Architecture Lab1 Report--Part 1

姜海天 19307110022

Step 1: Installation

The installation process was smooth, there were no errors other than what had been mentioned in the installation manual.

My appended environment variables in `~/.bashrc` is listed as follows:

```
export IDIR=~/simple
alias SIMFAST=$IDIR/simplesim-3.0/sim-fast
export TESTDIR=~/19307110022/architecturelab-handout/project/part1/tests
```

Step 2&3: Run test cases and analyze

This is a hard process as my TA didn't offer the test cases :). After spending several hours finding the test cases in various directories, I was sure that my TA forgot to offer it, so I contacted him and got the test cases.....

Now run the test case:

```
jht@ubuntu:~$ SIMFAST $TESTDIR/test1
...
sim: ** starting *fast* functional simulation **
panic: attempted to execute a bogus opcode [sim_main:sim-fast.c, line 444]
...
```

The output for test2 is almost the same. This is an inevitable result because I haven't start to write my implementation of the two instructions. If we take a glance at `sim-fast.c`, we can easily find the following code, which explains where the error info come from.

```
// part of file: sim-fast.c
switch (op){
#define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3)        \
    case OP:                                \
          SYMCAT(OP,_IMPL);                     \
          break;
#define DEFLINK(OP,MSK,NAME,MASK,SHIFT)                 \
        case OP:                            \
          panic("attempted to execute a linking opcode");
#define CONNECT(OP)
#define DECLARE_FAULT(FAULT)                        \
      { fault = (FAULT); break; }
#include "machine.def"
    default:
      panic("attempted to execute a bogus opcode");
}
```

Step 4: find opcode

Use `objdump` provided by the simple-scalar utilities.

```
jht@ubuntu:~$ $IDIR/simpleutils-990811/binutils/objdump --disassemble
$TESTDIR/test1

test1:     file format ss-coff-little

Disassembly of section .text:

00400140 <__start>:
  400140:   28 00 00 00     lw $16,0($29)
  400144:   00 00 10 1d
  400148:   a2 00 00 00     lui $28,4097
  40014c:   01 10 1c 00
  400150:   43 00 00 00     addiu $28,$28,-27296
  400154:   60 95 1c 1c
  400158:   43 00 00 00     addiu $17,$29,4
  40015c:   04 00 11 1d
    ...

00400240 <addOK>:
  400240:   43 00 00 00     addiu $29,$29,-40
  400244:   d8 ff 1d 1d
  400248:   34 00 00 00     sw $16,16($29)
  40024c:   10 00 10 1d
  400250:   42 00 00 00     addu $16,$0,$4
  400254:   00 10 04 00
  400258:   34 00 00 00     sw $17,20($29)
  40025c:   14 00 11 1d
  400260:   42 00 00 00     addu $17,$0,$5
  400264:   00 11 05 00
  400268:   34 00 00 00     sw $19,28($29)
  40026c:   1c 00 13 1d
  400270:   61 00 00 00     0x00000061:10111300
  400274:   00 13 11 10
  400278:   34 00 00 00     sw $31,32($29)
  40027c:   20 00 1f 1d
  400280:   34 00 00 00     sw $18,24($29)
  400284:   18 00 12 1d
  400288:   02 00 00 00     jal 4001f0 <test_addOK>
  40028c:   7c 00 10 00
    ...
```

MIPS takes first 16 bits as annotation and second 16 bits as opcode, so it's obvious that the opcode for `addOK` is 0x0061. Similarly, we can find that the opcode for `bitCount` is 0x0062.

Step 5: Implement the instruction

5.1 `addOK`

As mentioned in the lab manual, the format of the `addOk` instruction just follows the way of `add` instruction. For example, `addOK $s1, $s2, $s3` means setting `$s1` with value `0` if expression `$s2+$s3` overflows, otherwise with value `1`.

One one hand, since the simple-scalar uses MIPS as its subset of ISA, the `addOK` instruction is formatted in the way of normal MIPS instructions. So I can use the registers exactly in the same way as the `add` instruction. On the other hand, the fields in `DEFINST` of `addOK` should be exactly as same as those in `add`.

After reading the annotation in `machine.def`, I found that the author had implemented a series of "helper functions" to simplify the construction of instructions, one of which is `OVER(X,Y)` -- check for overflow for X+Y, both signed. So the implementation is just an encapsulation of the `OVER` function.

In a nutshell, My `addOK` instruction is implemented in the following way:

```
// part of file: machine.def
#define ADDOK_IMPL                \
  {                               \
    if (OVER(GPR(RS), GPR(RT))){  \
      SET_GPR(RD, 0);             \
    }else{                        \
      SET_GPR(RD, 1);             \
    }                             \
  }
DEFINST(ADDOK,      0x61,             // the same as ADD
  "addOK",          "d,s,t",
  IntALU,           F_ICOMP,
  DGPR(RD), DNA,    DGPR(RS), DGPR(RT), DNA)
```

After make the `sim-fast` again, I tested the new ISA and it worked.

```
jht@ubuntu:~$ SIMFAST $TESTDIR/test1
...
sim: ** starting *fast* functional simulation **
addOK(0x1, 0xffffffff)=1    Pass!
addOK(0x80000000, 0x80000000)=0 Pass!
...
```

5.2 `bitCount`

Again, as mentioned in the lab manual, the format of `bitCount` instruction is the same as `xori`. So `bitCount` is an instruction taking immediate as its operand. But in the manual, the example of the instruction is offered in the format of a function like `bitCount(0x5,1)`, therefore I have to infer which "parameter" is immediate and which is stored in register. So I again turn to the output of `objdump` for `test2`, and examine the fragment of `bitCount` instruction.

```
  ...
400358:   62 00 00 00     0x00000062:02030001
40035c:   01 00 03 02
  ...
400378:   62 00 00 00     0x00000062:02030000
40037c:   00 00 03 02
  ...
```

MIPS uses the last 16bits as immediate, therefore the 0-1 number is the immediate and the number to be counted is stored in register.

Similarly to the implementation of `xori`, `UIMM` takes the value of the immediate. And I used my implementation of bitwise operation in data-lab to implement this instruction.

```
// part of file: machine.def
#define BITCOUNT_IMPL             \
  {                               \
    int x = GPR(RS);              \
    int sum = 0;                  \
    int mask = (1 << 8) + 1;      \
    mask = (mask << 16) + mask;   \
    sum = x & mask;               \
    sum = sum + ((x >> 1) & mask);\
    sum = sum + ((x >> 2) & mask);\
    sum = sum + ((x >> 3) & mask);\
    sum = sum + ((x >> 4) & mask);\
    sum = sum + ((x >> 5) & mask);\
    sum = sum + ((x >> 6) & mask);\
    sum = sum + ((x >> 7) & mask);\
    sum = sum + (sum >> 16);      \
    sum = sum + (sum >> 8);       \
    int ones = sum & 0xff;        \
    if (UIMM == 1){               \
      SET_GPR(RT, ones);          \
    }else{                        \
      SET_GPR(RT, 32 - ones);     \
    }                             \
  }
DEFINST(BITCOUNT,   0x62,
  "xori",         "t,s,u",
  IntALU,         F_ICOMP|F_IMM,
  DGPR(RT), DNA,  DGPR(RS), DNA, DNA)
```

Testing results.

```
jht@ubuntu:~$ SIMFAST $TESTDIR/test2
...
sim: ** starting *fast* functional simulation **
bitCount(0x5, 1)=2  Pass!
bitCount(0x7, 1)=3  Pass!
bitCount(0x7, 0)=29 Pass!
...
```