《计算机组成与体系结构》实验报告

学院 智算学部 年级 2022 级 班级 5 班 组号 20

课程名称 计算机组成与体系结构 成绩_____

组长 李国鸿 同组实验者__

实验项目名称 指令集仿真器 -- TEMU

一. 实验目的

采用高级语言（C/C++、Java 或 Python 等）设计面向 32 位 MIPS 或 LoongArch 指令集子集的指令集仿真器 -- TEMU，用于实现对程序执行过程的模拟和调试。该指令集仿真器可模拟常见指令，支持单步执行、断点、显示寄存器信息、显示特定存储单元信息等功能，为后续的处理器设计实验提供仿真测试工具。

模拟指令的执行，产生 golden trace，方便后续与板上 SoC 进行交叉验证。

二. 实验内容

1. 实现调试功能

完善仿真器的调试功能，提供如下交互命令：

单步执行、

打印寄存器值 / 监视点值、

表达式求值、

扫描内存、

设置监视点、

删除监视点。

2. 模拟指令集

根据已有的框架代码，模拟每条指令的取值、译码、执行、访存、写回。对于所给的汇编代码，可以在仿真器上进行执行，并实时监测寄存器和内存中的值。

3. 记录 Golden Trace

对于每条指令，如果涉及到对寄存器的写入，要在 Golden Trace 文件中记录：

指令的 pc 值、

寄存器的编号、

写入的值。

4. 汇编测试

自行学习汇编代码的编写，测试指令集的模拟是否符合预期。

天津大学本科生实验报告

三. 实验原理与步骤（不需要贴代码）

请参考 TEMU 所需支持的 10 项功能和 golden trace，简单描述各项功能是如何实现的？注意：如果需要贴代码，只需要贴关键代码块，不要整段粘贴。

1. help, c, q

都已在框架中实现。

2. si [N]

直接调用 cpu_exec，如果 N 未给出默认为 1。

3. info SUBCMD

分成 r 和 w 两种情况。

r 情况，直接调用 display_reg()。

w 情况，递归输出每个监视点的信息：

```
void print_wp_rcs(WP *wp){
    if(wp == NULL){
        return;
    }
    print_wp_rcs(wp->next);
    printf("%d\t%s\t0x%08x\n", wp->NO, wp->expr, wp->value);
}
```

4. x N EXPR

调用表达式求值，得到地址值。

然后用 mem_read()，顺次访存 N 个字节，每四个进行一次换行。

5. w EXPR

从 _free 中取出一个新的监视点，插入到 head 前面，并调用表达式求值。

在每条指令执行后，都计算监视点中所有表达式的新值，比较和旧值是否相同，如果不同，则输出信息。

```
        /* TODO: check watchpoints here. */
        WP *wp = get_head();
        while(wp != NULL)
        {
            bool success = true;
            uint32_t value = expr(wp->expr, &success);
            if(value != wp->value)
            {
                // 输出信息
            }
            wp = wp->next;
        }
```

6. d N

从 head 开始，循环找到 N 号监视点，删除。

```
WP *p = get_head();
while(p != NULL){
    if(p->NO == n){
        free_wp(p);
        printf("Delete watchpoint #%d.\n", n);
        return 0;
    }
    p = p->next;
}
```

7. Golden Trace

仿照 log.txt 的记录，进行 golden.txt 的记录。

增加文件 golden.h 和 golden.c 实现对 Golden Trace 的初始化和记录功能。

```
FILE *golden_fp = NULL;
void init_golden() {
    golden_fp = fopen("golden.txt", "w");
    Assert(golden_fp, "Can not open 'golden.txt'");
}
void golden_write(uint32_t pc, uint32_t reg, uint32_t value) {
    fprintf(golden_fp, "0x%08x\t%u(%s)\t0x%08x\n", pc, reg, regfile[reg], value);
}
```

然后在指令的 make_helper 中，如果该指令涉及寄存器写入，就记录 Golden Trace。

以 lui 为例，在最后记录 pc，写寄存器，和寄存器新值：

```
make_helper(lui) {
    decode_imm_type(instr);
    reg_w(op_dest->reg) = (op_src2->val << 16);
    sprintf(assembly, "lui   %s,   0x%04x", REG_NAME(op_dest->reg), op_src2->imm);
    golden_write(cpu.pc, op_dest->reg, reg_w(op_dest->reg));
}
```

仿真程序 2：

四．实验结果（请给出仿真器 TEMU 的截图）

仿真程序 1：

logic（所给示例程序）

指令过程：

用 lui 和 ori 给 $at 和 $v0 赋值，

然后用 and 令 $t0 = $at & $v0。

结果截图：

```
(temu) si
1fc00010:   00 22 40 24         and   $t0,   $at,   $v0
(temu) si
temu: HIT GOOD TRAP at $pc = 0xbfc00014

1fc00014:   4a 00 00 00         temu_trap
```

Golden Trace：

```
0xbfc00000  1($at)  0x10100000
0xbfc00004  1($at)  0x10101010
0xbfc00008  2($v0)  0x01010000
0xbfc0000c  2($v0)  0x01011111
0xbfc00010  8($t0)  0x00001010
```

mem （所给示例程序）

指令过程：

用 lui 和 ori 给 $at 赋值，

用 sw 和 sb 将 $at 存到内存，

再用 lw 和 lb 从内存读取到 $v0。

结果截图：

```
1fc0007c:   ae 01 00 08         sw   $at,   0x0008($s0)
(temu) x 1 8+$s0
[./temu/src/monitor/expr.c,161,eval] reg found: $s0
0x80400008: 0x44556677
(temu) si
1fc00080:   8e 02 00 08         lw   $v0,   0x0008($s0)
(temu) p $v0
[./temu/src/monitor/expr.c,161,eval] reg found: $v0
0x44556677(1146447479)
(temu) si
1fc00084:   00 00 00 00         nop
(temu) si
temu: HIT GOOD TRAP at $pc = 0xbfc00088

1fc00088:   4a 00 00 00         temu_trap
```

Golden Trace：

```
0xbfc00000  16($s0) 0x80400000
0xbfc00004  1($at)  0x000000ff
0xbfc00018  1($at)  0x000000ee
0xbfc0002c  1($at)  0x000000dd
0xbfc00040  1($at)  0x000000cc
0xbfc00054  2($v0)  0x000000ff
0xbfc0005c  1($at)  0x44550000
0xbfc0006c  1($at)  0x44556677
0xbfc00080  2($v0)  0x44556677
```

仿真程序 3：

addi

指令过程：

用 lui 和 ori 给 $at 赋值，

然后用 addi 令 $v0 = $at - 2。

结果截图：

```
1fc00008:   20 22 ff fe         addi   $v0,   $at,   0xfffffffe(-2)
(temu) p $at
[./temu/src/monitor/expr.c,161,eval] reg found: $at
0x10101010(269488144)
(temu) p $v0
[./temu/src/monitor/expr.c,161,eval] reg found: $v0
0x1010100e(269488142)
(temu) si
temu: HIT GOOD TRAP at $pc = 0xbfc0000c

1fc0000c:   4a 00 00 00         temu_trap
```

Golden Trace：

```
0xbfc00000  1($at)  0x10100000
0xbfc00004  1($at)  0x10101010
0xbfc00008  2($v0)  0x1010100e
```

五．实验中遇到的问题和解决办法，并谈一下通过本次实验所获得的收获。

1. 大端序和小端序的判断

其实课上已经讲过了，但实验时候还是希望能确定一下，这里通过指令输出形式判断。

观察输出二进制指令处，根据循环看出是从高地址向低地址输出的。

```
void print_bin_instr(uint32_t pc) {
    int i;
    int l = sprintf(asm_buf, "%8x:   ", pc);
    for(i = 3; i >= 0; i --) {
        l += sprintf(asm_buf + l, "%02x ", instr_fetch(pc + i, 1)); // little endian
    }
    sprintf(asm_buf + l, "%*.s", 8, "");
}
```

再结合输出的二进制指令，和参考手册中的编码方式作对比。

1fc00014:　　4a 00 00 00　　temu_trap

可以发现高地址对应指令的大端，**低地址对应指令的小端**。

由此确定该 MIPS 处理器的存储方式为小端序，这样方便对示例程序 mem 的预期结果作判断。

2. 符号扩展的实现

一个简单的思路是判断最高位是 0 还是 1，然后扩展补相应的值。

不过经思考后，得到一种写法相对简单的做法：

```
// sign extent 16 bits to 32 bits
int32_t sext(int32_t x) {
    return (int32_t)((int16_t)x);
}
```

即利用 c 语言类型转换的特性，先截断成 16 位（因为只有低 16 位是有效的），再符号扩展到 32 位。

3. nop 指令的识别

如果一个指令的 32 位全 0， 就应当被识别为 nop，即什么也不做。

但是不特殊处理这种情况的话，全 0 指令可能会被识别为：

sll $zero, $zero, 0

虽然这样识别出来也没什么关系（因为也是什么都没有做），但输出的汇编信息可能会造成一些困扰。于是在 sll 的 make_helper 特殊处理如下：

```
make_helper(sll) {
    if(instr == 0){
        sprintf(assembly, "nop");
    }
    else{
        ...
    }
}
```

		教师签字： 年　月　日