FrASM

Pierre-Marie de Rodat

pmderodat@lse.epita.fr
http://lse.epita.fr

July 16, 2012

What is an assembler?

- A processor executes instructions
- Such instructions are stored in binary files
- Sometimes, those binaries are more than just concatenation of instructions
- Writing and reading binary files is not handy
- An assembler translates an easy to read/write source file to such a binary one

Why developping an assembler?

- Working on unusual architectures can happen :
  - Reverse engineering some mysterious hardware
  - Working on "software" ISA (CTF, software protection, ...)
- If an assembler exists for some case, it might not be available
- Then one may want to write one not to bother with raw binary

Alright, then what is FrASM?

- FrASM is a framework that aims to ease the task of creating assemblers
- It is written using Python3 and depends on two pure-python external libraries (Lexy and funcparselib)

Assembler steps

What a piece of software!

- Lexer
- Parser
- Use mnemonics and operands to determine opcode
- Sort by section
- Resolve symbols
- Encode opcodes
- Write output binary

Most of these steps are the same amongst assemblers.

Parsing subtelties 1

- `mov r1, 0x12`

  Basically, `instruction = opcode operand1, operand2, ...`

- `trap nop`

  Newline is meaningful to recognize `opcode operand` and `opcode1 opcode2`

Parsing subtelties 2

- The framework cannot know what operands will look like
- Examples: memory accesses
- `mov r0, [r1]`
- `mov r0, 4(r1)`
- `mov r0, (r1, r2)`
- ...
- Users must be able to extend the operands grammar
- The parser must cope with non-LL(1) grammars (cf. operands)

Parsing subtelties 3

- Once mnemonics and operands have been parsed instructions must be "selected"
- *But,* there is no "direct" relation
- Two different opcodes can have the same mnemonic, the same number of operands and the same kind of operands:

```
ld i, 0x28
ld v1, 0xf3
```

- The parsing/selection of instructions is not straightforward

Interface in brief

Users have to:

- Provide a list of allowed sections (if any)
- Provide a list of allowed data types: those can be used to insert data in sections (words, random bytes, strings, ...)
- Define methods to locate sections and to write output binary
- Optionally override default parameters (e.g.: allow undefined symbols, ...)
- Give a list of instructions and associated opcodes

Lexer/Parser

- A simple-to-use and runtime-configurable lexer: Lexy
- Parsers are also built at runtime: funcparserlib (parser combinators)
- ... not quite appropriate for non-LL(1) grammars, but this may change in the future

Data types

```
.string "hello world!"
.byte 0xff
.timestamp 2012 07 16 18 28
```

- A data type is a data structure that must define:
  - a list of tokens to parse as arguments
  - a validation method (called right after parsing)
  - an encoding method.
- Using this, data can be inserted anywhere in assembled binary.

Opcodes

An opcode kind is a data structure that must define:

- A mnemonic
- An operands compatibility checking method
- An encoding method

Operands (1/2)

- This is the hard part :
  - operands parsing cannot rely on opcode information since parsed operands are used to determine the opcode.
  - Many operand kinds can share the same parsers
- Parsing and conversion mechanisms are separated
- This way, parsers can easily be shared, and there is no ambiguity at parsing time
- This only requires a little bit more work in the framework to collect parsers from operands in opcodes and to dispatch parsed instructions to correct opcodes.

Operands (2/2)

- To extend builtin operands, first define a parser (using funcparserlib)
- An operand parser is a named tuple that contains the funcparserlib parser itself and a hash (to uniquely identify the parser)
- The funcparserlib parser must return another tuple containing the location, the parser itself and the parsed value
- Then, an operand type data structure must be defined, and it must have:
  - a name
  - a set of acceted parsers
  - a method to check the validity of parsed operands
  - a method to convert parsed operand to bytes

Around

- Most FrASM users will just assemble sources using their assembler.
- For them, the framework define an optional entry point that just do the job (read source, launch assembler, write output).

Data types

```
from frasm.data import BaseDataType

class StringType(BaseDataType):
    def __init__(self):
        super(StringType, self).__init__('string', string)

    # There is no data to validate: every string is valid
    def validate(self, string):
        pass

    def encode(self, string):
        return string.encode('latin-1')
```

```
.string "hello world!"
```

Operands

```
from frasm.operand import (
    RegisterOperand, Immediate, MemoryOperand)

# Address on 12 bits
addr_op = ImmediateOperand(size=12, signed=False)
# 8 and 4 bits constants
byte_op = ImmediateOperand(size=8, signed=False)
nibble_op = ImmediateOperand(size=4, signed=False)
# Syntax: an immediate can be an number or a name (label)

# Any general purpose register: v0-vf
gp_reg_op = RegisterOperand(
    {'v{}'.format(i): int(i, 16) for i in '0123456789abcdef'},
    size=4)

# Memory access register
i_op = RegisterOperand({'i': -1}, ignored=True)
indirect_i_op = MemoryOperand(i_op, ignored=True)
# this can handle "[i]"
```

Opcodes (1/2)

```python
from frasm.opcode import BaseOpcodeType

class Opcode(BaseOpcodeType):
    # Mapping operand size -> operand mask
    operand_masks = {i: 2**i - 1 for i in (4, 8, 12)}
    opcode_struct = struct.Struct('>H')

    def __init__(self, mnemonic, base_opcode,
            *args, **kwargs):
        # Every instruction is 2 bytes long
        kwargs['size'] = 2
        super(Opcode, self).__init__(mnemonic,
            *args, **kwargs)
        self.base_opcode = base_opcode

    def encode(self, *operands):
        operands_bits = # ... combine given operands
                        # into a bit field
        return self.opcode_struct.pack(
            self.base_opcode | operands_bits)
```

Opcodes (2/2)

```
opcodes = (
    Opcode('sys',   0x0000, addr_op),
    Opcode('cls',   0x00e0),
    Opcode('ret',   0x00ee),
    Opcode('jp',    0x1000, addr_op),
    Opcode('call',  0x2000, addr_op),
    Opcode('se',    0x3000, gp_reg_op, byte_op),    # !
    Opcode('sne',   0x4000, gp_reg_op, byte_op),
    Opcode('se',    0x5000, gp_reg_op, gp_reg_op), # !
    Opcode('ld',    0x6000, gp_reg_op, byte_op),
    Opcode('add',   0x7000, gp_reg_op, byte_op),
    # ...
)
```

Assembler

```python
from frasm.assembler import BaseAssembler

class Assembler(BaseAssembler):
    def locate_sections(self):
        text = self.sections['text']
        data = self.sections['data']
        text.set_address(0x200)
        data.set_address(text.address + text.size)

    def write_binary(self, fp):
        self.sections['text'].write(fp)
        self.sections['data'].write(fp)
```

Putting all together

```
import sys
# lexy & frasm imports

# Opcode type class
# Data type classes
# Operand definitions
# Opcode definitions

assembler = Assembler(
    sections = ('text', 'data'),
    data_types = (ByteType(), StringType()),
    opcodes = opcodes,

    warning_lazy_opcode_size = True,
)

if __name__ == '__main__':
    main(assembler, sys.argv[1:])
```

Example Chip-8 source : UFO

```
.section text

entry_point:
    ld i, 0x2cd
    ld v9, 0x38
    ld va, 0x8
    drw v9, va, 0x3
    ld i, 0x2d0
    ld vb, 0x0
    ld vc, 0x3
    drw vb, vc, 0x3
    ld i, 0x2d6
    ld v4, 0x1d
    ld v5, 0x1f
    drw v4, v5, 0x1
    ld v7, 0x0
    ld v8, 0xf
    call @some_function
; ...
```

Temporary weaknesses (1/2)

- Opcode/operands/data type definitions: there is only the core, currently no "library sugar"
- I miss tests!!!!!!!!
- The operand subsystem is still too complex, and there are some workaround with funcparserlib
- Users have to deal with Lexy and funcparserlib directly
- It might be useful to improve data types parsing system

Temporary weaknesses (2/2)

- Overall parsing capabilities need improvement for complex ISAs (VLIW, ...)
- The formalism (sections that contain opcodes and data) might not be appropriate for very exotic architectures
- Working with more and more architectures should be used to improve the overall formalism to match as many systems as possible

Weaknesses that might not change

- "Hey, I cannot master completely the syntax!"
- This may be the price for the generic parser
- Since parsing is easy, conversion tools to more specific syntax should be easy to write using existing codebase...
- ... production assemblers are expected to be as fast as possible?

Questions?

- pmderodat@lse.epita.fr
- PM @ rezosup
- http://git.lse.epita.fr/?p=frasm.git