Part1-design.pdf

1. machine.def 新增:

```
/* addOK */
#define ADDOK_IMPL                          \
{                                   \
    SET_GPR(RD, !OVER(GPR(RS), GPR(RT)));   \
}
DEFINST(ADDOK,      0x61,
    "addOK",    "d,s,t",
    IntALU,     F_ICOMP,
    DGPR(RD), DNA,  DGPR(RS), DGPR(RT), DNA)
```

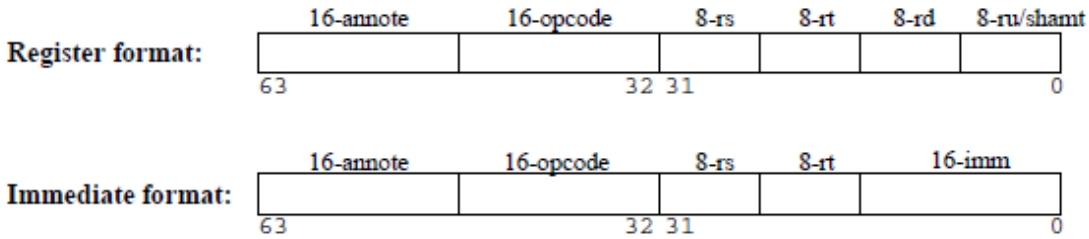
```
/* bitCount */
#define BITCOUNT_IMPL                           \
{                                       \
    int _result;                            \
    int _tmp_mask1 = (0x55) | (0x55 << 8);              \
    int _mask1 = (_tmp_mask1) | (_tmp_mask1 << 16);         \
    int _tmp_mask2 = (0x33) | (0x33 << 8);              \
    int _mask2 = (_tmp_mask2) | (_tmp_mask2 << 16);         \
    int _tmp_mask3 = (0x0f) | (0x0f << 8);              \
    int _mask3 = (_tmp_mask3) | (_tmp_mask3 << 16);         \
    int _mask4 = (0xff) | (0xff << 16);                 \
    int _mask5 = (0xff) | (0xff << 8);                  \
    _result = (GPR(RS) & _mask1) + ((GPR(RS) >> 1) & _mask1);       \
    _result = (_result & _mask2) + ((_result >> 2) & _mask2);       \
    _result = (_result + (_result >> 4)) & _mask3;          \
    _result = (_result + (_result >> 8)) & _mask4;          \
    _result = (_result + (_result >> 16)) & _mask5;         \
```

```
        SET_GPR(RT,UIMM ? _result : (32 - _result));
    \
     }
    DEFINST(BITCOUNT,   0x62,
        "bitCount", "t,s,u",
        IntALU,     F_ICOMP|F_IMM,
        DGPR(RT), DNA,  DGPR(RS), DNA, DNA)
```

2. 分析

第一步：分析指令格式

1. `Project 1 Instruction.pdf`中给出：`addOk` 和 `bitCount` 分别与 `add` 和 `xori` 指令的格式相同，因此可以得到：
   - `addOk` 符合 Register format
   - `bitCount` 符合 Immediate format



2. 仿照 `add` 和 `xori` 指令的定义可以得到新增指令的定义格式，但是此时两条新增指令的 `opcode` 并不能确定

第二步：得到opcode

1. 反汇编测试文件 `test1` ：

```
$SIMPLESCALAR/bin/sslittle-na-sstrix-objdump -d test1 > test1.s
```

2. 得到 `test1.s` 如下图所示，红框中的 `61` 就是 `addOk` 的opcode：

```
004001e0 <_ftext+a0> jal 004005d0 <exit>
...
004001f0 <test_addOK> addu $v1[3],$a0[4],$a1[5]
004001f8 <test_addOK+8> addu $v0[2],$zero[0],$zero[0]
00400200 <test_addOK+10> bgez $a0[4],00400218 <test_addOK+28>
00400208 <test_addOK+18> bgez $a1[5],00400218 <test_addOK+28>
00400210 <test_addOK+20> bgez $v1[3],00400238 <test_addOK+48>
00400218 <test_addOK+28> blez $a0[4],00400230 <test_addOK+40>
00400220 <test_addOK+30> blez $a1[5],00400230 <test_addOK+40>
00400228 <test_addOK+38> blez $v1[3],00400238 <test_addOK+48>
00400230 <test_addOK+40> addiu $v0[2],$zero[0],1
00400238 <test_addOK+48> jr $ra[31]
00400240 <addOK> addiu $sp[29],$sp[29],-40
00400248 <addOK+8> sw $s0[16],16($sp[29])
00400250 <addOK+10> addu $s0[16],$zero[0],$a0[4]
00400258 <addOK+18> sw $s1[17],20($sp[29])
00400260 <addOK+20> addu $s1[17],$zero[0],$a1[5]
00400268 <addOK+28> sw $s3[19],28($sp[29])
00400270 <addOK+30> 0x00000061:10111300
00400278 <addOK+38> sw $ra[31],32($sp[29])
00400280 <addOK+40> sw $s2[18],24($sp[29])
00400288 <addOK+48> jal 004001f0 <test_addOK>
00400290 <addOK+50> addu $s2[18],$zero[0],$v0[2]
00400298 <addOK+58> lui $a0[4],4096
004002a0 <addOK+60> addiu $a0[4],$a0[4],0
004002a8 <addOK+68> addu $a1[5],$zero[0],$s0[16]
004002b0 <addOK+70> addu $a2[6],$zero[0],$s1[17]
```

3. 同理，反汇编测试文件 `test2` ：

```
$SIMPLESCALAR/bin/sslittle-na-sstrix-objdump -d test2 > test2.s
```

4. 同样得到 `test2.s` 如下图所示，得到 `bitCount` 的opcode=62：

```
00400328 <bitCount+20> sw $a0[4],32($s8[30])
00400330 <bitCount+28> sw $a1[5],36($s8[30])
00400338 <bitCount+30> lw $v0[2],36($s8[30])
00400340 <bitCount+38> addiu $v1[3],$zero[0],1
00400348 <bitCount+40> bne $v0[2],$v1[3],00400370 <bitCount+68>
00400350 <bitCount+48> lw $v0[2],32($s8[30])
00400358 <bitCount+50> 0x00000062:02030001
00400360 <bitCount+58> sw $v1[3],16($s8[30])
00400368 <bitCount+60> j 00400388 <bitCount+80>
00400370 <bitCount+68> lw $v0[2],32($s8[30])
00400378 <bitCount+70> 0x00000062:02030000
00400380 <bitCount+78> sw $v1[3],16($s8[30])
00400388 <bitCount+80> lui $a0[4],4096
00400390 <bitCount+88> addiu $a0[4],$a0[4],0
00400398 <bitCount+90> lw $a1[5],32($s8[30])
004003a0 <bitCount+98> lw $a2[6],36($s8[30])
```

第三步：写出指令的DEFINST定义

- 根据前两步就可以确定定义，如 machine.def 新增 中所示

第四步：写出指令的具体实现

- 这个是最简单的部分，参照 ICS 中指令的实现即可，同样如 machine.def 新增 中所示

3. dubug & solution

1

1. bug描述：编译时遇到错误：`machine.def:200: stray '\' in program`
2. 原因：这是由于在 `#define` 行后面有注释导致的
3. 解决：去掉即可解决

2

1. bug描述：
   `panic: attempted to execute a bogus opcode [sim_main:sim-fast.c, line 444]`
2. 分析：根据报错信息，找到报错位置：

```
---
429       /* execute the instruction */
430       switch (op)
431         {
432 #define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3)            \
433         case OP:                                                        \
434           SYMCAT(OP,_IMPL);                                             \
435           break;
436 #define DEFLINK(OP,MSK,NAME,MASK,SHIFT)                                 \
437         case OP:                                                        \
438           panic("attempted to execute a linking opcode");
439 #define CONNECT(OP)
440 #define DECLARE_FAULT(FAULT)                                            \
441           { /* uncaught... */break; }
442 #include "machine.def"
443         default:
444           panic("attempted to execute a bogus opcode");
445         }
446
```

- 上面代码片段的逻辑是：根据 opcode 判断，当要执行的执行在 `machine.def` 中定义时，就会报错
- 在开始写指令的定义的时候，我并不知道应该如何得到两条指令的 opcode，按照 `machine.def` 中已有指令的定义，选取了一个没有被用到的值作为 `addOK` 的 opcode —— 0x60，所以才有以上报错信息
- 根据报错信息想到，opcode 一定是有特定的唯一值，不能随便定义
  1. 简单分析代码逻辑，`machine.def` 和 `sim-fast.c` 是分离的，他们之间相当于配置文件和程序代码之间的关系，所以排除 opcode 在 `sim-fast.c` 等源代码中定义的可能，得到 opcode 应当是在 `test` 测试文件中定义的
  2. 解决：将 `test` 文件 反汇编 很容易就得到了确定的、唯一的 opcode