# Computer Architecture

## Project1 Part1 Design

Due on Wednesday, April 14, 2013

*Zhang Weihua*

**Wang Xin**

# Contents

**Overview** **3**

**Instruction Format** **4**

**AddOk** **5**

**BitCount** **5**

**Result** **6**

# Overview

To get started with the SimpleScalar simulator structure, we should first get familiar with the simulation process of the CPU. As described in the project document:

> The core of the sim-fast simulator resides in two files: sim-fast.c and machine.def. There is a "while (TRUE)" loop in sim-fast.c which you should pay much attention. There is a detailed specification in machine.def. I would suggest you to read it carefully before you begin your work.

Check the main structure of sim-fast.c, we found the main loop of the file (Listing 1):

Listing 1: main loop structure of sim-fast.c

```
 while (TRUE)
    {
      /* maintain $r0 semantics */
      regs.regs_R[MD_REG_ZERO] = 0;
      /* keep an instruction count */
#ifndef NO_INSN_COUNT
      sim_num_insn++;
#endif /* !NO_INSN_COUNT */

      /* load instruction */
      MD_FETCH_INST(inst, mem, regs.regs_PC);

      /* decode the instruction */
      MD_SET_OPCODE(op, inst);
      /* execute the instruction */

      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)                               \
       { /* uncaught... */break; }
#include "machine.def"
     default:
       panic("attempted to execute a bogus opcode");
     }
      /* execute next instruction */
      regs.regs_PC = regs.regs_NPC;
      regs.regs_NPC += sizeof(md_inst_t);
    }
```

The structure of the loop was a simulation of a CPU clock cycle.

- At the beginning of every loop, Line 4 set the MD_REG_ZERO to zero to maintain the semantics of register $r0
- The macro MD_FETCH_INST corresponds to the Instruction Fetch stage in a single cycle datapath. CPU will fetch the instruction stored at regs.regs.PC.

Overview continued on next page...

• MD_SET_OPCODE corresponds to the Decode stage but it only determine the type of the opcode. The operator will be stored in the variable op.
• DEFINST check whether the instruction opcode was defined, if so, SYMCAT will combine the implementation defined in machine.def with the related instruction and excute the following steps. Otherwise, it will throw an exception.
• Then the simulator will update the PC and fetch the next instruction (Line 33-35).

## Instruction Format

The file machine.def is actually a symbolic link which points to target-pisa/pisa.def. This file defines all aspects of the SimpleScalar instruction set architecture. Each instruction set in the architecture has a DEFINST() macro call included below. In other words, the DEFINST defines the format of the instruction including the opcode, opname, operands, dependencies. The #define XXX_IMPL implements the instruction being defined. Hence we should implement the expression for new instructions here.
Here is a example:

Listing 2: definition of add instruction

```
#define ADD_IMPL                                        \
  {                                                     \
    if (OVER(GPR(RS), GPR(RT)))                         \
      DECLARE_FAULT(md_fault_overflow);                 \
                                                        \
    SET_GPR(RD, GPR(RS) + GPR(RT));                     \
  }
DEFINST(ADD,              0x40,
     "add",               "d,s,t",
     IntALU,         F_ICOMP,
     DGPR(RD), DNA,       DGPR(RS), DGPR(RT), DNA)
```

Listing 2 demonstrates the implementation of "add" instruction. In the macro #define ADD_IMPL is the C expression of the instruction. From the name of each macros we can get the meaning of each variable. This instruction get the source variables from register RS and RT. The helper function OVER detects whether the sum of two variables overflows. If not, add the value of two variables and write the sum in register RD. The following information defines the format of instruction.
• The opcode is 0x40
• The operands is "d,s,t". This field is used by disassembler to specify the printed order.
• The function unit is Int_ALU and the instruction flag is I_COMP, which is predefined in ss.h
• The output dependency designator is RD only, hence we need to use a spaceholder DNA to mark the unused field.
• The input dependency are register RS and RT.
Here is another sample using immediate number:

Listing 3: definition of xori instruction

```
#define XORI_IMPL                                       \
  {                                                     \
    SET_GPR(RT, GPR(RS) ^ UIMM);                            \
  }
DEFINST(XORI,             0x53,
     "xori",              "t,s,u",
     IntALU,              F_ICOMP|F_IMM,
     DGPR(RT), DNA,            DGPR(RS), DNA, DNA)
```

Instruction Format continued on next page... 

The most significant difference between “add” and ”xori” is the operands field. Instruction “xori” uses unsigned immediate value “u” and the function flag includes F_IMM to mark the immediate number.

## AddOk

In order to add a new instruction, we should first know the opcode of the instruction. Therefore we need to disassemble the test case:

Listing 4: Disassembling the test case

```
sslittle-na-sstrix-objdump -x -d test1>test1.dump
```

In the dump file of test1, we can find the trace of the target instruction “addOK”:

Listing 5: addOK instruction in dump file

```
00400270 <addOK+30> 0x00000061:10111300
```

The opcode 0x61 has not appeared in the machine.def, therefore it must be the opcode of “addOK”. Then we can get the implementation of “addOK” easily (List 6):

Listing 6: addOK instruction implementation

```
#define ADDOK_IMPL                              \
    {                                               \
        if(OVER(GPR(RS),GPR(RT)))           \
                SET_GPR(RD,0);              \
        else                                \
                SET_GPR(RD,1);              \
    }
DEFINST(ADDOK,      0x61,
        "addOK",    "d,s,t",
        IntALU,         F_ICOMP,
        DGPR(RD), DNA,      DGPR(RS), DGPR(RT),DNA)
```

The process of this instruction is only a subset of “add” instruction.

## BitCount

Similarly, we can get the opcode of “bitCount” instruction from dump file of test2.

Listing 7: bitCount instruction in dump file

```
00400358 <bitCount+50> 0x00000062:02030001
```

We should pay attention to the format of this instruction for it involves an immediate number to distinguish which digit should we count: 1 or 0.

Listing 8: bitCount instruction implementation

```
#define BITCOUNT_IMPL                                   \
  {                                             \
    unsigned int count = 0;             \
    unsigned int num = 0;                   \
    for(num = (unsigned)GPR(RS); num > 0; num >>= 1)  \
        count += (num & 0x1);                   \
    SET_GPR(RT, IMM ? count : (32 - count));        \
```

BitCount continued on next page...

```
  }
DEFINST(BITCOUNT,              0x62,
     "bitCount",               "t,s,u",
     IntALU,              F_ICOMP|F_IMM,
     DGPR(RT), DNA,            DGPR(RS), DNA, DNA)
```

# Result

Then we rebuild the sim-fast simulator:

Listing 9: Rebuild the sim-fast simulator

```
make clean
make config-pisa
make sim-fast
```

Run the two test cases using sim-fast and get the right output:

Listing 10: Output

```
sim: ** starting *fast* functional simulation **
addOK(0x1, 0xffffffff)=1 Pass!
addOK(0x80000000, 0x80000000)=0    Pass!

sim: ** starting *fast* functional simulation **
bitCount(0x5, 1)=2  Pass!
bitCount(0x7, 1)=3  Pass!
bitCount(0x7, 0)=29 Pass!
```