

Lab1 仿真实验

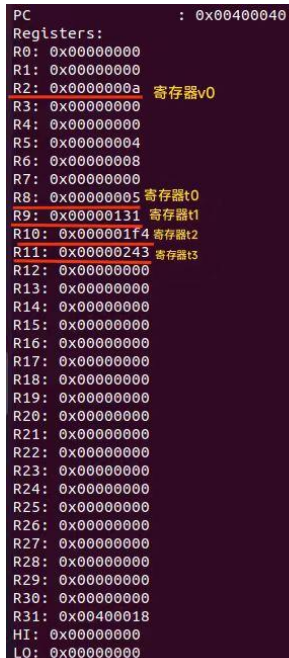
第一部分：实验分析及执行

1.addiu.s

```
.data
.text
.globl main
main:
    addiu $v0, $zero, 10 #将寄存器 v0 的值设置为 10
    addiu $t0, $zero, 5#将寄存器 t0 的值设置为 5
    addiu $t1, $t0, 300#将寄存器 t0 的值和 300 相加，并将结果存入寄存器 t1
    addiu $t2, $zero, 500#将寄存器 t2 的值设置为 500
    addiu $t3, $t2, 34#将寄存器 t2 的值和 34 相加，并将结果存入寄存器 t3
    addiu $t3, $t3, 45#将寄存器 t3 的值和 45 相加，并将结果存入寄存器 t3
    Syscall
```

MIPS-SIM> go

MIPS-SIM> rdump



```
PC : 0x00400040
Registers:
R0: 0x00000000
R1: 0x00000000
R2: 0x0000000a 寄存器v0
R3: 0x00000000
R4: 0x00000000
R5: 0x00000004
R6: 0x00000008
R7: 0x00000000
R8: 0x00000005 寄存器t0
R9: 0x00000131 寄存器t1
R10: 0x000001f4 寄存器t2
R11: 0x00000243 寄存器t3
R12: 0x00000000
R13: 0x00000000
R14: 0x00000000
R15: 0x00000000
R16: 0x00000000
R17: 0x00000000
R18: 0x00000000
R19: 0x00000000
R20: 0x00000000
R21: 0x00000000
R22: 0x00000000
R23: 0x00000000
R24: 0x00000000
R25: 0x00000000
R26: 0x00000000
R27: 0x00000000
R28: 0x00000000
R29: 0x00000000
R30: 0x00000000
R31: 0x00400018
HI: 0x00000000
LO: 0x00000000
```

V0 寄存器十进制为 10，十六进制为 a;

T0 寄存器十进制为 5，十六进制为 5;

V1 寄存器十进制为 305，十六进制为 131;

V2 寄存器十进制为 500，十六进制为 1f4;

V3 寄存器十进制为 579，十六进制为 243;

在 src/ 中，有模拟器文件和用于编译它们的 Makefile，inputs/ 包含用于测试模拟器的输入文件，使用 Qtspim 模拟器来将它们转换为十六进制代码。

```
$ cd src/
```

```
$ make
```

使用 Qtspim 模拟器来将它们转换为十六进制代码.x

```
src/sim inputs/hex/addiu.x
```

2.arithtest.s(基本算术指令测试)src/sim inputs/hex/arithtest.x

```
# Basic arithmetic instructions
```

```
# This is a hodgepodge of arithmetic instructions to test
```

```
# your basic functionality.
```

```
# No overflow exceptions should occur
```

```
.data
```

```
.text
```

```
.globl main
```

```
main:
```

addiu	\$2, \$zero, 1024	# 将寄存器\$2 设置为 1024
addu	\$3, \$2, \$2	# 将寄存器\$3 设置为\$2 的两倍
or	\$4, \$3, \$2	# 将寄存器\$4 设置为\$3 和\$2 的按位或结果
add	\$5, \$zero, 1234	# 将寄存器\$5 设置为 1234
sll	\$6, \$5, 16	# 将寄存器\$6 设置为\$5 左移 16 位的结果
addiu	\$7, \$6, 9999	# 将寄存器\$7 设置为\$6 加 9999 的结果
subu	\$8, \$7, \$2	# 将寄存器\$8 设置为\$7 减去\$2 的结果
xor	\$9, \$4, \$3	# 将寄存器\$9 设置为\$4 和\$3 的按位异或结果
xori	\$10, \$2, 255	# 将寄存器\$10 设置为\$2 和 255 的按位异或结果
srl	\$11, \$6, 5	# 将寄存器\$11 设置为\$6 右移 5 位的结果
sra	\$12, \$6, 4	# 将寄存器\$12 设置为\$6 算术右移 4 位的结果
and	\$13, \$11, \$5	# 将寄存器\$13 设置为\$11 和\$5 的按位与结果
andi	\$14, \$4, 100	# 将寄存器\$14 设置为\$4 和 100 的按位与结果
sub	\$15, \$zero, \$10	# 将寄存器\$15 设置为 0 减去\$10 的结果
lui	\$17, 100	# 将寄存器\$17 的高 16 位设置为 100
addiu	\$v0, \$zero, 0xa	# 将寄存器\$v0 设置为 10
syscall		# 执行系统调用

```
for target=vc600-machine: /1010/0x/instruction-level-mips-32
MIPS Simulator
Read 26 words from program into memory.
MIPS-SIM> go
Simulating...
Simulator halted
MIPS-SIM> rdump
Current register/bus values :
-----
Instruction Count : 23
PC : 0x00400068
Registers:
R0: 0x00000000
R1: 0x00000000
R2: 0x0000000a
R3: 0x00000000
R4: 0x000000c0
R5: 0x000004d2
R6: 0x04d20000
R7: 0x04d2270f
R8: 0x04d2230f
R9: 0x00000400
R10: 0x000004ff
R11: 0x00269000
R12: 0x004d2000
R13: 0x00000000
R14: 0x00000000
R15: 0xfffffb01
R16: 0x00000000
R17: 0x00640000
R18: 0x00000000
R19: 0x00000000
R20: 0x00000000
R21: 0x00000000
```

3.Brtest0.s(分支跳转指令测试) src/sim inputs/hex/brtest0.x

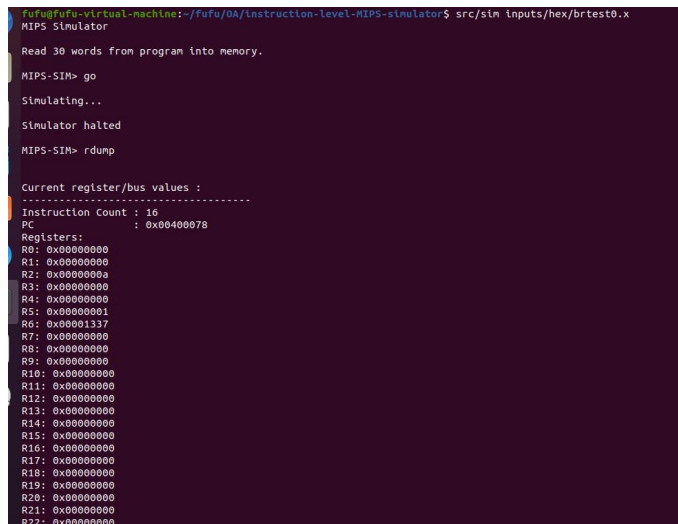
```
# Basic branch test

.data
.text
.globl main
main:
    addiu $v0, $zero, 0xa    # 将寄存器$v0 的值设置为 10
l_0:
    addiu $5, $zero, 1      # 将寄存器$5 的值设置为 1
    j l_1                  # 无条件跳转到标签 l_1 处
    addiu $10, $10, 0xf00    # 将寄存器$10 的值加上 0xf00
    ori $0, $0, 0           # 将寄存器$0 的值与 0 相或, 结果仍为 0
    ori $0, $0, 0           # 将寄存器$0 的值与 0 相或, 结果仍为 0
    addiu $5, $zero, 100    # 将寄存器$5 的值设置为 100
    syscall                # 系统调用
l_1:
    bne $zero, $zero, l_3   # 如果$zero 不等于$zero, 则跳转到标签 l_3 处 (永远不会发生)
    ori $0, $0, 0           # 将寄存器$0 的值与 0 相或, 结果仍为 0
    ori $0, $0, 0           # 将寄存器$0 的值与 0 相或, 结果仍为 0
    addiu $6, $zero, 0x1337 # 将寄存器$6 的值设置为 0x1337
l_2:
    beq $zero, $zero, l_4   # 如果$zero 等于$zero, 则跳转到标签 l_4 处 (永远会发生)
    ori $0, $0, 0           # 将寄存器$0 的值与 0 相或, 结果仍为 0
    ori $0, $0, 0           # 将寄存器$0 的值与 0 相或, 结果仍为 0
    # 不应该到达这里
```

```

addiu $7, $zero, 0x347 # 将寄存器$7 的值设置为 0x347
syscall                # 系统调用
l_3:
# 不应该到达这里
addiu $8, $zero, 0x347 # 将寄存器$8 的值设置为 0x347
syscall                # 系统调用
l_4:
addiu $2, $zero, 10    # 将寄存器$2 的值设置为 10
syscall                # 系统调用

```



```

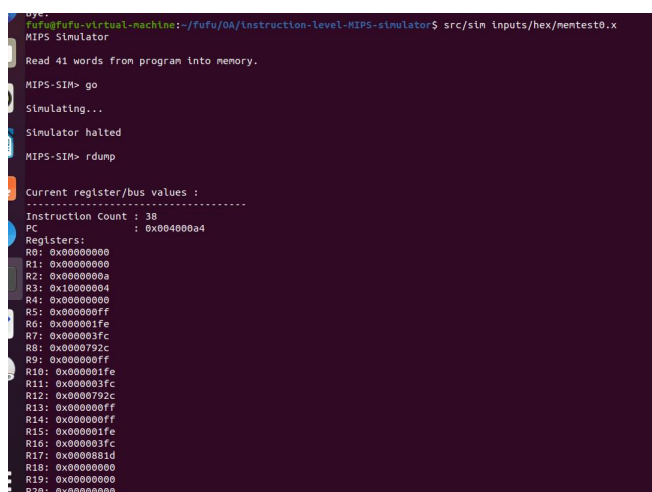
fufu@fufu-virtual-machine:~/fufu/0A/instruction-level-MIPS-simulator$ src/sim inputs/hex/brtest0.x
MIPS Simulator
Read 30 words from program into memory.
MIPS-SIM> go
Simulating...
Simulator halted
MIPS-SIM> rdump

Current register/bus values :
-----
Instruction Count : 16
PC                : 0x00400078
Registers:
R0: 0x00000000
R1: 0x00000000
R2: 0x0000000a
R3: 0x00000000
R4: 0x00000000
R5: 0x00000001
R6: 0x00001337
R7: 0x00000000
R8: 0x00000000
R9: 0x00000000
R10: 0x00000000
R11: 0x00000000
R12: 0x00000000
R13: 0x00000000
R14: 0x00000000
R15: 0x00000000
R16: 0x00000000
R17: 0x00000000
R18: 0x00000000
R19: 0x00000000
R20: 0x00000000
R21: 0x00000000
R22: 0x00000000

```

同理我们分析 src/sim inputs/hex/brtest1.x, src/sim inputs/hex/brtest2.x。

4.Load/Store 测试 (src/sim inputs/hex/memtest0.x)



```

fufu@fufu-virtual-machine:~/fufu/0A/instruction-level-MIPS-simulator$ src/sim inputs/hex/memtest0.x
MIPS Simulator
Read 41 words from program into memory.
MIPS-SIM> go
Simulating...
Simulator halted
MIPS-SIM> rdump

Current register/bus values :
-----
Instruction Count : 38
PC                : 0x004000a4
Registers:
R0: 0x00000000
R1: 0x00000000
R2: 0x0000000a
R3: 0x10000004
R4: 0x00000000
R5: 0x000000ff
R6: 0x000001fe
R7: 0x000003fc
R8: 0x0000792c
R9: 0x000000ff
R10: 0x000001fe
R11: 0x000003fc
R12: 0x0000792c
R13: 0x000000ff
R14: 0x000000ff
R15: 0x000001fe
R16: 0x000003fc
R17: 0x0000881d
R18: 0x00000000
R19: 0x00000000
R20: 0x00000000

```

同理分析 src/sim inputs/hex/memtest1.x。

第二部分：思路及代码

1. 要求：

在 `sim.c` 文件中实现 `process_instruction()` 函数。`process_instruction()` 函数应能够模拟以下 MIPS 指令的指令级执行：

- `add`: 将两个寄存器的值相加并将结果存入目标寄存器。
- `sub`: 将两个寄存器的值相减并将结果存入目标寄存器。
- `lw`: 从内存中加载一个字（4 个字节）到寄存器中。
- `sw`: 将寄存器的值存入内存中。
- `beq`: 如果两个寄存器的值相等，则跳转到指定的地址。
- `j`: 无条件跳转到指定的地址。
-

在 `process_instruction()` 函数中实现这些指令的模拟执行，编写代码来模拟这些指令的操作，并根据指令的操作码和操作数执行相应的操作。

如对于 `add` 指令，从指令中提取操作数和目标寄存器的编号，然后将两个操作数相加，并将结果存入目标寄存器；`lw` 指令，从指令中提取内存地址和目标寄存器的编号，然后从内存中加载相应的数据，并将其存入目标寄存器。

J	JAL	BEQ	BNE	BLEZ	BGTZ
ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI
XORI	LUI	LB	LH	LW	LBU
LHU	SB	SH	SW	BLTZ	BGEZ
BLTZAL	BGEZAL	SLL	SRL	SRA	SLLV
SRLV	SRAV	JR	JALR	ADD	ADDU
SUB	SUBU	AND	OR	XOR	NOR
SLT	SLTU	MULT	MFHI	MFLO	MTHI
MTLO	MULTU	DIV	DIVU	SYSCALL	

2.Shell 命令

1、`go`: 模拟程序执行，直到遇到一个 `SYSCALL` 指令，并且该指令中的 `$v0` 寄存器（寄存器 2）的值等于 `0x0A`，此时模拟器将停止运行。

2、`run <n>`: 模拟器执行 `n` 条指令。

3、`mdump <low> <high>`: 将内存中从低地址 `low` 到高地址 `high` 的内容输出到屏幕和 `dumpsim` 文件。

- 4、rdump: 将当前指令计数器、R0-R31 寄存器的内容以及 PC 寄存器的值输出到屏幕和 dumsim 文件。
- 5、input reg_num reg_val: 将通用寄存器 reg_num 的值设置为 reg_val。
- 6、high value: 将 HI 寄存器的值设置为 value。
- 7、low value: 将 LO 寄存器的值设置为 value。
- 8、?: 打印出所有 shell 命令的列表。
- 9、quit: 退出 shell。

3.Sim.c

①首先，它将从内存中读取一个 32 位的指令，并将其存储在变量 instruction 中。然后，它将从 instruction 中提取出操作码（op）、源寄存器（rs）、目标寄存器（rt）、目的寄存器（rd）、移位量（shamt）、函数码（funct）、立即数（imm）和目标地址（target）。

接下来，根据不同的操作码，程序会执行不同的操作。如果指令为 0，即 nop 指令，那么将执行 nop 函数。如果操作码为 0，表示为 R-Type 指令，将执行 rtype 函数，参数为 rs、rt、rd、shamt 和 funct。如果操作码为 1，表示为特殊分支指令，将执行 itype_branches 函数，参数为 rs、rt 和 imm。如果操作码为 2 或 3，表示为 J-Type 指令，将执行 jtype 函数，参数为 op 和 target。否则，将执行 itype 函数，参数为 op、rs、rt 和 imm。

最后，程序会将 NEXT_STATE.PC 增加 4，以便执行下一条指令。

void process_instruction() {

```
uint32_t instruction = 0x0;
instruction = mem_read_32(CURRENT_STATE.PC);
uint8_t op = 0;
uint8_t rs = 0, rt = 0, rd = 0, shamt = 0, funct = 0;
uint16_t imm = 0;
uint32_t target = 0;
op = ((instruction >> 26) & 0x3F);
rs = ((instruction >> 21) & 0x1F);
rt = ((instruction >> 16) & 0x1F);
rd = ((instruction >> 11) & 0x1F);
shamt = ((instruction >> 6) & 0x1F);
funct = (instruction & 0x3F);
imm = instruction;
target = instruction & 0x03FFFFFF;

if(instruction == 0)
```

```

        nop();
    else if(op == 0) // R-Type Instructions.
        rtype(rs, rt, rd, shamt, funct);
    else if(op == 1) // Special Branches.
        itype_sbranches(rs, rt, imm);
    else if(op == 2 || op == 3)
        jtype(op, target);
    else
        itype(op, rs, rt, imm);

    NEXT_STATE.PC += 4;
}

```

②函数名为 `rtype`，参数类型为 `uint8_t`，参数名分别为 `rs`、`rt`、`rd`、`shamt` 和 `funct`。

该函数根据 `funct` 的值调用不同的子函数来执行不同的操作。具体如下：

如果 `funct` 小于等于 7，则调用 `shifts` 函数，参数为 `rt`、`rd`、`shamt` 和 `funct`。

如果 `funct` 的值在 24 到 27 之间，或者在 32 到 35 之间，则调用 `arithmetic` 函数，参数为 `rs`、`rt`、`rd` 和 `funct`。

如果 `funct` 的值在 36 到 39 之间，则调用 `logical` 函数，参数为 `rs`、`rt`、`rd` 和 `funct`。

如果 `funct` 的值等于 42 或 43，则调用 `conditional` 函数，参数为 `rs`、`rt`、`rd` 和 `funct`。

如果 `funct` 的值等于 8 或 9，则调用 `rtype_jump` 函数，参数为 `rs`、`rd` 和 `funct`。

如果 `funct` 的值在 16 到 19 之间，则调用 `rtype_lo_hi` 函数，参数为 `rs`、`rd` 和 `funct`。

如果 `funct` 的值等于 12，则调用 `syscall` 函数。

如果以上条件都不满足，则什么也不做。

void rtype(uint8_t rs, uint8_t rt, uint8_t rd, uint8_t shamt, uint8_t funct) {

```

    if(funct <= 7)

        shifts(rt, rd, shamt, funct);

    else if( (funct >= 24 && funct <= 27) || (funct >= 32 && funct <= 35) )

        arithmetic(rs, rt, rd, funct);

    else if(funct >= 36 && funct <= 39)

        logical(rs, rt, rd, funct);

```

```

        else if(funcnt == 42 || funcnt == 43)

            conditional(rs, rt, rd, funcnt);

        else if(funcnt == 8 || funcnt == 9)

            rtype_jump(rs, rd, funcnt);

        else if(funcnt >= 16  && funcnt <= 19)

            rtype_lo_hi(rs, rd, funcnt);

        else if(funcnt == 12)

            syscall();

        else

            ;

    }

```

③定义了一个名为 **itype** 的函数，该函数接受四个参数：**op**、**rs**、**rt** 和 **imm**，其中 **op** 和 **rt** 的类型是 **uint8_t**，**imm** 的类型是 **int16_t**。

函数根据 **op** 的值来执行不同的操作：

如果 **op** 的值在 4 到 7 之间，则调用 **itype_branches** 函数，传递参数 **op**、**rs**、**rt** 和 **imm**。

如果 **op** 的值在 8 到 15 之间，则调用 **itype_arithmetic** 函数，传递参数 **op**、**rs**、**rt** 和 **imm**。

如果 **op** 的值在 32 到 43 之间，则调用 **mem_load_store** 函数，传递参数 **op**、**rs**、**rt** 和 **imm**。

如果以上条件都不满足，则什么也不做。

```

void itype(uint8_t op, uint8_t rs, uint8_t rt, int16_t imm) {

```

```

    if(op >= 4 && op <= 7)

        itype_branches(op, rs, rt, imm);

    else if(op >= 8 && op <= 15)

        itype_arithmetic(op, rs, rt, imm);

    else if(op >= 32 && op <= 43)

        mem_load_store(op, rs, rt, imm);

    else

        ;

}

```

④定义了一个名为 **jtype** 的函数，该函数接受两个参数：**op** 和 **target**，其中 **op** 的类型是 **uint8_t**，**target** 的类型是 **uint32_t**。

函数根据 **op** 的值来执行不同的操作：

如果 **op** 的值等于 **J**，则调用 **j** 函数，传递参数 **target**，并立即返回。

如果 `op` 的值等于 `JAL`，则调用 `jal` 函数，传递参数 `target`，并立即返回。

```
void jtype(uint8_t op, uint32_t target) {
```

```
    switch(op) {
        case J:
            j(target);
            return;
        case JAL:
            jal(target);
            return;
    }
}
```

⑤定义了一个名为 `shifts` 的函数，该函数接受四个参数：`rt`、`rd`、`shamt` 和 `funct`，其中 `rt`、`rd`、`shamt` 和 `funct` 的类型都是 `uint8_t`。

函数根据 `funct` 的值来执行不同的操作：

如果 `funct` 的值等于 `SLL`，则调用 `sll` 函数，传递参数 `rt`、`rd` 和 `shamt`，并立即返回。

如果 `funct` 的值等于 `SRL`，则调用 `srl` 函数，传递参数 `rt`、`rd` 和 `shamt`，并立即返回。

如果 `funct` 的值等于 `SRA`，则调用 `sra` 函数，传递参数 `rt`、`rd` 和 `shamt`，并立即返回。

如果 `funct` 的值等于 `SLLV`，则调用 `sllv` 函数，传递参数 `rt`、`rd` 和 `shamt`，并立即返回。

如果 `funct` 的值等于 `SRLV`，则调用 `srlv` 函数，传递参数 `rt`、`rd` 和 `shamt`，并立即返回。

如果 `funct` 的值等于 `SRAV`，则调用 `srav` 函数，传递参数 `rt`、`rd` 和 `shamt`，并立即返回。

```
void shifts(uint8_t rt, uint8_t rd, uint8_t shamt, uint8_t funct) {
```

```
    switch(funct) {
        case SLL:
            sll(rt, rd, shamt);
            return;
        case SRL:
            srl(rt, rd, shamt);
            return;
        case SRA:
            sra(rt, rd, shamt);
            return;
        case SLLV:
            sllv(rt, rd, shamt);
            return;
        case SRLV:
            srlv(rt, rd, shamt);
            return;
        case SRAV:
            srav(rt, rd, shamt);
            return;
    }
}
```

⑥定义了一个名为 `arithmetic` 的函数，该函数接受四个参数：`rs`、`rt`、`rd` 和 `funct`，其中 `rs`、`rt`、`rd` 和 `funct` 的类型都是 `uint8_t`。

函数根据 `funct` 的值来执行不同的操作：

如果 `funct` 的值等于 `ADD`，则调用 `add` 函数，传递参数 `rs`、`rt` 和 `rd`，并立即返回。

如果 `funct` 的值等于 `ADDU`，则调用 `addu` 函数，传递参数 `rs`、`rt` 和 `rd`，并立即返回。

如果 `funct` 的值等于 `SUB`，则调用 `sub` 函数，传递参数 `rs`、`rt` 和 `rd`，并立即返回。

如果 `funct` 的值等于 `SUBU`，则调用 `subu` 函数，传递参数 `rs`、`rt` 和 `rd`，并立即返回。

如果 `funct` 的值等于 `MULT`，则调用 `mult` 函数，传递参数 `rs` 和 `rt`，并立即返回。

如果 `funct` 的值等于 `MULTU`，则调用 `multu` 函数，传递参数 `rs` 和 `rt`，并立即返回。

如果 `funct` 的值等于 `DIV`，则调用 `div` 函数，传递参数 `rs` 和 `rt`，并立即返回。

如果 `funct` 的值等于 `DIVU`，则调用 `divu` 函数，传递参数 `rs` 和 `rt`，并立即返回。

```
void arithmetic(uint8_t rs, uint8_t rt, uint8_t rd, uint8_t funct) {
```

```
    switch(funct) {
        case ADD:
            add(rs, rt, rd);
            return;
        case ADDU:
            addu(rs, rt, rd);
            return;
        case SUB:
            sub(rs, rt, rd);
            return;
        case SUBU:
            subu(rs, rt, rd);
            return;
        case MULT:
            mult(rs, rt);
            return;
        case MULTU:
            multu(rs, rt);
            return;
        case DIV:
            div(rs, rt);
            return;
        case DIVU:
            divu(rs, rt);
            return;
    }
```

```
}
```

⑦定义了一个名为 `logical` 的函数，该函数接受四个参数：`rs`、`rt`、`rd` 和 `funct`，其中 `rs`、`rt`、`rd` 和 `funct` 的类型都是 `uint8_t`。

函数根据 `funct` 的值来执行不同的操作：

如果 `funct` 的值等于 `AND`，则调用 `and` 函数，传递参数 `rs`、`rt` 和 `rd`，并立即返回。

如果 `funct` 的值等于 `OR`，则调用 `or` 函数，传递参数 `rs`、`rt` 和 `rd`，并立即返回。

如果 `funct` 的值等于 `XOR`，则调用 `xor` 函数，传递参数 `rs`、`rt` 和 `rd`，并立即返回。

如果 `funct` 的值等于 `NOR`，则调用 `nor` 函数，传递参数 `rs`、`rt` 和 `rd`，并立即返回。

void logical(uint8_t rs, uint8_t rt, uint8_t rd, uint8_t funct) {

```
    switch(funct) {
        case AND:
            and(rs, rt, rd);
            return;
        case OR:
            or(rs, rt, rd);
            return;
        case XOR:
            xor(rs, rt, rd);
            return;
        case NOR:
            nor(rs, rt, rd);
            return;
    }
}
```

⑧定义了一个名为 `conditional` 的函数，该函数接受四个参数：`rs`、`rt`、`rd` 和 `funct`，其中 `rs`、`rt`、`rd` 和 `funct` 的类型都是 `uint8_t`。

函数根据 `funct` 的值来执行不同的操作：

如果 `funct` 的值等于 `SLT`，则调用 `slt` 函数，传递参数 `rs`、`rt` 和 `rd`，并立即返回。

如果 `funct` 的值等于 `SLTU`，则调用 `sltu` 函数，传递参数 `rs`、`rt` 和 `rd`，并立即返回。

void conditional(uint8_t rs, uint8_t rt, uint8_t rd, uint8_t funct) {

```
    switch(funct) {
        case SLT:
            slt(rs, rt, rd);
            return;
        case SLTU:
            sltu(rs, rt, rd);
            return;
    }
}
```

⑨定义了一个名为 `rtype_jump` 的函数，该函数接受三个参数：`rs`、`rd` 和 `funct`，其中 `rs` 和 `rd` 的类型都是 `uint8_t`，而 `funct` 的类型也是 `uint8_t`。

函数根据 `funct` 的值来执行不同的操作：

如果 `funct` 的值等于 `JR`，则调用 `jr` 函数，传递参数 `rs`，并立即返回。

如果 `funct` 的值等于 `JALR`，则调用 `jalr` 函数，传递参数 `rs` 和 `rd`，并立即返回。

```
void rtype_jump(uint8_t rs, uint8_t rd, uint8_t funct) {
```

```
    switch(funct) {
        case JR:
            jr(rs);
            return;
        case JALR:
            jalr(rs, rd);
            return;
    }
}
```

⑩定义了一个名为 `rtype_lo_hi` 的函数，该函数接受三个参数：`rs`、`rd` 和 `funct`，其中 `rs` 和 `rd` 的类型都是 `uint8_t`，而 `funct` 的类型也是 `uint8_t`。

函数根据 `funct` 的值来执行不同的操作：

如果 `funct` 的值等于 `MFHI`，则调用 `mfhi` 函数，传递参数 `rd`，并立即返回。

如果 `funct` 的值等于 `MFLO`，则调用 `mflo` 函数，传递参数 `rd`，并立即返回。

如果 `funct` 的值等于 `MTHI`，则调用 `mthi` 函数，传递参数 `rs`，并立即返回。

如果 `funct` 的值等于 `MTLO`，则调用 `mtlo` 函数，传递参数 `rs`，并立即返回。

```
void rtype_lo_hi(uint8_t rs, uint8_t rd, uint8_t funct) {
```

```
    switch(funct) {
        case MFHI:
            mfhi(rd);
            return;
        case MFLO:
            mflo(rd);
            return;
        case MTHI:
            mthi(rs);
            return;
        case MTLO:
            mtlo(rs);
            return;
    }
}
```

11.定义了一个名为 `itype_sbranches` 的函数，该函数接受三个参数：`rs`、`rt` 和 `imm`，其中 `rs` 和 `rt` 的类型都是 `uint8_t`，而 `imm` 的类型是 `int16_t`。

函数根据 `rt` 的值来执行不同的操作：

如果 `rt` 的值等于 `BLTZ`，则调用 `bltz` 函数，传递参数 `rs` 和 `imm`，并立即返回。

如果 `rt` 的值等于 `BGEZ`，则调用 `bgez` 函数，传递参数 `rs` 和 `imm`，并立即返回。

如果 `rt` 的值等于 `BLTZAL`，则调用 `bltzal` 函数，传递参数 `rs` 和 `imm`，并立即返回。

如果 `rt` 的值等于 `BGEZAL`，则调用 `bgezal` 函数，传递参数 `rs` 和 `imm`，并立即返回。

```
void itype_sbranches(uint8_t rs, uint8_t rt, int16_t imm) {
```

```
    switch(rt) {
        case BLTZ:
            bltz(rs, imm);
            return;
        case BGEZ:
            bgez(rs, imm);
            return;
        case BLTZAL:
            bltzal(rs, imm);
            return;
        case BGEZAL:
            bgezal(rs, imm);
            return;
    }
}
```

12. 定义了一个名为 `itype_arithmetic` 的函数，该函数接受四个参数：`op`、`rs`、`rt` 和 `imm`，其中 `op`、`rs` 和 `rt` 的类型都是 `uint8_t`，而 `imm` 的类型是 `int16_t`。

函数根据 `op` 的值来执行不同的操作：

如果 `op` 的值等于 `ADDI`，则调用 `addi` 函数，传递参数 `rs`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `ADDIU`，则调用 `addiu` 函数，传递参数 `rs`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `SLTI`，则调用 `slti` 函数，传递参数 `rs`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `SLTIU`，则调用 `sltiu` 函数，传递参数 `rs`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `ANDI`，则调用 `andi` 函数，传递参数 `rs`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `ORI`，则调用 `ori` 函数，传递参数 `rs`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `XORI`，则调用 `xori` 函数，传递参数 `rs`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `LUI`，则调用 `lui` 函数，传递参数 `rt` 和 `imm`，并立即返回。

```
void itype_arithmetic(uint8_t op, uint8_t rs, uint8_t rt, int16_t imm) {
```

```
    switch(op) {
        case ADDI:
            addi(rs, rt, imm);
            return;
        case ADDIU:
            addiu(rs, rt, imm);
            return;
        case SLTI:
            slti(rs, rt, imm);
            return;
```

```

        case SLTIU:
            sltiu(rs, rt, imm);
            return;
        case ANDI:
            andi(rs, rt, imm);
            return;
        case ORI:
            ori(rs, rt, imm);
            return;
        case XORI:
            xori(rs, rt, imm);
            return;
        case LUI:
            lui(rt, imm);
            return;
    }
}

```

13.定义了一个名为 `itype_branches` 的函数，该函数接受四个参数：`op`、`rs`、`rt` 和 `imm`，其中 `op`、`rs` 和 `rt` 的类型都是 `uint8_t`，而 `imm` 的类型是 `int16_t`。

函数根据 `op` 的值来执行不同的操作：

如果 `op` 的值等于 `BEQ`，则调用 `beq` 函数，传递参数 `rs`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `BNE`，则调用 `bne` 函数，传递参数 `rs`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `BLEZ`，则调用 `blez` 函数，传递参数 `rs` 和 `imm`，并立即返回。

如果 `op` 的值等于 `BGTZ`，则调用 `bgtz` 函数，传递参数 `rs` 和 `imm`，并立即返回。

```

void itype_branches(uint8_t op, uint8_t rs, uint8_t rt, int16_t imm) {

```

```

    switch(op) {
        case BEQ:
            beq(rs, rt, imm);
            return;
        case BNE:
            bne(rs, rt, imm);
            return;
        case BLEZ:
            blez(rs, imm);
            return;
        case BGTZ:
            bgtz(rs, imm);
            return;
    }
}

```

14.定义了一个名为 `mem_load_store` 的函数，该函数接受四个参数：`op`、`base`、`rt` 和 `imm`，其中 `op`、`base` 和 `rt` 的类型都是 `uint8_t`，而 `imm` 的类型是 `int16_t`。

函数根据 `op` 的值来执行不同的操作：

如果 `op` 的值等于 `LB`，则调用 `lb` 函数，传递参数 `base`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `LH`，则调用 `lh` 函数，传递参数 `base`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `LW`，则调用 `lw` 函数，传递参数 `base`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `LBU`，则调用 `lbu` 函数，传递参数 `base`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `LHU`，则调用 `lhu` 函数，传递参数 `base`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `SB`，则调用 `sb` 函数，传递参数 `base`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `SH`，则调用 `sh` 函数，传递参数 `base`、`rt` 和 `imm`，并立即返回。

如果 `op` 的值等于 `SW`，则调用 `sw` 函数，传递参数 `base`、`rt` 和 `imm`，并立即返回。

```
void mem_load_store(uint8_t op, uint8_t base, uint8_t rt, int16_t imm) {
```

```
    switch(op) {
        case LB:
            lb(base, rt, imm);
            return;
        case LH:
            lh(base, rt, imm);
            return;
        case LW:
            lw(base, rt, imm);
            return;
        case LBU:
            lbu(base, rt, imm);
            return;
        case LHU:
            lhu(base, rt, imm);
            return;
        case SB:
            sb(base, rt, imm);
            return;
        case SH:
            sh(base, rt, imm);
            return;
        case SW:
            sw(base, rt, imm);
            return;
    }
```

```
}
```