

体系结构仿真实验一

——2114042 林子淳

代码: [Keven-Lzc/arch \(github.com\)](https://github.com/Keven-Lzc/arch)

一、预备工作

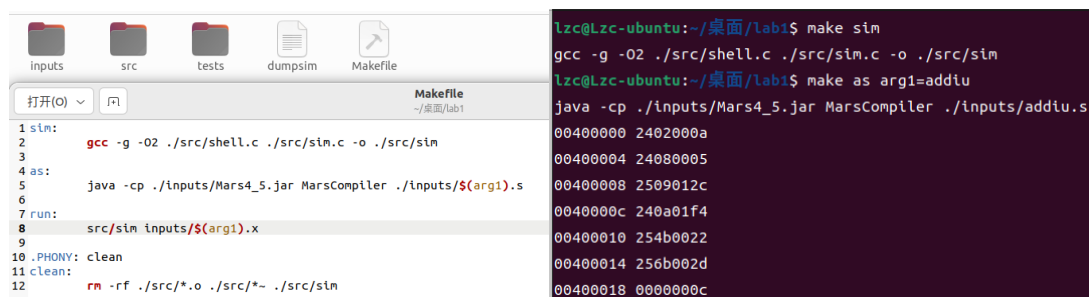
J	JAL	BEQ	BNE	BLEZ	BGTZ
ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI
XORI	LUI	LB	LH	LW	LBU
LHU	SB	SH	SW	BLTZ	BGEZ
BLTZAL	BGEZAL	SLL	SRL	SRA	SLLV
SRLV	SRAV	JR	JALR	ADD	DDU
SUB	SUBU	AND	OR	XOR	NOR
SLT	SLTU	MULT	MFHI	MFLO	MTHI
MTLO	MULTU	DIV	DIVU		SYSCALL

本次实验要实现的是一个指令集 MIPS 模拟器，涉及到的指令及分类如上图所示，具体实现和解释见下文代码部分。

相关资料内已经帮我们实现好了 shell，我们需要完成两个工作：将 mips 指令翻译成汇编码；设计 sim.c 程序，其可以通过读入机器码，翻译并模拟出 mips 指令在寄存器以及内存层面上的调度。除此以外，实验还要求我们自行设计覆盖全部指令的 mips 代码。

二、翻译工作

指导手册内推荐使用的是 spim，但由于其会出现一些报错（版本问题），查找资料后我最终选择在网上下载了 mars，其作为一个将 mips 指令翻译为机器码的插件，在配置好环境后，可以通过 `java -cp Mars4_5.jar MarsCompiler [文件名]` 指令输出对应机器码。



```
lzc@Lzc-ubuntu:~/桌面/lab1$ make sim
gcc -g -O2 ./src/shell.c ./src/sim.c -o ./src/sim
lzc@Lzc-ubuntu:~/桌面/lab1$ make as arg1=addiu
java -cp ./inputs/Mars4_5.jar MarsCompiler ./inputs/addiu.s
00400000 2402000a
00400004 24080005
00400008 2509012c
0040000c 240a01f4
00400010 254b0022
00400014 256b002d
00400018 0000000c
```

对应完善相关 makefile，就可以简单的完成 mips 指令到汇编码的翻译。

```
lzc@Lzc-ubuntu:~/桌面/lab1$ make run arg1=addiu
src/sim inputs/addiu.x
MIPS Simulator

Read 14 words from program into memory.

MIPS-SIM> rdump

Current register/bus values :
-----
Instruction Count : 0
PC                : 0x00400000
Registers:
R0: 0x00000000
```

根据 makefile 的设计，输入 make run arg1[文件名]就可以启动 shell 开始工作，就此我们只需要再完成 sim.c 的代码设计。

三、代码设计

Sim.c 需要完成的工作有三步：使用 mem_read_32 从内存中取出指令，完成机器码的读入和识别；根据识别的结果判断指令的功能并实现对应的操作；更新 PC 的值，使其能够取出下一条指令。

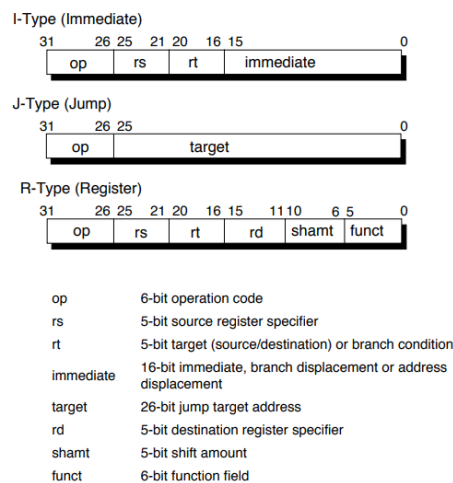


Figure A-1 CPU Instruction Formats

我们需要先了解机器码的组成，才能更好的完善读入工作。

```
uint32_t inst = mem_read_32(CURRENT_STATE.PC);

printf("Instruction: 0x%08x\n", inst);

// 根据机器码获得对应值
uint32_t op = inst >> 26;
uint32_t rs = (inst >> 21) & 0x1f;
uint32_t rt = (inst >> 16) & 0x1f;
uint32_t imm = inst & 0xffff;
uint32_t rd = (inst >> 11) & 0x1f;
uint32_t shamt = (inst >> 6) & 0x1f;
uint32_t funct = inst & 0x3f;
```

我们要做的第一步就是用 mem_read_32 函数获取该行机器码，随后分割出它们 op,rs 等对应的值。

```
#define OP_RTYPE 0x0
#define OP_ADDI 0x8
#define OP_ADDIU 0x9
#define OP_ANDI 0xc
#define OP_SLTI 0x0A
#define OP_SLTIU 0x0B
#define OP_ORI 0xd
#define OP_XORI 0xe
#define OP_BEQ 0x4
#define OP_BNE 0x5
#define OP_BLEZ 0x6
#define OP_BGTZ 0x7
#define OP_J 0x2
#define OP_JAL 0x3
#define OP_LUI 0xf

#define FUNCT_SLL 0x00
#define FUNCT_SRL 0x02
#define FUNCT_SRA 0x03
#define FUNCT_SLLV 0x04
#define FUNCT_SRLV 0x06
#define FUNCT_SRAV 0x07
#define FUNCT_JR 0x08
#define FUNCT_JALR 0x09
#define FUNCT_SYSCALL 0x0c
#define FUNCT_MFHI 0x10
#define FUNCT_MTHI 0x11
#define FUNCT_MFLO 0x12
#define FUNCT_MTLO 0x13
#define FUNCT_MULT 0x18
#define FUNCT_MULTU 0x19
#define FUNCT_DIV 0x1a

#define OP_BSPECIAL 0x01
#define RT_BLTZ 0x00
#define RT_BLTZAL 0x10
#define RT_BGEZ 0x01
#define RT_BGEZAL 0x11
```

为识别指令，我们考虑与其有关的三个值 op，funct 以及 rt，并通过查找 mips 有关规范，将不同指令类型且需要用到的对应值进行重定义，方便下面代码的阅读。

//先根据 op 字段区分

switch (op)

{

// R 型指令

case OP_RTYPE:

// 再根据 funct 字段区分

switch (funct)

{

case FUNCT_SLL:

// 省略

default:

}

// 其他 I 型和 J 型指令

case OP_J:

// 省略

// 特殊的跳转指令

case OP_BSPECIAL:

// 根据 rt 字段区分

switch (rt)

{

case RT_BLTZ:

case RT_BGEZ:

case RT_BLTZAL:

case RT_BGEZAL:

default:

}

```
default:
}
```

我们可以简化其流程，首先先识别 op 字段，其中我们可以直接识别出大部分的 i 型和 j 型指令，但存在两类特殊：对于 r 型指令，我们需要额外使用 funct 来区分，同时对于四个特殊的 j 型指令，我们需要 rt 进行进一步区分。

```
uint32_t sign_ext(uint32_t imm) {
    int32_t signed_imm = *((int16_t*)&imm);
    uint32_t extended_imm = *((uint32_t*)&signed_imm);
    return extended_imm;
}

uint32_t sign_ext_byte(uint8_t imm) {
    int32_t signed_imm = *((int8_t*)&imm);
    uint32_t extended_imm = *((uint32_t*)&signed_imm);
    return extended_imm;
}

uint32_t sign_ext_half(uint16_t imm) {
    int32_t signed_imm = *((int16_t*)&imm);
    uint32_t extended_imm = *((uint32_t*)&signed_imm);
    return extended_imm;
}

uint32_t zero_ext(uint32_t imm) { return imm; }
uint32_t zero_ext_byte(uint8_t imm) { return imm; }
uint32_t zero_ext_half(uint16_t imm) { return imm; }
```

除此以外，函数还需要我们进行一些立即数和零的符号扩展和 0 扩展，以及位数转换，我们需要额外设计一些函数进行调用。

```
case FUNCT_SLL: {
    NEXT_STATE.REGS[rd] = CURRENT_STATE.REGS[rt] << shamt;
    NEXT_STATE.PC = CURRENT_STATE.PC + 4;
    break;
}
```

Case 语句内部需要我们使用指令字段值、CURRENT_STATE 的各个寄存器和内存来更新 NEXT_STATE 的各个寄存器和内存，别忘了 PC+4 来读取下一条指令。

由于代码非常庞大，指令涉及非常多，由于它们都是严格按照 mips 指令的规范来进行更新的，没有过于复杂的逻辑（本来就是基础指令），具体代码可以看上传文件，就不再赘述。

打开(O) [n]		编号	寄存器名称	寄存器描述
1	.text	0	zero	第0号寄存器，其值始终为0
2 __start:	addiu \$v0, \$zero, 10	1	\$at	保留寄存器
3	addiu \$t0, \$zero, 5	2~3	\$v0~v1	values, 保存表达式或函数返回结果
4	addiu \$t1, \$t0, 300	4~7	\$a0~a3	arguments, 作为函数的前4个参数
5	addiu \$t2, \$zero, 500	8~15	t0 t7	temporaries, 供汇编程序使用的临时寄存器
6	addiu \$t3, \$t2, 34	16~23	s0 s7	saved values, 子函数使用时需要先保存原寄存器的值
7	addiu \$t3, \$t3, 45	24~25	\$t8~t9	temporaries, 供汇编程序的临时寄存器，补充\$t0~t7
8	syscall	26~27	k0 k1	保留，中断处理函数使用
		28	\$gp	global pointer, 全局指针
		29	\$sp	stack pointer, 堆栈指针，指向堆栈的栈顶
		30	\$fp	frame pointer, 保存帧指针
		31	\$ra	return address, 返回地址

通过查阅 mips 代码以及对应寄存器编号，我们可以使用 run [n]和 rdump 来查看寄存器随着指令读取后的变化，上图以 addi 为例，经过测试全部通过，同样不过多赘述。

三、额外代码设计（一共有三个）

main:

```
# 初始化寄存器 $2 和 $8
li $2, 0x1000
```

```
li $8, 0x1234
```

```
# 存储字节、半字、字
```

```
sb $9, 1($2)      # 存储字节
```

```
sh $10, 2($2)     # 存储半字
```

```
sw $12, 4($2)     # 存储字
```

```
# 从内存中加载数据
```

```
lb $9, 1($2)      # 加载字节
```

```
lbu $11, 1($2)    # 加载无符号字节
```

```
lh $10, 2($2)     # 加载半字
```

```
lhu $13, 2($2)    # 加载无符号半字
```

```
lw $12, 4($2)     # 加载字
```

```
# 条件分支
```

```
bltzal $1, label5 # 如果 $1 < 0, 则跳转到 label5
```

```
addi $1, $1, 0x13
```

```
label1:
```

```
addi $1, $1, 0x0a
```

```
# 条件判断
```

```
sltiu $4, $1, 0x05
```

```
beq $4, $0, label3
```

```
label2:
```

```
andi $5, $1, 0x02
```

```
beq $5, $0, label1
```

```
label3:
```

```
ori $6, $1, 1
```

```
xor $7, $6, $1
```

```
bne $7, $0, label5
```

```
label4:
```

```
li $1, 0x11
```

```
li $2, 0x23
```

```
# 条件判断
```

```
slti $3, $1, 0x05
```

```
bne $3, $0, label2
```

```
label5:
```

```
li $v0, 0x0a
syscall
```

额外代码设计如上图，我们可以 run 1+rdump 来查看代码，部分截图如下：

```
Instruction Count : 30
PC                : 0x00400064
Registers:
R0: 0x00000000
R1: 0x0000382d
R2: 0x10000000
R3: 0x00000001
R4: 0x00000000
R5: 0x00000000
R6: 0x00000007
R7: 0x00000000
R8: 0x12340000
R9: 0x00000013
R10: 0x00000014
R11: 0x00000013
R12: 0x00000012
R13: 0x00000014
R14: 0x00000000
```

注意时间所限，我完成最后一次跳转验证后没有设计程序出口，验证完程序后程序会陷入死循环，应避免用 go 命令。