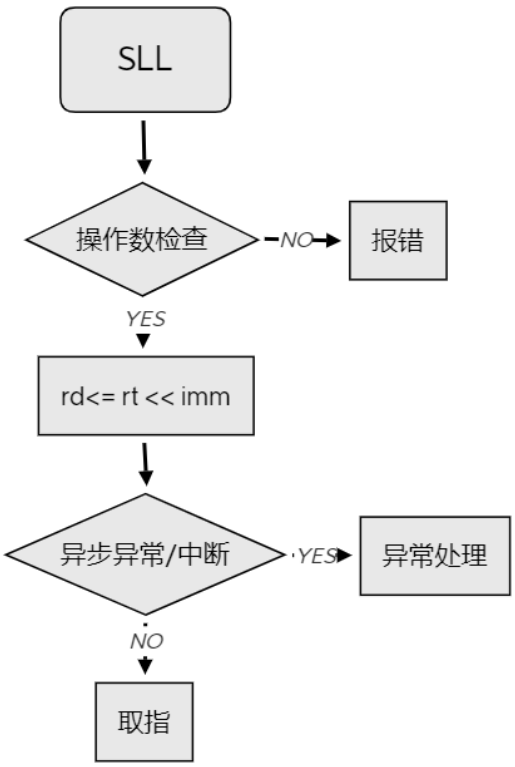
SLL

描述：逻辑、算术左移，其中位移量直接输入

格式：slt rd, rt，imm

功能：rd = rt << imm

备注：rt，rd 均使用寄存器寻址，imm 是立即数



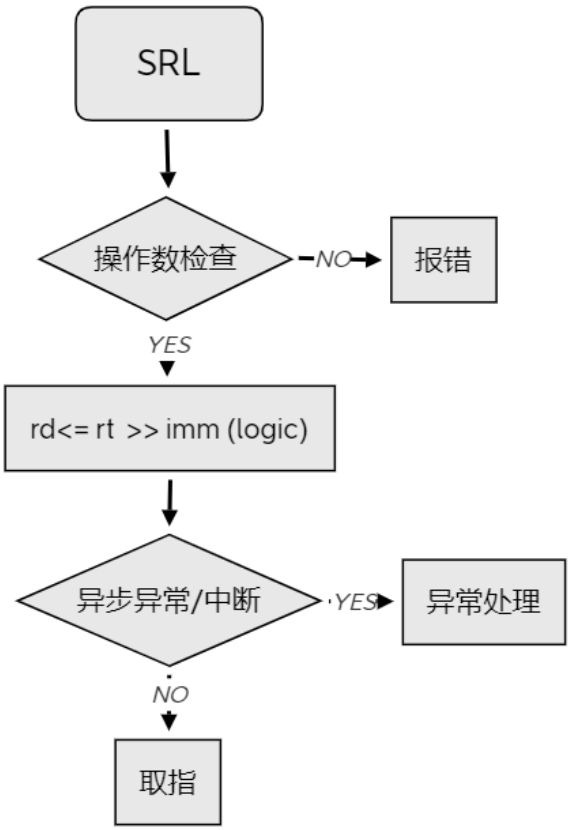
SRL

描述：逻辑右移，其中位移量直接输入

格式：srl rd, rt，imm

功能：rd = rt >> imm (logical)

备注：rt，rd 均使用寄存器寻址，imm 是立即数



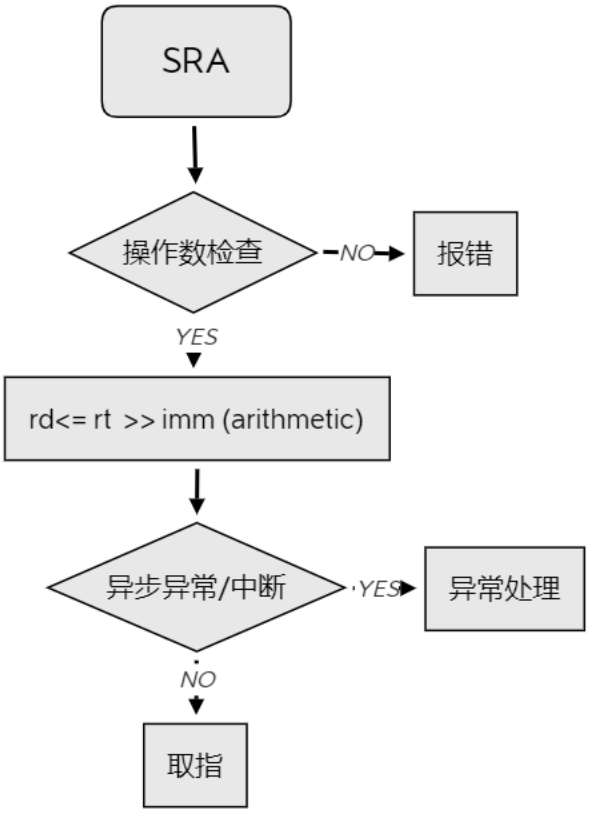
SRA

描述：算数右移，其中位移量直接输入

格式：srl rd, rt，imm

功能：rd = rt >> imm (arithmetic)

备注：1. rt，rd 均使用寄存



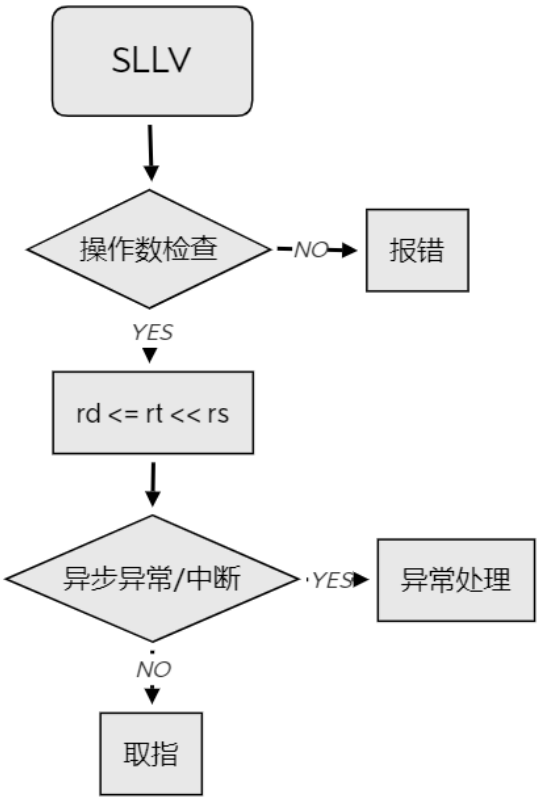
SLLV

描述：逻辑、算术左移，其中位移量存放在寄存器中

格式：slt rd, rt, rs

功能：rd <= rt << rs

备注：rs, rt, rd 均使用寄存器寻



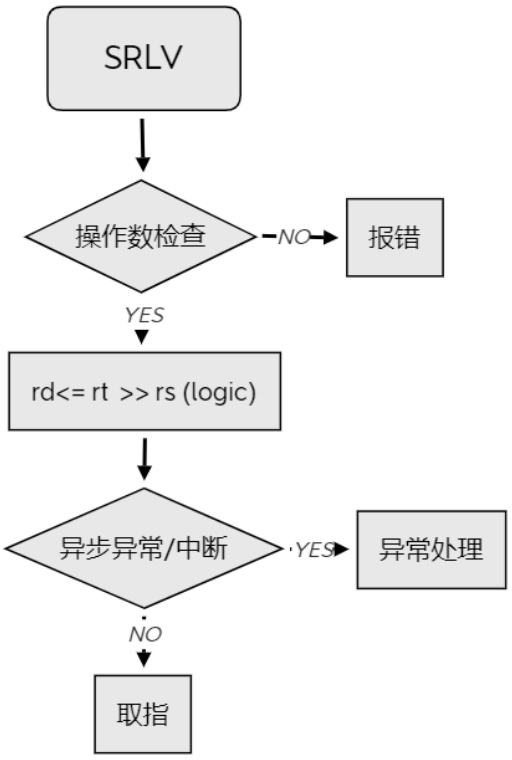
SRLV

描述：逻辑右移，其中位移量存放在寄存器中

格式：slt rd, rt, rs

功能：rd <= rt >> rs (logic)

备注：rs, rt, rd 均使用寄存器寻址



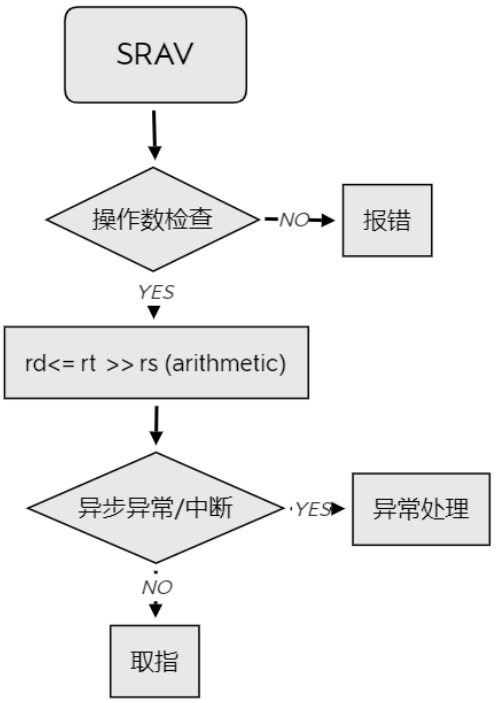
SRAV

描述：算术右移，其中位移量存放在寄存器中

格式：slt rd, rt, rs

功能：rd <= rt >> rs (arithmetic)

备注：
1. rs, rt, rd 均使用寄存器寻址
2. 注意运算保留符号位



- 跳转指令

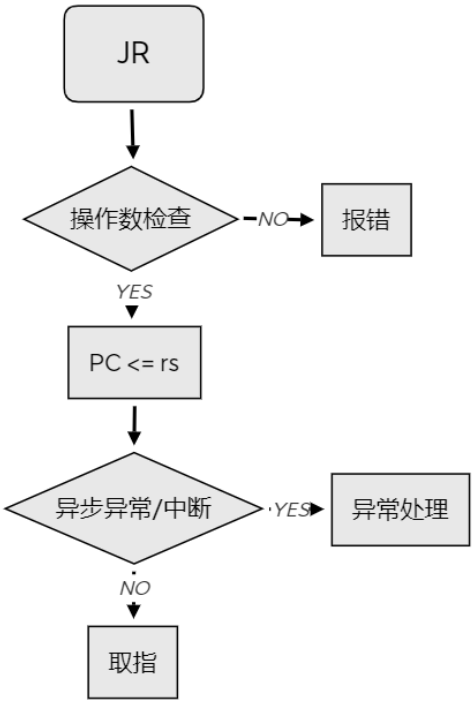
JR

描述：跳转指令，跳转到寄存器中存放的地址

格式：jr rs

功能：PC <= rs

备注：rs 使用寄存器寻址



3.5.2 I 型指令

- 算数指令

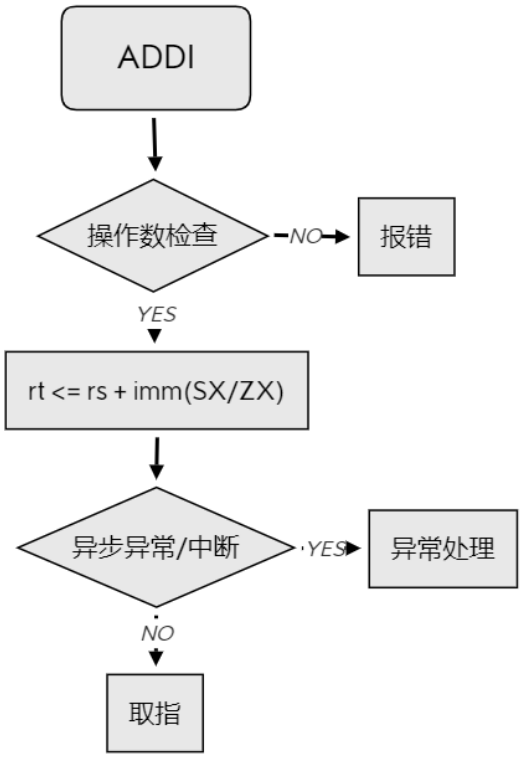
ADDI/ADDIU

描述：有符号/无符号算术立即数加法

格式：addi rt,rs,imm

功能：rt <= rs + imm(SX/ZX)

备注：rs、rt 使用寄存器寻址
imm 是立即数

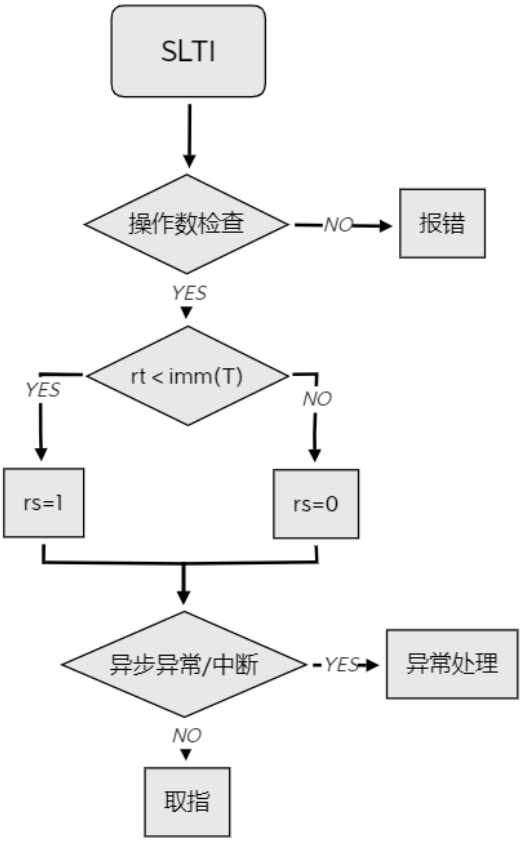


SLTI

描述：寄存器小于立即数置一

格式：slti rs,rt,imm

功能：
if (rs <(sign-extend)immediate) rt=1 else rt=0 ；
备注：rt 使用寄存器寻址



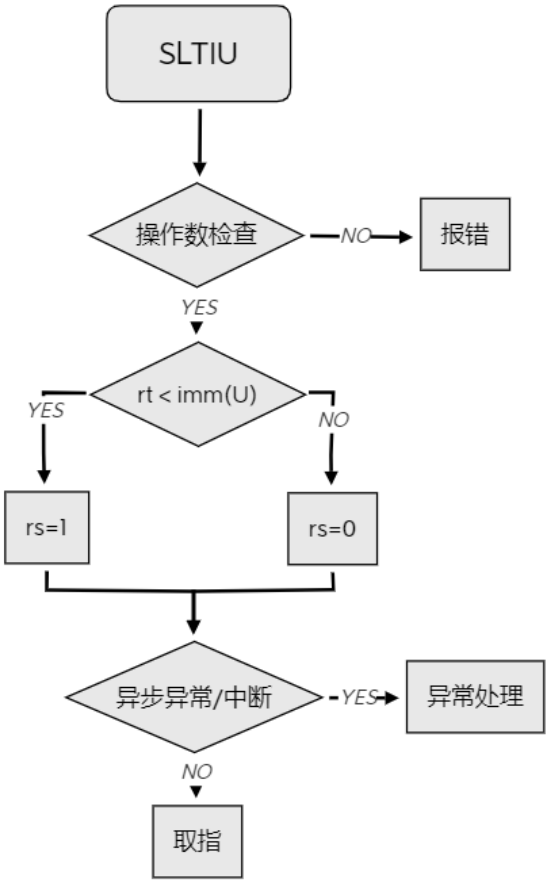
SLTIU

描述：寄存器小于立即数置一(无符号)

格式：sltiu rs,rt,imm

功能：
if (rs <(zero-extend)immediate) rt=1 else rt=0 ；

备注：rt 使用寄存器寻址
imm 是立即数



- 逻辑类指令

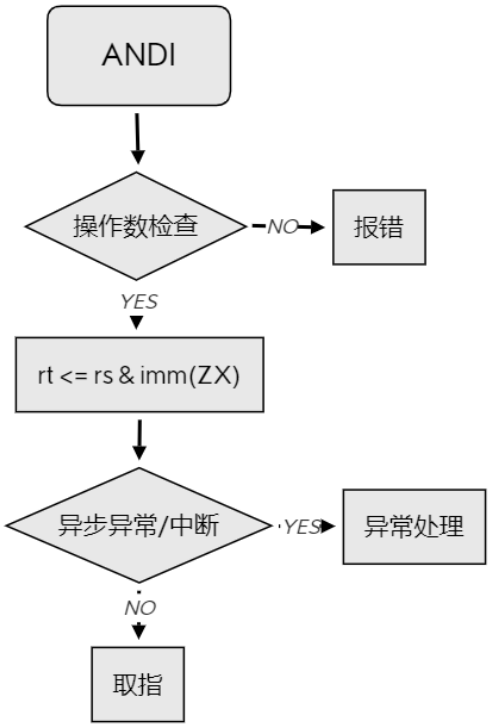
ANDI

描述：立即数逻辑与

格式：andi rt,rs,imm

功能：rt <= rs & imm(ZX)

备注：rs、rt 使用寄存器寻址
imm 是立即数(零扩展)



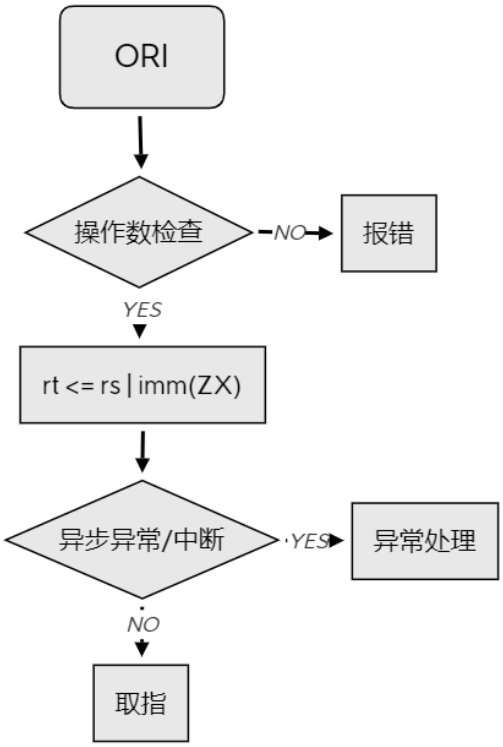
ORI

描述：立即数逻辑或

格式：ori rt,rs,imm

功能：rt <= rs |imm(ZX)

备注：rs、rt 使用寄存器寻址
imm 是立即数(零扩展)



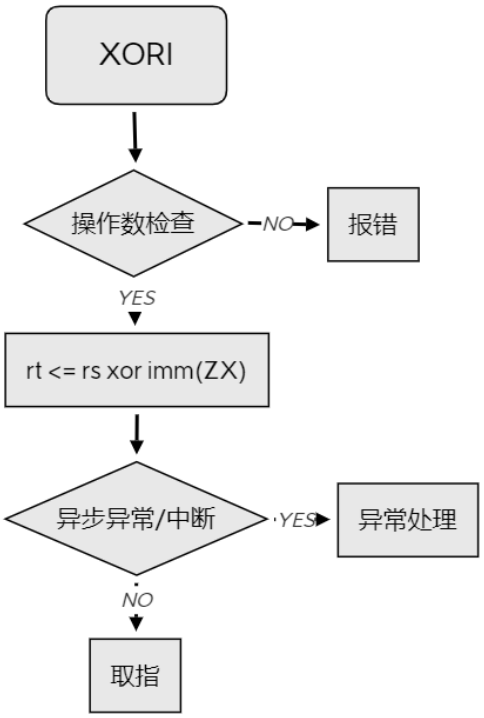
XORI

描述：立即数逻辑异或

格式：xori rt,rs,imm

功能：rt <= rs ^ imm(ZX)

备注：rs、rt 使用寄存器寻址
imm 是立即数(零扩展)



- 载入类指令

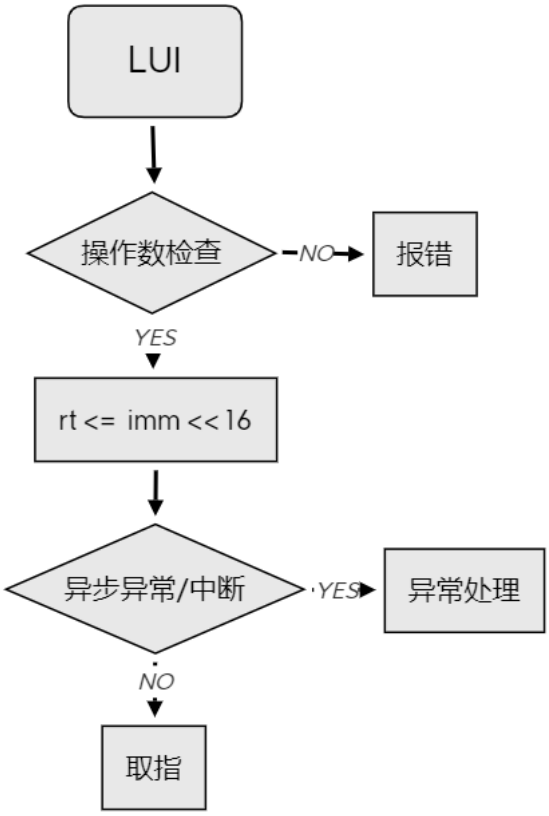
LUI

描述：加载高位立即数

格式：lui rt,,imm

功能：rt <= imm<<16

备注：rt 使用寄存器寻址
imm 是立即数



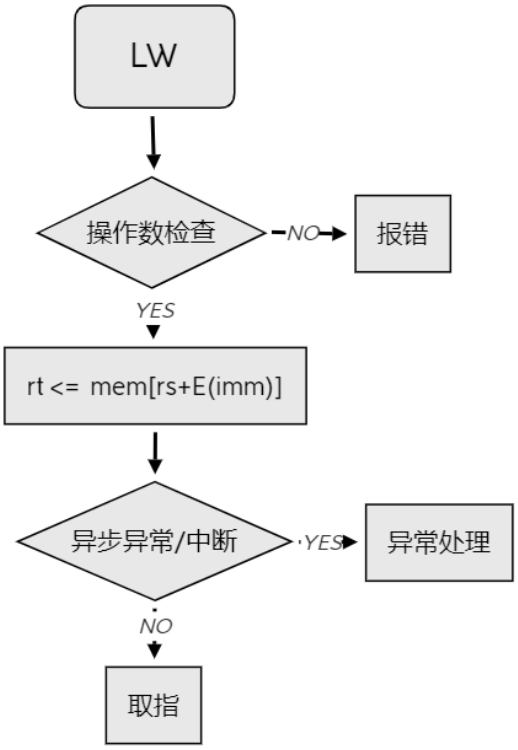
LW

描述：从存储器中读取数据

格式：lw　rt, imm(rs)

功能：rt ← mem[rs+E(imm)]

备注：rt 使用寄存器寻址
imm 是立即数



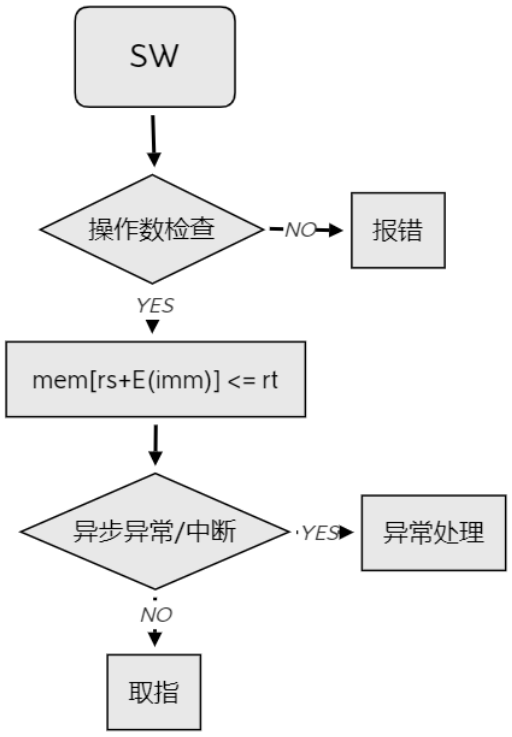
SW

描述：把数据保存到存储器

格式：sw　rt, imm(rs)

功能：mem[rs+E(imm)] ←rt

备注：rt 使用寄存器寻址
imm 是立即数



- 跳转类指令

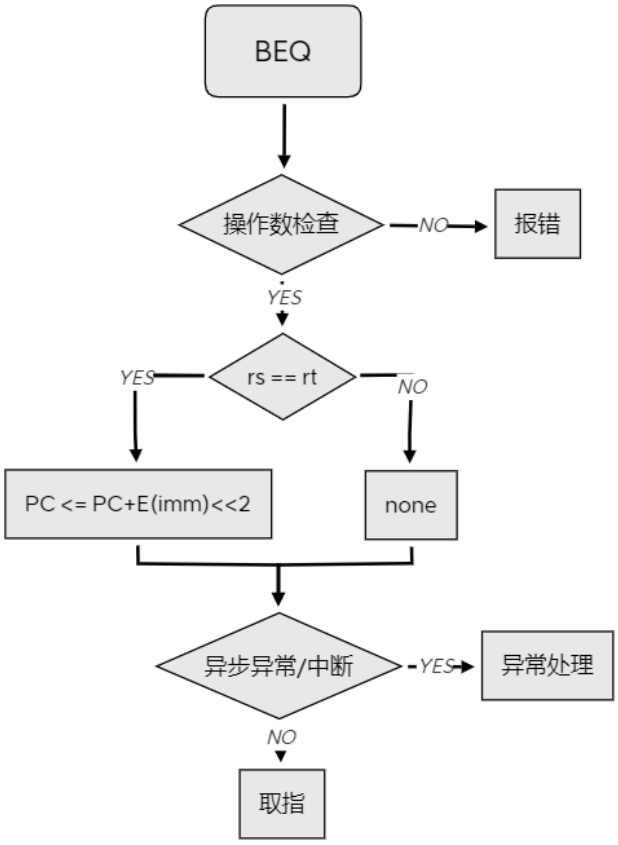
BEQ

描述：寄存器相等则转移

格式：beq　rs, rt, imm

功能：
if(rs==rt)
PC←PC+E(imm)<<2

备注：rt，rs 使用寄存器寻址



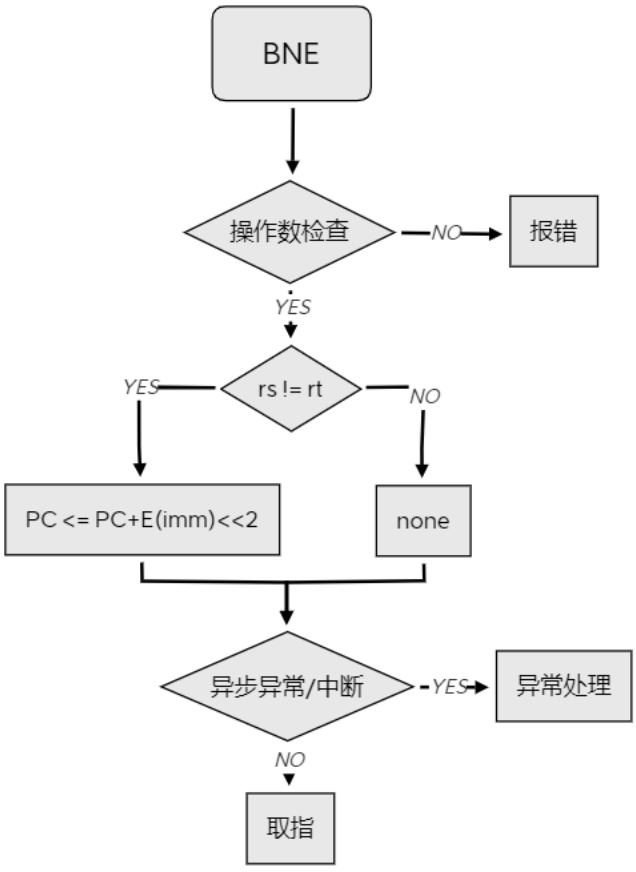
BNE

描述：寄存器不等则转移

格式：bne　rs, rt, imm

功能：
if(rs!=rt)
PC←PC+E(imm)<<2

备注：rs,rt 使用寄存器寻址



3.5.3 J 型指令

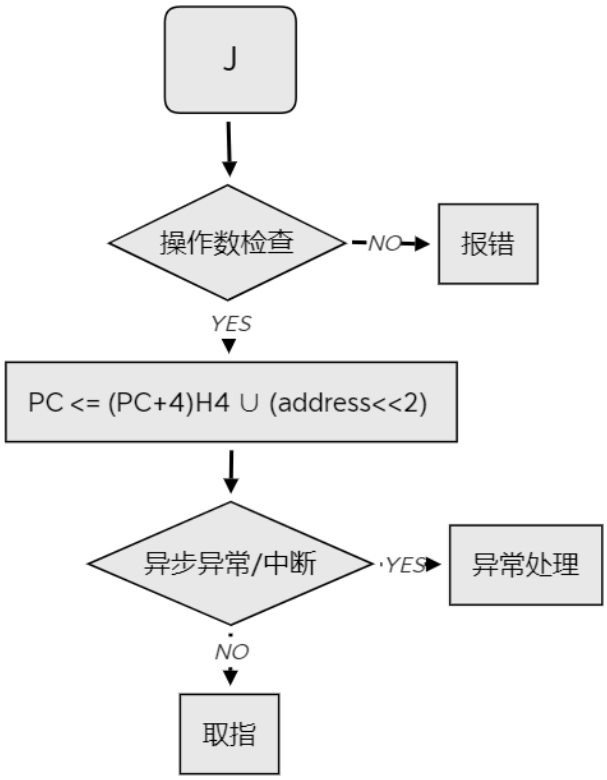
- 跳转类指令

J

描述：无条件跳转

格式：j　address

功能：PC← (PC+4)H4 ∪ (address<<2)

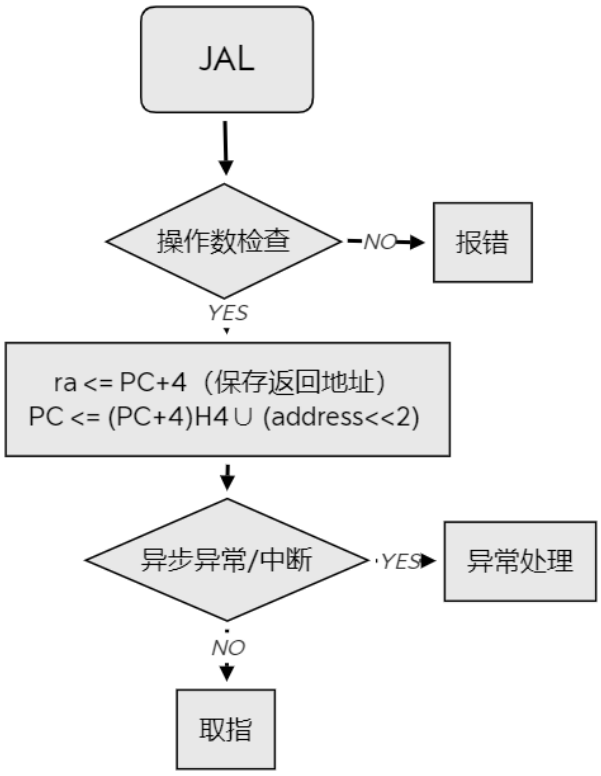


JAL

描述：调用与联接

格式：jal　address

功能：ra←PC+4（保存返回地址） PC ← (PC+4)H4∪ (address<<2)



3.5.4 其他指令

READ

描述：读入终端输入的数据

格式：read rs

功能：rs=input()

WRITE

描述：在终端输出的寄存器的值

格式：write rs

功能：print rs

DISP

描述：在终端输出的字符串方便调试

格式：disp imm

功能：cout<<disp(imm);

EI

描述：允许/禁止程序中断

格式：ei

功能：ei = ~ei

INTI

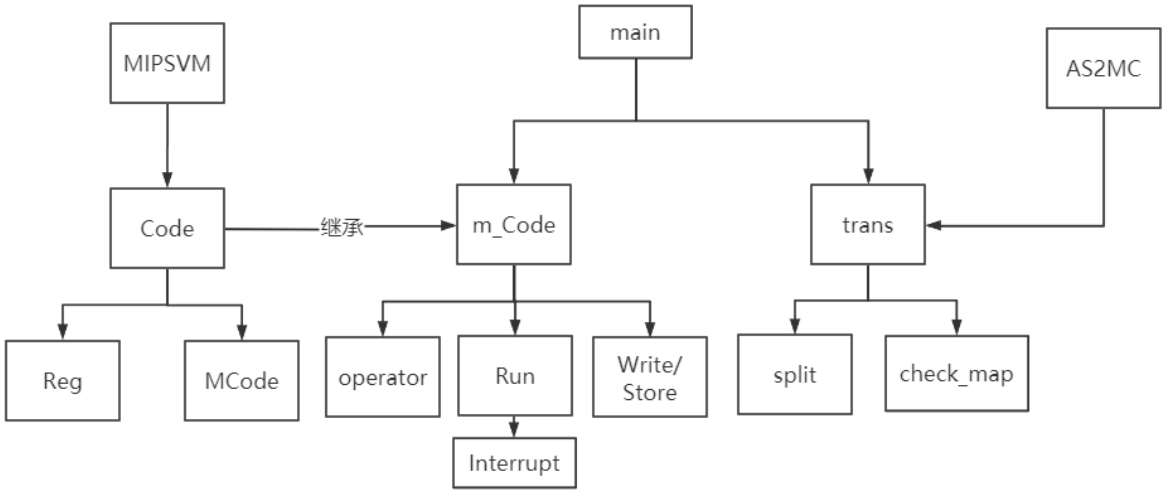
描述：中断指令

格式：inti address

功能：利用中断向量跳转到中断程序入口

3.6 各模块功能

模块调用关系图：



main.cpp

- `int main(int argc, char **argv);`

功能描述：程序的主函数，对输入文件格式进行判断。

参数说明：argc 为命令行参数个数，agrv 存放命令行参数。

- `void Run()`

功能描述：从 PC 开始执行内存中的指令。

- `void Store(uint32_t inst)`

功能描述：将指令储存在内存中。

参数说明：inst 为指令机器码。

- `void Write(uint32_t inst)`

功能描述：将指令写入内存，PC 不动以便后续程序执行。

参数说明：inst 为指令机器码。

AS2MC.h

- `class AS2MC;`

功能描述：AS2MC 类，进行汇编语言到机器码的转换。

- `static bool check_map(const std::string &s);`

功能描述：检查一个参数是否为有效的寄存器

参数说明：递归调用的终结状态

```
static bool check_map(const std::string& s,Args&&... args);
```

功能描述：检查 n 个参数是否为有效的寄存器

参数说明：s 为第一个寄存器，args 为参数包，在 **sizeof…(args)≥** 1时递归调用

MIPSVM.h

- `class Code;`

功能描述：Code 类

- `uint32_t GetPC();`

功能描述：获取当前 PC 值

- `uint32_t &mem(uint32_t idx);`

功能描述：获取 idx 位置的内存值

参数说明：idx 为指定地址

- `void ResetPC(uint32_t address);`

功能描述：重置当前 PC 值为 address

- `struct Reg;`

功能描述：建立寄存器结构

- `Reg(uint8_t val) : idx(val);`

功能描述：隐式构造函数,以便传参时直接传入寄存器的序号。

参数说明：val 为寄存器序号。

- `Reg &operator=(const Reg &rhs);` `Reg &operator=(uint32_t rhs);`

功能描述：重载赋值操作符

- `explicit MCode(uint8_t op, Reg rs, Reg rt, Reg rd, uint8_t shamt, uint8_t fcn );`

功能描述：建立 R-Type 指令

- `explicit MCode(uint8_t op, Reg rt, Reg rs, uint16_t imm);`

功能描述：建立 I-Type 指令

- `explicit MCode(uint8_t op, uint32_t addr);`

功能描述：建立 J-Type 指令

- `union {struct{}J_type;struct{}J_type;struct{}J_type;};`

功能描述：使用 union+struct 建立指令格式

- `uint32_t xxx(Reg rs, Reg rt, Reg rd, uint8_t shamt);`

功能描述：完成 xxx 指令功能

参数说明：rs，rt，rd 分别为 MIPS 指令中的源和目的寄存器，shamt 为位移量

- `static int IE;`

参数说明：= 0 中断允许，初值不允许中断

- `static int EXL, ERL;`

参数说明：任何一个=1，禁止中断

- `static int excpHdAddre[MAXN];`

参数说明：定义中断向量表，查找中断服务程序入口

- `static int inta;`

参数说明：是否有中断响应

- `static int canInterrupt;`

参数说明：限制异步执行单级中断

AS2MC.cpp

- `vector<string> AS2MC::split(const string &s, const string &seperator);`

功能描述：对输入字符串根据操作符进行分割格式化

参数说明：s 为输入字符串，seperator 为分隔符

- `uint32_t AS2MC::trans(std::string ins);`

功能描述：将汇编语言转换为机器语言以便处理

参数说明：ins 为指令的汇编语言

四、 实验过程日志（遇到的问题及解决）

4.1 代码实现

首先参考并学习了一些关于 MIPS 具体实现的代码，如 JIT、基于 qt 的 mips 虚拟机等等，之后我利用 C++面向对象编程完成了对机器语言和汇编语言的执行。之后添加了加载指令到内存以及中断的功能。

4.2 虚拟机调试

构建完 MIPS 虚拟机，编写汇编程序测试时，虽然能正确输出结果，但效果不直观。我们在 MIPS 虚拟机中设计了 write 指令，输出指定寄存器的值，打印到终端中。因此在打印结果时的操作就是依次将存储器中的指定位置地值拷贝到特定寄存器中，再 write 该寄存器。这样的设置导致的问题是，在输出中会参杂许多指令信息与状态信息，不能直观地显示结果数列。

因此我们更改 write 指令功能，不直接打印，而是存储在一个 vector 中，同时将该 vector 设置成 extern 属性，保证其能够在多个部件中被调用。在检测到指令全部运行完毕后，再调用该 vector 依次输出。

4.3 中断调试

加载程序到内存的时候，在第一次虚拟机中只有 Write 函数，此函数 PC 值不会随指令的写入而改变，但在加载内存时，不需要固定 PC 值，因此另写一个 Store 函数，通过改变 PC 值将程序加载到指定地址的内存中。

五、 测试说明

5.1 运行说明

- 斐波那契数列程序：输入数列长度 n，输出长度为 n 的斐波那契数列。
- 冒泡排序程序：输入数列长度 n，并逐个输入数组中的 n 个元素，输出排序后的结果。
- 中断程序：首先将中断服务程序加载到所给地址位置，然后执行终端测试程序，输入要执行的中断服务程序编号，然后用中断向量映射到对应的地址上，执行中断服务程序，在执行结束后返回主程序。

5.2 程序汇编代码

- 斐波那契数列汇编程序代码

```
1.  #AS
2.  read $a0
```

```
addi $t0,$zero,0
addi $t1,$zero,0
addi $a1,$zero,0
addi $a2,$zero,0
addi $a3,$zero,1
sw $a3,1000($t1)
add $a1,$a2,$zero
add $a2,$a3,$zero
add $a3,$a1,$a2
addi $t0,$t0,1
addi $t1,$t1,4
bne $t0,$a0,-32
addi $t0,$zero,0
addi $t1,$zero,0
lw $a1,1000($t1)
write $a1
addi $t0,$t0,1
addi $t1,$t1,4
bne $t0,$a0,-24
```

- 冒泡排序汇编程序代码

```
#AS
read $a0
addi $t0,$zero,0
addi $t1,$zero,0
read $a1
sw $a1,1000($t1)
addi $t0,$t0,1
addi $t1,$t1,4
bne $t0,$a0,-24
addi $t0,$zero,0
addi $t1,$zero,0
addi $t2,$t0,1
addi $t3,$t1,4
lw $s0,1000($t1)
lw $s1,1000($t3)
slt $s2,$s1,$s0
beq $s2,$zero,16
addi $s3,$s0,0
addi $s0,$s1,0
addi $s1,$s3,0
sw $s0,1000($t1)
sw $s1,1000($t3)
```

```
addi $t2,$t2,1
addi $t3,$t3,4
bne $t2,$a0,-52
addi $t0,$t0,1
addi $t1,$t1,4
addi $t2,$t0,1
addi $t3,$t1,4
bne $t2,$a0,-72
addi $t0,$zero,0
addi $t1,$zero,0
lw $a1,1000($t1)
write $a1
addi $t0,$t0,1
addi $t1,$t1,4
bne $t0,$a0,-24
```

- 中断服务程序代码

```
#AS
0
addi $s0 $a1,0
addi $a1 $zero,0
disp 4
ei
addi $a1 $s0,0
jr $ra
32
addi $s0 $a1,0
slti $a1 $zero,1
disp 4
ei
addi $a1 $s0,0
jr $ra
64
addi $s0 $a2,0
ori $a2 $zero,1
disp 4
ei
addi $a2 $s0,0
jr $ra
```

- 同步中断代码

```
#AS
```

```
disp ,1
ei
disp ,2
inti ,0
disp ,3
disp ,5
```

- **异步中断**代码

```
if (KEY_DOWN(VK_LBUTTON) || KEY_DOWN(VK_RBUTTON)|| _kbhit()) {
    std::cout << "canInt = " << Code::Reg::canInt << std::endl;
    if (Code::Reg::canInt) {
        Code::Reg::canInt = 0;
        Code::Reg::inta = 1;
        char ch;
        if(KEY_DOWN(VK_LBUTTON) || KEY_DOWN(VK_RBUTTON))
            std::cout << "鼠标按下，进入中断" << std::endl;
        else {
            ch = _getch();
            std::cout << "键盘按下 "<<ch<<"，进入中断" << std::endl;
        }
        std::cout << "**********************************" << std::endl;
        if (Code::Reg::inta == 1 && !Code::Reg::IE)
            Interrupt();//执行中断
    }
}
```

检测到前面有鼠标点击或键盘输入，则异步中断。

- **异步中断**测试汇编代码

```
#AS
ei
addi $t0,$zero,0
addi $t1,$zero,0
addi $a1,$zero,0
addi $a2,$zero,0
addi $a3,$zero,1
add $a1,$a2,$zero
addi $t0,$t0,1
addi $t1,$t1,4
```

简单执行 addi 指令，其中可以进行鼠标或键盘的异步中断

5.3 实际运行截图

- 斐波那契数列运行过程

  命令行执行 JZMid fibonacci.txt：

```
F:\vs2019\docu\Release>JZMid Fibonacci.txt
PC=0
PC->AR AR=0
DR=mem(0)
DR->IR
Command:11000000100000000000000000000000
请输入数列长度n:
3
```

  此时输入数列长度 n=5，继续运行，中间输出指令微操作：

```
PC=72
PC->AR AR=72
DR=mem(72)
DR->IR
Command:00100001001010010000000000000100
Command:addi Reg9, Reg9, 4
Reg9=20
Reg9+4=20 overflow=0,Regwrite=1
Reg9=20

PC=76
PC->AR AR=76
DR=mem(76)
DR->IR
Command:00010100100010001111111111101000
Command:bne Reg4, Reg8, 65512
Reg4=5,Reg8=5
Reg8!=Reg4?
False
```

  最终得到长度为 n 的斐波那契数列：

```
打印存储器中的结果:
Mem[1000]=1
Mem[1004]=1
Mem[1008]=2
Mem[1012]=3
Mem[1016]=5
```

- 冒泡排序运行过程：

  命令行运行 JZMid Sort.txt

```
F:\vs2019\docu\Release>JZMid Sort.txt
PC=0
PC->AR AR=0
DR=mem(0)
DR->IR
Command:11000000100000000000000000000000
请输入数列长度n:
5
```

此时输入数列长度 n，继续运行，中间输出数据微操作：

```
PC=28
PC->AR AR=28
DR=mem(28)
DR->IR
Command:00010100100010001111111111101000
Command:bne Reg4, Reg8, 65512
Reg4=5,Reg8=1
Reg8!=Reg4?
True,PC+4+expand(imm)=12

PC=12
PC->AR AR=12
DR=mem(12)
DR->IR
Command:11000000101000000000000000000000
请输入第2个数字
3
```

输入 5 3 4 1 2，最终得到结果：

```
打印存储器中的结果:
Mem[1000]=1
Mem[1004]=2
Mem[1008]=3
Mem[1012]=4
Mem[1016]=5
```

- **同步中断**

命令行运行 JZMid SynIntTest.txt IntService.txt，将 InService.txt 中的中断服务程序加载到指定地址的内存中，随后执行 SynIntTest.txt 中的程序：

```
F:\vs2019\docu\JZMid\Debug>JZMid SynIntTest.txt IntService.txt
PC=200
PC->AR AR=200
DR=mem(200)
DR->IR
Command:11001000000000000000000000000001
Command:disp
测试中断

打印存储器中的结果:

PC=204
PC->AR AR=204
DR=mem(204)
DR->IR
Command:11001100000000000000000000000000
Command:ei
EI=0

打印存储器中的结果:

PC=208
PC->AR AR=208
DR=mem(208)
DR->IR
Command:11001000000000000000000000000010
Command:disp
中断服务

打印存储器中的结果:

PC=212
PC->AR AR=212
DR=mem(212)
DR->IR
Command:11010000000000000000000000000000
Command:inti 0
Reg(ra)=0
PC = ((PC + 4) & 0xf0000000) | ((0 & 0x03ffffff) << 2)=0
Reg(ra)=216

中断响应结束，进入中断服务
```

程序打印“测试中断”，并将 EI 置为 0 允许中断，同步中断执行 inti 指令，打印“中断响应结束，进入中断服务”，随后进入中断服务程序：

```
PC=0
PC->AR AR=0
DR=mem(0)
DR->IR
Command:00100000101100000000000000000000
Command:addi Reg16, Reg5, 0
Reg5=0
Reg5+0=0 overflow=0,Regwrite=1
Reg16=0

PC=4
PC->AR AR=4
DR=mem(4)
DR->IR
Command:00100000000001010000000000000000
Command:addi Reg5, Reg0, 0
Reg0=0
Reg0+0=0 overflow=0,Regwrite=1
Reg5=0

PC=8
PC->AR AR=8
DR=mem(8)
DR->IR
Command:11001000000000000000000000000100
Command:disp
中断了，成功

PC=12
PC->AR AR=12
DR=mem(12)
DR->IR
Command:11001100000000000000000000000000
Command:ei
EI=1

PC=16
PC->AR AR=16
DR=mem(16)
DR->IR
Command:00100010000001010000000000000000
Command:addi Reg5, Reg16, 0
Reg16=0
Reg16+0=0 overflow=0,Regwrite=1
Reg5=0
```

打印中断服务程序中的操作，首先保存现场（寄存器值，如 a0），输出“中断了，成功”，并关闭中断。

```
PC=20
PC->AR AR=20
DR=mem(20)
DR->IR
Command:00000011111000000000000000001000
Command:jr Reg31
Reg31=216
PC=216

打印存储器中的结果:

PC=216
PC->AR AR=216
DR=mem(216)
DR->IR
Command:11001000000000000000000000000101
Command:disp
测试结束
```

jr 指令跳回主程序，打印“测试结束”。

- **异步中断（附加功能）**

首先命令行执行：JZMid AsyIntTest.txt IntService.txt，将 InService.txt 中的中断服务程序加载到指定地址的内存中，随后执行 AsyIntTest.txt 中的程序

```
F:\vs2019\docu\JZMid\Debug>JZMid AsyIntTest.txt IntService.txt
PC=200
PC->AR AR=200
DR=mem(200)
DR->IR
Command:11001100000000000000000000000000
Command:ei
EI=0

打印存储器中的结果:
```

首先执行 ei 指令允许中断，随后正常执行程序。按下鼠标或键盘后打印事件信息“鼠标按下，进入中断”或“键盘按下 ch，进入中断”，其中 ch 为键盘的按键对应的 char（如字母 c）。

```
键盘按下 c，进入中断
**************************************
中断响应结束，进入中断服务
PC=0
PC->AR AR=0
DR=mem(0)
DR->IR
Command:00100000101100000000000000000000
Command:addi Reg16, Reg5, 0
Reg5=0
Reg5+0=0 overflow=0,Regwrite=1
Reg16=0
```

```
PC=208
PC->AR AR=208
DR=mem(208)
DR->IR
Command:00100000000010010000000000000000
Command:addi Reg9, Reg0, 0
Reg0=0
Reg0+0=0 overflow=0,Regwrite=1
Reg9=0

鼠标按下，进入中断
**************************************
中断响应结束，进入中断服务
PC=0
PC->AR AR=0
DR=mem(0)
DR->IR
Command:00100000101100000000000000000000
Command:addi Reg16, Reg5, 0
Reg5=0
Reg5+0=0 overflow=0,Regwrite=1
Reg16=0

PC=4
PC->AR AR=4
DR=mem(4)
DR->IR
Command:00100000000001010000000000000000
Command:addi Reg5, Reg0, 0
Reg0=0
Reg0+0=0 overflow=0,Regwrite=1
Reg5=0

PC=8
PC->AR AR=8
DR=mem(8)
DR->IR
Command:11001000000000000000000000000100
Command:disp
中断了，成功

PC=12
PC->AR AR=12
DR=mem(12)
DR->IR
Command:11001100000000000000000000000000
Command:ei
EI=1
```

按下键盘或鼠标后，打印“中断响应结束，进入中断程序”，随后进入中断程序执行代码，先保存现场后用 ei 关闭中断，最后 jr 跳回主程序：

```
PC=20
PC->AR AR=20
DR=mem(20)
DR->IR
Command:00000011111000000000000000001000
Command:jr Reg31
Reg31=216
PC=216
```

六、 个人体会

通过本次对 MIPS 指令集的仿真实验，从最开始对 MIPS 的学习，到制作 MIPS 指令集流程图，再到最后对 MIPS 指令的仿真实验和样例测试，以及最后一次的中断服务程序，从中我学会了 MIPS 指令集的格式及分类、指令和操作数寻址方式、所用的寄存器以及中断机制，在实践中对 CPU 的运行机理有了更深入的认识，完成了 MIPS 虚拟机。这次团队合作也让我学会了如何与团队中的成员相处。