UNISIM
GenISSLib Manual

Gilles Mouchard
Yves Lhuillier

CEA List

1 Introduction

Instruction set simulators are usefull for processor simulation, either architecture or micro-architecture simulation. Architecture simulation, also called functional simulation, refers to simulation of the processor instruction set, whereas micro-architecture simulation refers to components inside the processor such as pipelines, caches, functional units, branch predictors. Implementing the instruction set into a software instruction set simulator is needed for developping and testing software before the target processor is available or for analyzing softwares without disturbing their execution. To go from the instruction set specification to a software implementation of the instruction set, it is convenient to have a description language for easily describing the instruction encoding, but most description languages are restricted to only few syntaxical constructions for describing the instructions behavior. Beside, it is important to have a multipurpose instruction decoder as needs evolves. GenISSLib is intended for developping such instruction set simulators. It uses a description language for describing instruction encoding. C++ source code blended with the description language allow the use of complex instruction behavior, or even more functionalities like disassembling, binary translation, system calls translation.

2 Quick Start

We will now presents you how to use GenISSLib through an example. Suppose that we want to build an instruction set simulator for a simple 16-bit RISC processor, see table 2. We need GenISSLib to generate C++ class 'Decoder' to decode an instruction, and a C++ class 'Operation' representing the decoded instruction. class 'Operation' would have a C++ method named 'execute' to execute the instruction, and a C++ method to disassemble the instruction into a string buffer:

- `Operation *Decoder::decode(uint16_t addr, uint16_t instruction)`
  - `addr` is the address of the instruction
  - `instruction` is the instruction word to decode
  - `decode` returns an instance of class `Operation` containing the decoded instruction
- `void Operation::execute()`
- `void Operation::disasm(uint16_t pc, char *s)`
  - `pc` is the address of the instruction
  - `s` is a string where to disassemble the instruction

GenISSLib automatically generates the `Decoder` class. To ask GenISSLib to generate `execute` and `disasm` methods, we have to declare them using the description language, see figure 1. The grammar of the description language is on figure 15. In the description language, `execute` and `disasm` are actions having a prototype. The action prototype is simply a template for the action. As you can see, the declaration in the instruction set simulator source code directly derives from these action prototypes. If an instruction is invalid, we can tell GenISSLib to use a default implementation for the action. The default action implementation for `execute` is printing `"Unknown instruction\n"` and exit whereas the default action implementation for `disasm` is writing a `"?"` into the string `s`.

```
action {void} execute() {
    printf("Unknown instruction\n");
    exit(-1);
}

action {void} disasm({uint16_t} {pc}, {char *} {s}) {
    sprintf(s, "?");
}
```

Figure 1: Declaring the `execute` and `disasm` action prototypes.

instructions	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
add	0	0	0	0	rd				rs1				rs2			
sub	0	0	0	1	rd				rs1				rs2			
or	0	0	1	0	rd				rs1				rs2			
and	0	0	1	1	rd				rs1				rs2			
not	0	1	0	0	rd				rs				x	x	x	x
shl	0	1	0	1	rd				rs				x	x	x	x
shr	0	1	1	0	rd				rs				x	x	x	x
load	0	1	1	1	rd				base				index			
store	1	0	0	0	rd				base				index			
branch	1	0	0	1	addr											
bne	1	0	1	0	rs				offset							

Figure 2: Instruction Set Summary.

add add

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	rd				rs1				rs2			

(rd) $\leftarrow$ (rs1) + (rs2)

Figure 3: The add instruction.

```
op add(0b0000[4]:rd[4]:rs1[4]:rs2[4])
```

Figure 4: declaration of the add operation

not negation

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	rd				rs				x	x	x	x

(rd) $\leftarrow$ (rs)

Figure 5: The not instruction.

```
op not(0b0100[4]:rd[4]:rs[4]:?[4])
```

Figure 6: declaration of the not operation

We can now declare instruction encoding and the action implementation with the description language. First, let's consider the add instruction, see figures 3 and 2: it has a 4-bit opcode which is 0000 and three 4-bit long operand register numbers encoded into the instruction word which are rd, rs1 and rs2. In the description language `operation` refers to an instruction. `0b0000[4]` refers to the 4-bit opcode, `rd[4]` to the destination register number which is 4-bit long, `rs1[4]` and `rs2[4]` respectively to the first and second source register numbers. Each bit fields are separated by `:`. Instead of using binary digits as in `0b0000[4]`, we can also use decimal digits as in `0[4]`, or hexadecimal digits as in `0x0[4]`.

Now, let's declare the not instruction encoding, see figures 5 and 2: it has a 4-bit opcode which is 0001, two 4-bit long operand register numbers which are rd and rs, and 4 don't care bits which can be either 0 or 1. The 4 don't care bits are represented by `?[4]` in the declaration of the not operation.

Now, let's implement the actions of the add operation, see figures 3 and 7. Implementation of an action is just C code. We can directly use rd, rs1 and rs2 which have been declared in the operation. We use an array (`gpr`) to represent the registers. The external declaration of that C array can be done anywhere into the description language : we've just added between { and }, the external declarations of `gpr` and the memory access functions `mem_read` and `mem_write`, see figure 8.

```
add.execute = {
    gpr[rd] = gpr[rs1] + gpr[rs2];
}

add.disasm = {
    sprintf(s, "add r%u, r%u, r%u", rd, rs1, rs2);
}
```

Figure 7: implementation of the `execute` and `disasm` of the add operation.

Now, let's write the declaration of the load operation, and the implementation of its actions, see figures 9 and 10: it has a 4-bit index which must be sign extended and added to a base register to compute the effective address of the load. Thus we use the `sext` modifier into the operation declaration. The minimum size in bits of the C variable `index` holding the `index` operand field of the operation is supplied between <and >. When decoding a load instruction, GenISSLib will sign extends the `index` bit field, and store it into the `index` C variable.

The implementation of the `disasm` action of the bne operation uses the `pc` parameter because we need that the disassembling of the bne instruction contains the target address of the conditional branch, that is `pc + offset` not `offset`, see figures 11 and 12.

```
{
#include <inttypes.h>
#include <stdio.h>

extern uint16_t cia;
extern uint16_t nia;
extern uint16_t gpr[16];

extern uint16_t mem_read(uint16_t addr);
extern void mem_write(uint16_t addr, uint16_t value);
}
```

Figure 8: external declaration of C/C++ variables and functions.

load load word

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	rd				base				index			

```
ea ← (base) + index
(rd) ← Mem[ea]
```

Figure 9: The load instruction.

```
op load(0b0111[4]:rd[4]:base[4]:sext<16> immed[4])

load.execute = {
    gpr[rd] = mem_read(gpr[base] + immed);
}

load.disasm = {
    sprintf(s, "load r%u, %d(r%u)", rd, immed, base);
}
```

Figure 10: The load declaration and implementation.

bne conditional branch

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	rs				offset							

```
if (rs) <>0 then
    NIA ← CIA + offset
else
    NIA ← CIA + 1
```

Figure 11: The bne instruction.

Finally, we uses the generated function to implement the instruction set simulator main loop, see figure 14. We also implement the external function used into the action implementations,

```
op bne(0b1010[4]:rs[4]:sext<16> offset[8])
bne.execute = {
    if(gpr[rs] != 0)
        nia = cia + offset;
}
bne.disasm = {
    sprintf(s, "bne r%u, 0x%04x", rs, pc + offset);
}
```

Figure 12: The bne declaration and implementation.

see figure 13.

```
#include <inttypes.h>"

uint16_t cia;
uint16_t nia;
uint16_t gpr[16];
uint16_t mem[1 << 16];

uint16_t mem_read(uint16_t addr) {
    return mem[addr];
}

void mem_write(uint16_t addr, uint16_t value) {
    mem[addr] = value;
}
```

Figure 13: The implementation of the external data and functions.

```
#include "risc16.hh"
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;
    FILE *f;
    uint16_t instruction;
    Operation *operation;
    char disasm_buffer[256];

    cia = 0;
    nia = 0;
    for(i = 0; i < 16; i++) gpr[i] = 0;
    f = fopen("image", "rb");
    fread(mem, 1, sizeof(mem), f);
    fclose(f);
    Decoder decoder;
    for(i = 0; i < 1000; i++) {
        instruction = mem_read(cia);
        operation = decoder.decode(cia, instruction);
        operation->disasm(cia, disasm_buffer);
        printf("0x%04x: 0x%04x %s\n", cia, instruction,
                                                disasm_buffer);
        nia = cia + 1;
        operation->execute();
        cia = nia;
    }
    return 0;
}
```

Figure 14: The main simulation loop.

3 Technical Reference

input $\rightarrow$ decl_list
decl_list $\rightarrow \epsilon \mid$ decl_list decl $\hookleftarrow$
decl $\rightarrow \epsilon \mid$ op_decl $\mid$ action_proto_decl $\mid$ action_decl $\mid$ source_code_decl $\mid$ include_decl
source_code_decl $\rightarrow$ **source_code**
op_decl $\rightarrow$ **op identifier (** bitfield_list **)**
bitfield_list $\rightarrow$ bitfield $\mid$ bitfield_list **:** bitfield
bitfield $\rightarrow$ **integer** [**integer**] $\mid$ sext_modifier size_modifier **identifier** [**integer**] $\mid$? [**integer**]
sext_modifier $\rightarrow \epsilon \mid$ **sext**
size_modifier $\rightarrow \epsilon \mid$ **<** **integer** **>** $\mid$ **<** **>**
action_proto_decl $\rightarrow$ static_modifier **action source_code identifier (** params **) source_code**
params $\rightarrow \epsilon \mid$ **source_code**
static_modifier $\rightarrow \epsilon \mid$ **static**
action_decl $\rightarrow$ **identifier . identifier = source_code**
include_decl $\rightarrow$ **include string**

source_code is C++ source between { and }
identifier is an alphanumeric identifier
integer is a positive integer either binary, decimal or hexadecimal
string is a double-quoted character string
C (/* */) et C++ (//) comments are allowed anywhere.

Figure 15: The description language grammar.