`rv32i`

—

RISC-V RV32I EMULATOR

selendym
(selendym@tuta.io)

1. Introduction

The subject of this project is the emulation of the RISC-V instruction-set architecture[1] (ISA), more precisely, the "RV32I base integer instruction set" variant, which is a 32-bit ISA.

The current implementation does what it should: it emulates RV32I and is able to run programs compiled with the GCC cross-compilation toolset for RISC-V bare-metal targets[2]. As the implementation is "bare-metal"[3], the full C standard library cannot be used to write test programs. However, to support `malloc`, the implementation does emulate handling of `sbrk` system calls, although very primitively. Also, input-output (IO) is handled with custom `ecall` environment calls that are mapped to the standard input and output of the emulator. The custom `ecall`s the test program uses are implemented with inline assembly, because C does not support executing arbitrary instructions directly.

The implementation provides a rudimentary command-line interface (CLI) for the emulator; this is described in more detail in subsection 3.1.

2. Structure

A class diagram of the project is given in Figure 1. Arrows represent dependencies and point from the requiring entity to the required one.

Date: 2020-01-30 08:17:11Z.

[1] https://riscv.org/specifications/

[2] https://aur.archlinux.org/packages/riscv64-unknown-elf-gcc/

[3] There is no support from the runtime environment, no operating system to handle system calls.

```
const reg_arr_t &get_reg_arr() const;
const word_t &get_reg( reg_idx_t index ) const;
const word_t &get_sp_reg() const;
const word_t &get_pc_reg() const;
void set_reg_arr( const reg_arr_t &reg_arr );
void set_reg( reg_idx_t index, word_t value );
void set_sp_reg( word_t value );
void set_pc_reg( word_t value );
void step();
```

Listing 1: Public interface of class `Cpu`.

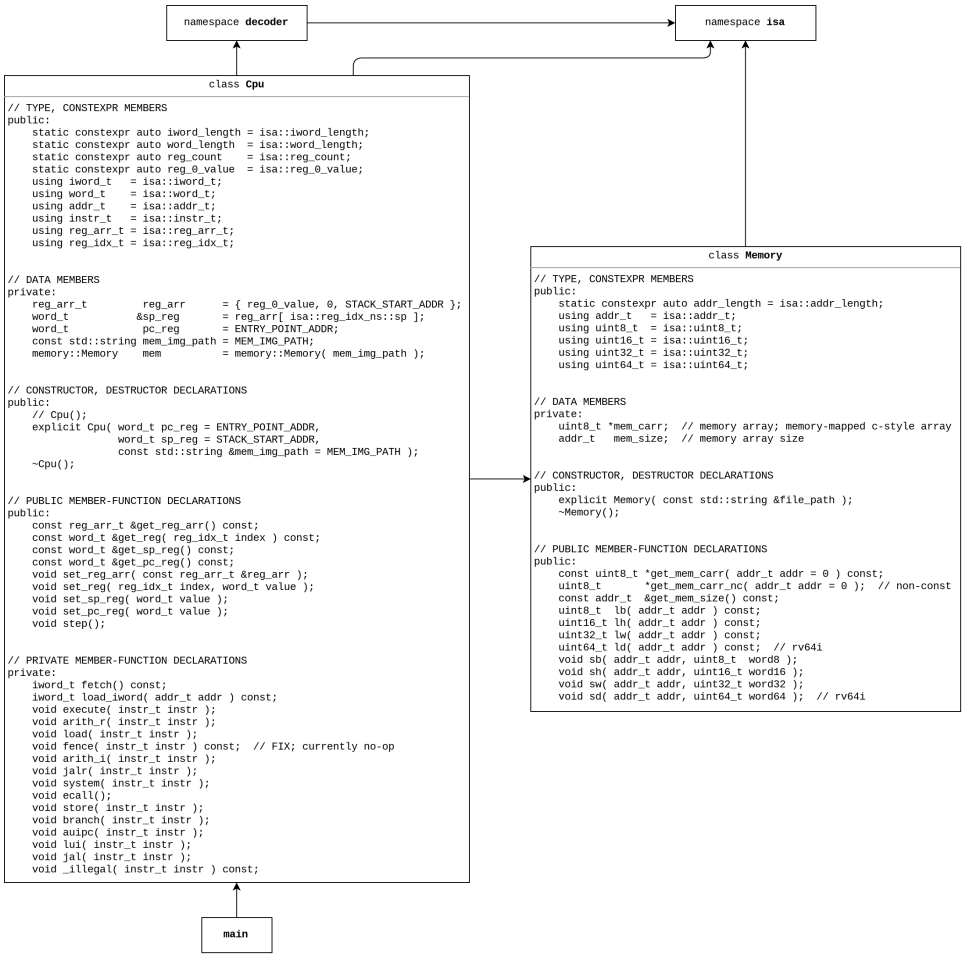

FIGURE 1. Class diagram of the project.

2.1. **Cpu.** Class `Cpu` (`src/cpu/cpu.{h,c}pp`) implements the main interface of the emulator proper. Its public interface is given in Listing 1.

`Cpu` implements the `get_*` and `set_*` member functions for getting and setting, and the `step` member function for advancing, the state of the emulation run. Each invocation of `step` corresponds to the execution of the instruction (in emulated memory) pointed to by the current value of the `pc_reg` data member. With each step, `pc_reg` is updated automatically; thus, to advance the emulation run, only `step` needs to be called.

Each invocation of `step` proceeds by executing in succession the functions `fetch`, `decode`, and `execute`. `fetch` fetches the instruction to be executed from the (emulated) memory address corresponding to `pc_reg`. `decode` decodes the fetched instruction and returns a struct comprising the decoded fields. `execute` executes the decoded instruction and changes the state of the emulation run accordingly.

`fetch` has a trivial implementation, and `decode` is described in detail in subsection 2.2. The implementation of `execute` composes the body of `Cpu`. Instructions are first divided into groups corresponding to their major opcodes. These groups are then executed by their namesake functions; for example, `arith_i` is responsible for executing the instructions belonging to the group “immediate arithmetic”.

```
instr_t decode( iword_t instr_word );
```

Listing 2: "Public interface" of namespace `decoder`.

```
struct instr_t
{
// raw instruction word
    iword_t instr_word;
// instruction-word bit fields
    opcode_e opcode;  // instr_word[ 6: 0]
    iword_t funct3;   // instr_word[14:12]
    iword_t funct7;   // instr_word[31:25]
    iword_t rd;       // instr_word[11: 7]
    iword_t rs1;      // instr_word[19:15]
    iword_t rs2;      // instr_word[24:20]
    word_t imm;  // variant; depends on opcode format type; sign-extended
// decoded instruction and type
    opcode_type_e opcode_type;  // opcode format type
    mnem_e mnem;  // mnemonic; 'fully resolved opcode'; e.g., 'XOR'
};  // END struct instr_t
```

Listing 3: Struct `instr_t`.

```
const uint8_t *get_mem_carr( addr_t addr = 0 ) const;
uint8_t       *get_mem_carr_nc( addr_t addr = 0 );  // non-const
const addr_t  &get_mem_size() const;
uint8_t  lb( addr_t addr ) const;
uint16_t lh( addr_t addr ) const;
uint32_t lw( addr_t addr ) const;
uint64_t ld( addr_t addr ) const;  // rv64i
void sb( addr_t addr, uint8_t  word8 );
void sh( addr_t addr, uint16_t word16 );
void sw( addr_t addr, uint32_t word32 );
void sd( addr_t addr, uint64_t word64 );  // rv64i
```

Listing 4: Public interface of class `Memory`.

2.2. **Decoder.** Namespace `decoder` (`src/cpu/decoder.{h,c}pp`) implements the instruction decoder of the emulator.[4] Its "public interface" is given in Listing 2.

`decoder` implements the `decode` function for decoding a raw instruction word into an `instr_t` struct (Listing 3) comprising the decoded fields. `decode` serves as the interface for decoding and performs some preliminary steps before calling more specific decoding functions. As with `execute`, instructions are first divided into groups corresponding to their major opcodes. These groups are then decoded by their namesake functions with `decode_mnem_` prepended; for example, `decode_mnem_arith_i` is responsible for decoding the instructions belonging to the group "immediate arithmetic". `decode_imm` differs from the other functions by not belonging to any single group and is responsible for decoding immediate values for all groups.

[4] `decoder` can be considered as a stateless class, comprising only (static) member functions.

```
using std::uint8_t,  std::int8_t;
using std::uint16_t, std::int16_t;
using std::uint32_t, std::int32_t;
using std::uint64_t, std::int64_t;
constexpr unsigned iword_length = 32;  // instruction bit width
using iword_t = uint32_t;  // instruction word type
using siword_t = int32_t;  // instruction word type; signed
constexpr unsigned word_length = 32;  // register bit width
using word_t = uint32_t;  // register word type
using sword_t = int32_t;  // register word type; signed
constexpr unsigned addr_length = word_length;  // memory address bit width
using addr_t = word_t;  // memory address type
constexpr unsigned reg_count = 32;  // register count
constexpr word_t reg_0_value = 0;  // register-0 fixed value
using reg_arr_t = std::array< word_t, reg_count >;
using reg_idx_t = reg_arr_t::size_type;
constexpr auto iword_mask = util::bit_mask< iword_t, iword_length >;
constexpr auto iword_extract = util::bit_extract< iword_t, iword_length >;
constexpr auto word_mask = util::bit_mask< word_t, word_length >;
constexpr auto word_extract = util::bit_extract< word_t, word_length >;
constexpr auto addr_mask = word_mask;
constexpr auto addr_extract = word_extract;
namespace reg_idx_ns;
enum class opcode_type_e;
enum class opcode_e;
enum class mnem_e;
struct instr_t;
```

Listing 5: Contents of namespace `isa`.

2.3. **Memory.** Class `Memory` (`src/cpu/memory.{h,c}pp`) implements the emulated memory system that `Cpu` uses. Its public interface is given in Listing 4.

The `get_mem_size` member function returns the memory array size in (8-bit) bytes. The `get_mem_carr` member function and its non-constant variant are used to provide direct access to the underlying raw memory array; these can be considered as a rudimentary form of direct memory access (DMA) and are used to provide a simple interface to IO functions. The `l{b,h,w,d}` member functions allow addressed loads (reads) from a given memory location and return values of 8 (byte), 16 (half-word), 32 (word), and 64 (double-word) bits, respectively. The `s{b,h,w,d}` member functions allow addressed stores (writes) to a given memory location of values of 8 (byte), 16 (half-word), 32 (word), and 64 (double-word) bits, respectively.

`Memory` implements the raw memory array using Linux system library call `mmap` to memory-map a file, serving as the memory image, onto the emulated memory address space. The semantics of `mmap` allow any modifications made by the emulation run to transparently and automatically be mirrored in the memory image file, which can then be examined during or after the run to help in troubleshooting possible problems.

2.4. **ISA.** Namespace `isa` (`src/cpu/isa.hpp`) implements the shared type definitions of the emulator. Its contents are given in Listing 5.

2.5. **Main.** Function `main` (`src/cpu/main.cpp`) implements a simple CLI for the emulator.

2.6. **Test.** Function `main` (`src/test/test.c`) implements a test program for the emulator to run.

3. Instructions for building and usage

3.1. **Main.** The CLI for the emulator, `main`, is built by invoking `make [all]` (the brackets signify optionality) in the `src/cpu` directory. After a successful compilation, the `main` executable is located at `src/test/main`. Note that a GCC version supporting `-std=c++17` is required.

Usage: `./main [<step_count>] [<pc>] [<sp>]`

If `step_count` is not given, the largest possible value, -1, is used (with unsigned arithmetic, this will wrap around). Note that unless required, `pc` and `sp` should not be set explicitly; these correspond to the initial program counter (pc), which should point to the address of `_start`, and the initial stack pointer (sp), which by default points to just past the end of the memory image.

3.2. **Test.** The test program for the emulator, `test`, is built by invoking `make [all]` in the `src/test` directory. After a successful compilation, two memory images, `mem.img.clean` and `mem.img` are created (the latter is simply a copy of the former). Note that this step is optional, as a precompiled `mem.img.clean` is provided; otherwise, a GCC version supporting RISC-V bare-metal cross-compilation is required.

After an emulation run, the memory image file, `mem.img`, has most likely been modified by the run. If desired, the image can be restored to a clean state by invoking `make reset-mem` in the `src/test` directory; this will also create the image if it does not exist.

4. Testing

Testing of the emulator proper was mostly realized by having the emulator run a test program compiled for the implemented ISA, RV32I, using GCC as a cross-compiler. As the amount of differing instructions in RV32I is about 50 in total, a slightly more extensive test program should be able to cover these many times over. In case bugs are present, it is very likely that the test program will not run correctly, if at all. The lack of any unit tests was mostly due to lack of time and also of need in this particular case — the coverage of the test program might not be 100%, but it is likely reasonably close to this.

The test program is written mostly in C, with start-files and some utilities written in the RV32I subset of RISC-V assembly language. As the target for compilation is bare-metal, custom start-files are required to set up and tear down the emulated environment to allow correct program execution. Namely, the `_start` and `_exit` functions set up and tear down the emulation, respectively, and print diagnostic messages along the way to ease troubleshooting.

Due to the way the memory system is implemented, examining the contents of the emulated memory during or after an emulation run is made easy. There are two files for the memory image, `mem.img.clean` and `mem.img`. The former is the untouched, "clean", image, while the latter is the actually used memory image and may have changes made by the emulator. The made changes are permanent, so if a "clean" emulation run is required, invoking `make reset-mem` in the test directory allows resetting the used memory image to the clean state. To see the changes

made in the memory image without great pains, a tool with hex diff capability is recommended[5].

[5]One such tool is `binwalk`, called with `binwalk --hexdump --red mem.img.clean mem.img`.