MPACT-Sim Assembler

Assembler generation from instruction descriptions.

@torerik

Last modified: Jan 16, 2025

Motivation

In order to properly evaluate the impact of new ISA additions (or an entirely new ISA) it is necessary to create and run code. However, except for hand assembling or decoding instructions from their textual form, it requires some tool chain support.

Assemblers are key tools in writing code in the early stage in developing new instructions/ISAs for which there is no compiler support. Even for ISAs with compiler support, there may be instructions that cannot be targeted by the compiler, either because the compiler implementation lags the ISA, or due to the difficulty of mapping higher level languages to their semantics.

In general, writing assemblers is a well understood task. Some codegen tools like LLVM even have support for generating assemblers from its architectural description. However, the effort in creating these descriptions and producing the tools is significant, as it is aimed at producing a full compiler toolchain.

The MPACT-Sim simulator infrastructure uses simple instruction description files to generate C++ code for instruction decoding, greatly reducing the effort in creating (or extending) instruction level simulators. These same description files can now be used to generate core pieces of an assembler, greatly increasing the velocity of providing software development tools in early development stages. The incremental effort of adding additional instructions to the assembler becomes nearly trivial.

MPACT Instruction Descriptions

MPACT-Sim describes instructions in two different format files. The `.isa` file describes the instruction set in terms of operation name, instruction operands, resource usage, semantic function, and disassembly formatting. The `.bin_fmt` file describes the binary instruction formats and the encodings of individual instructions.Together they provide sufficient information to generate most of the instruction decoder. The only code that the simulator writer has to write is to map instruction operand encodings to simulator instruction operand objects, and to write the top level control of the decoder.

A similar approach is used for the assembler, using the information from the disassembly format to parse assembly instruction source, and information from the instruction encoding specification to provide support for encoding the binary instruction words.

Generated Code

As part of generating code for the decoders, MPACT-Sim now generates two additional pairs of files, `*_encoder.{cc,h}`, and `*_bin_encoder.{cc,h}`, which, as the names suggest, constitute the instruction and the binary field encoding.

The binary field encoding implements methods to encode opcodes and operand values into bit fields, even breaking an immediate value into a set of discontinuous fields as seen in many RiscV instruction encodings. The generated instruction encoding code consists of a "SlotMatcher" class that uses regular expressions generated from the disassembly specification to create a set of possible syntactic instruction matches, with string values for operands. It is supplemented by generated functions that attempt to encode each match based on the given operand strings. What is not generated is the code to translate the operand strings to values. This is where the user must write code, but thankfully, the number of operand types are limited, and are generally easy to parse and/or resolve. More specifically, the user must write a class that implements the `EncoderInterfaceBase` virtual interface described below.

EncoderInterfaceBase

The following shows the EncoderInterfaceBase for an ISA/bundle <Name>

```
class <Name>EncoderInterfaceBase {
 public:
  virtual ~<Name>EncoderInterfaceBase() = default;
  // Return the opcode encoding and size (in bits) of the opcode.
  virtual absl::StatusOr<std::tuple<uint64_t, int>> GetOpcodeEncoding(
      SlotEnum slot, int entry, OpcodeEnum opcode,
      ResolverInterface *resolver) = 0;
  // Return the encoding of the given source operand text.
  virtual absl::StatusOr<uint64_t> GetSrcOpEncoding(uint64_t address,
      absl::string_view text, SlotEnum slot, int entry, OpcodeEnum opcode,
      SourceOpEnum source_op, int source_num,
      ResolverInterface *resolver) = 0;
  // Append relocation entry for the source operand if it is relocatable.
  virtual absl::Status AppendSrcOpRelocation(uint64_t address,
      absl::string_view text, SlotEnum slot, int entry, OpcodeEnum opcode,
      SourceOpEnum source_op, int source_num, ResolverInterface *resolver,
      std::vector<RelocationInfo> &relocations) = 0;
  // Return the encoding of the given destination operand text.
```

```
  virtual absl::StatusOr<uint64_t> GetDestOpEncoding(uint64_t address,
      absl::string_view text, SlotEnum slot, int entry, OpcodeEnum opcode,
      DestOpEnum dest_op, int dest_num, ResolverInterface *resolver) = 0;
  // Append relocation entry for the destination operand if it is relocatable.
  virtual absl::Status AppendDestOpRelocation(uint64_t address,
      absl::string_view text, SlotEnum slot, int entry, OpcodeEnum opcode,
      DestOpEnum dest_op, int dest_num, ResolverInterface *resolver,
      std::vector<RelocationInfo> &relocations) = 0;
  // Return the encoding for the given source list operand text.
  virtual absl::StatusOr<uint64_t> GetListSrcOpEncoding(uint64_t address,
      absl::string_view text, SlotEnum slot, int entry, OpcodeEnum opcode,
      ListSourceOpEnum source_op, int source_num,
      ResolverInterface *resolver) = 0
  // Return the encoding for the given destination list operand text.
  virtual absl::StatusOr<uint64_t> GetListDestOpEncoding(
      uint64_t address, absl::string_view text, SlotEnum slot, int entry,
      OpcodeEnum opcode, ListDestOpEnum dest_op, int dest_num,
      ResolverInterface *resolver) = 0;
  // Return the encoding of the given predicate operand text.
  virtual absl::StatusOr<uint64_t> GetPredOpEncoding(uint64_t address,
      absl::string_view text, SlotEnum slot, int entry, OpcodeEnum opcode,
      PredOpEnum pred_op, ResolverInterface *resolver) = 0;
};
```

Let's take a closer look at this interface:

GetOpcodeEncoding(...)

This method is responsible for returning the binary encoding and size for the given opcode enum. It passes in the slot, the slot entry number, and a symbol resolver interface as well, but those are typically not needed for simple scalar architectures.

GetSrcOpEncoding(...)

This method is responsible for returning the binary encoding for the source operand `source_op`. Often no other information is needed. For some operands the provided instruction address is necessary to compute offsets/displacements, and/or the symbol resolver is required to look up the value of a symbol (e.g., branch destination). The additional parameters provide more context that might be useful for some ISAs.

An example would be to request the encoding of the source operand kRs1 (on RiscV) with the string "a1", returning the value 11 (a1 is equivalent to x11).

AppendSrcOpRelocation(...)

This method parses the text to determine if it refers to a symbol for which a relocation entry should be created for the given opcode and source operand. If so, it appends the relocation entry information to the relocations vector.

GetDestOpEncoding(...)

This method is similar to GetSrcOpEncoding, but applies to the destination operands.

AppendDestOpRelocation(...)

This method is similar to AppendSrcOpEncoding, but applies to the destination operands.

GetListSrcOpEncoding(...)

This method is currently not used, but is a placeholder for handling list based source operands (e.g., list of registers to save/restore to/from the stack).

GetListDestOpEncoding(...)

Similar to GetListSourceOpEncoding.

GetPredOpEncoding(...)

This method is similar to GetSrcOpEncoding, but for a predicate operand.

SlotMatcher

The <Name>SlotMatcher class is generated in the `*_encoder.{cc,h}` files for each instruction slot in the ISA definition.

```
class Riscv64xSlotMatcher {
 public:
  Riscv64xSlotMatcher(RiscV64XEncoderInterfaceBase *encoder);
  ~Riscv64xSlotMatcher();
  // Initialize the regular expressions.
  absl::Status Initialize();
  // Encode the given string, returning the encoding and size.
  absl::StatusOr<std::tuple<uint64_t, int>> Encode(uint64_t address,
     absl::string_view text, int entry, ResolverInterface *resolver);

 private:
  bool Match(absl::string_view text, std::vector<int> &matches);
  bool Extract(absl::string_view text, int index,
               std::vector<std::string> &values);
```

```
  RiscV64XEncoderInterfaceBase *encoder_;
  std::vector<RE2 *> regex_vec_;
  RE2::Set regex_set_;
  std::string args[3];
  std::array<RE2::Arg*, 3> re2_args = {nullptr, nullptr, nullptr};
};
```

The slot matcher takes the slot EncoderInterfaceBase class as an argument, as it provides the means for encoding operand strings. The method `Encode` is called by the architecture independent assembler to encode a string as an instruction.

The encode method uses `RE2::Set` to determine a set of potential instruction matches for the string. Then calls the Encode<Instruction> function for each such match until a successful match is made, or all matches fail. The Encode<Instruction> functions are generated, one for each opcode in the ISA slot. The function builds up the encoding of the instruction by calling the methods in the EncoderInterface for the opcode and each of the source and destination operands. If any of these calls fail, it returns a fail status.

Assembler Library

MPACT-Sim util/asm contains the main libraries for creating the final assembler. Currently, three classes are defined: OpcodeAssemblerInterface, ResolverInterface, and SimpleAssembler. These are described below.

OpcodeAssemblerInterface

```
class OpcodeAssemblerInterface {
public:
 virtual ~OpcodeAssemblerInterface() = default;
 // Takes the current address, the text for the assembly instruction, and a
 // symbol resolver interface.Return ok status if the text is successfully
 // encoded into the bytes vector.
 virtual absl::Status Encode(uint64_t address, absl::string_view text,
                             AddSymbolCallback add_symbol_callback,
                             ResolverInterface *resolver,
                             std::vector<uint8_t> &bytes) = 0;
};
```

The OpcodeAssemblerInterface defines a single method Encode that must be implemented by the user. It is called by the SimpleAssembler to encode the given text string for the instruction at the given address into a sequence of bytes, using the resolver to obtain the value of any symbol references that may occur.

The text string may span multiple lines of input if the line endings were escaped. This can be useful for instance, if the target ISA is a VLIW for which an instruction bundle may be specified across multiple lines. In this case it is up to the Encode function to properly parse and partition the text into its constituent single instructions and pass each such snipped to the appropriate slot matcher.
Additionally, the text may contain label definitions (labels start in the first position on a line) that have to be handled. Symbols for these labels are added by calling the `add_symbol_callback` function with the appropriate ELF values.

A typical implementation of the Encode method for a single slot architecture (like RiscV) is shown below.

```
absl::Status Encode(uint64_t address, absl::string_view text,
                    AddSymbolCallback add_symbol_callback,
                    ResolverInterface* resolver,
                    std::vector<uint8_t>& bytes,
                    std::vector<RelocationInfo>& relocations) override {
  // First check to see if there is a label, if so, add it to the symbol
  // table with the current address.
  std::string label;
  // label_re: "^(\\S+)\\s*:"
  if (RE2::Consume(&text, label_re_, &label)) {
    auto status = add_symbol_callback(label, address, 0, STT_NOTYPE, 0, 0);
    if (!status.ok()) return status;
  }
  auto res = matcher_->Encode(address, text, 0, resolver, relocations);
  if (!res.status().ok()) return res.status();
  auto [value, size] = res.value();
  union {
    uint64_t i;
    uint8_t b[sizeof(uint64_t)];
  } u;
  u.i = value;
  for (int i = 0; i < size / 8; ++i) {
    bytes.push_back(u.b[i]);
  }
  return absl::OkStatus();
}
```

ResolverInterface

The ResolverInterface provides a method by which the various operand encoding methods in the EncodingInterfaceBase derived classes can look up the value of a symbol. There are two implementations of this class in SimpleAssembler.

```
class ResolverInterface {
public:
 virtual ~ResolverInterface() = default;
 virtual absl::StatusOr<uint64_t> Resolve(absl::string_view text) = 0;
};
```

SimpleAssembler

The SimpleAssembler class provides a simple assembler that parses an input stream and creates an ELF64 non-relocatable, executable file with three sections (in addition to string and symbol tables): .text, .data and .bss in two segments. The constructor requires the appropriate values for the ELF os_abi, type, and machine, as well as the base address of the program segment, and a pointer to the OpcodeAssemblerInterface instance to use. For instance, for a 64 bit RiscV executable, the values ELFOSABI_LINUX, ET_EXEC and EM_RISCV would be appropriate.

To parse a file (or other stream), call the method Parse with an std::istream reference. If the parsing is successful, the method will return OkStatus. At this point, the ELF sections and segments have been populated. SimpleAssembler performs two passes over the input text. The first computes the sizes of the section and the location of the symbols. During this section a special ZeroResolver is used to resolve any symbol values. It returns 0 for any symbol that is queried, regardless if it is ever defined or not. During the second pass, this is replaced by a resolver that is linked to the ELF symbol table and will provide the accurate values, or a failure, if the symbol is not defined.

Next, the entry point (since this is a self contained executable) must be specified. Either as a text string containing a symbol defined in the assembly file, or an address.

Finally, call the method Write to write the ELF file to the given output stream.

Additionally, the ELF object can be accessed after Parse using the writer() accessor.

```
  SimpleAssembler(int os_abi, int type, int machine, uint64_t base_address,
                  OpcodeAssemblerInterface *opcode_assembler_if);
  // Parse the input stream as assembly.
```

```
absl::Status Parse(std::istream &is);
// Set the entry point. Either pass a symbol or an address.
absl::Status SetEntryPoint(const std::string &value);
absl::Status SetEntryPoint(uint64_t address);
// Write out the ELF file.
absl::Status Write(std::ostream &os);
// Accessor for the elf object.
ELFIO::elfio &writer() { return writer_; }
```

Supported Assembly Directives

The following assembly directives are supported in SimpleAssembler

- .align <N>
  Adds space to the current section to make sure it is aligned on an N byte boundary. N should be a power of 2.

- .bss
  Set the current section to .bss.

- .bytes <val> [,<val>]*
  Add the given byte values to the current section (not valid in bss).

- .char '<c>' [, '<c>']*
  Add the given characters to the current section (not valid in bss).

- .cstring "<string>" [, "<string>"]*
  Add the given strings as null-terminated strings to the current section (not valid in bss).

- .data
  Set the current section to .data.

- .global <name> [, <name>]*
  Set the listed symbols to have global binding.

- .long <val> [, <val>]*
  Add the given signed 64 bit values to the current section (not valid in bss).

- .short <val> [, <val>]*
  Add the given signed 16 bit values to the current section (not valid in bss).

- .space <N>
  Add N bytes of space to the current section (must be bss).

- .string "<string>" [, "<string>"]*
  Add the given strings to the current section without a null terminator (not valid in bss).

- .text
  Set the current section to .text.

- .uchar <val> [, <val>]*
  Add the given unsigned char values to the current section (not valid in bss).

- .ulong <val> [, <val>]*
  Add the given unsigned 64 bit values to the current section (not valid in bss).

- .ushort <val> [, <val>]*
  Add the given unsigned 16 bit values to the current section (not valid in bss).

- .uword <val> [, <val>]*
  Add the given unsigned 32 bit values to the current section (not valid in bss).

- .word <val> [, <val>]*
  Add the given signed 32 bit values to the current section (not valid in bss).

SimpleAssembler Class

The `SimpleAssembler` class in `mpact_sim/utils/asm` provides a higher level interface to the assembler functionality intended to be called from the assembler driver/main function. The public interface is shown below:

```
class SimpleAssembler {
public:
 SimpleAssembler(absl::string_view comment, int elf_file_class,
                 OpcodeAssemblerInterface *opcode_assembler_if);
 SimpleAssembler(const SimpleAssembler &) = delete;
 SimpleAssembler &operator=(const SimpleAssembler &) = delete;
 virtual ~SimpleAssembler();

 // Parse the input stream as assembly.
 absl::Status Parse(std::istream &is);
 // Add the symbol to the symbol table for the current section.
 absl::Status AddSymbolToCurrentSection(const std::string &name,
                                        ELFIO::Elf64_Addr value,
                                        ELFIO::Elf_Xword size, uint8_t type,
```

```
                                          uint8_t binding, uint8_t other);
 // Add the symbol to the symbol table for the given section.
 absl::Status AddSymbol(const std::string &name, ELFIO::Elf64_Addr value,
                        ELFIO::Elf_Xword size, uint8_t type,
                        uint8_t binding, uint8_t other,
                        const std::string &section_name);
 // Create an executable ELF file with the given value as the entry point.
 // The text segment will be laid out starting at base address, followed by
 // the data segment.
 absl::Status CreateExecutable(uint64_t base_address,
                               const std::string &entry_point);
 absl::Status CreateExecutable(uint64_t base_address, uint64_t entry_point);
 // Create a relocatable ELF file.
 absl::Status CreateRelocatable();
 // Write the ELF file to the given output stream.
 absl::Status Write(std::ostream &os);
 // Access the ELF writer.
 ELFIO::elfio &writer() { return writer_; }
…
};
```

The constructor takes the comment initializer, such as ';', '#' or '//', the elf file class, either ELFCLASS32 or ELFCLASS64 to signify either 32 or 64 bit ELF file, and finally a pointer to an object of a class that implements the OpcodeAssembler interface.

Once the class has been constructed, the user should set any ELF header values that are required. This is done using the ELF writer accessor. Values that may need to be set are the abi and machine values. An example for RiscV would be to set the os_abi to ELFOSABI_LINUX and the machine to EM_RISCV (macros from ELFIO). For example:

```
  assembler_ = new SimpleAssembler(";", ELFCLASS64, &riscv_64x_assembler_);
  assembler_->writer().set_os_abi(ELFOSABI_LINUX);
  assembler_->writer().set_machine(EM_RISCV);
```

After the constructor, the following classes have been created: .shstrtab, .symbtab, .strtab, .text, .data, and .bss classes. Additional classes may be created by the user through the ELFIO interface accessible on the same ELF writer object used to set ELF header values.

Special symbols can be added using the AddSymbol(...) methods.

Once the global initialization has been performed, the user calls Parse() with an open input stream for the assembly source code. Parse will process the assembly source and return OkStatus if no errors occurred.

Provided that no errors occurred, the user may now create either a relocatable or executable ELF object. A relocatable ELF object will require linking to produce an executable that can be run on a simulator or hardware. The executable is a full standalone, ready to run executable ELF object. In order to create an executable, two values have to be provided: the base address of the text (instructions) segment and the program entry point. The base address is used to set the load address of the text segment in memory. The data segment will have a load address right after the text segment (subject to alignment restrictions). Creating a relocatable object requires no values to be specified.

Finally, the user calls Write() with an open output stream to which the ELF object should be written.