Mya Specification

Luiz Felipe Silva (Silva97)

January 13, 2025

Abstract

Mya (acronym for "Make Your Assembler") is a formal language to write specifications of an ISA (Instruction Set Architecture).

Contents

1 Syntax notation

Wirth syntax notation (WSN) of Mya language:

```
PROGRAM = { DECLARATION | COMMAND } .

DECLARATION = BITFIELD_DECLARATION
            | REGISTER_DECLARATION
            | INSTRUCTION_DECLARATION .

BITFIELD_DECLARATION = "bitfield" BITFIELD_NAME SIZE_SPEC [ "{"
BITFIELD_BODY "}" ] .
BITFIELD_NAME = UPPERCASE_LETTER { ALPHACHARACTER } .
BITFIELD_BODY = BITFIELD_FIELD_DECLARATION {
BITFIELD_FIELD_DECLARATION } .
BITFIELD_FIELD_DECLARATION = IDENTIFIER SIZE_SPEC .
BITFIELD_SPEC = BITFIELD_NAME "{" ( EXPRESSION |
BITFIELD_SPEC_FIELD { "," BITFIELD_SPEC_FIELD } ) "}" .
BITFIELD_SPEC_FIELD = IDENTIFIER "=" EXPRESSION [ "," ] .

REGISTER_DECLARATION = "register" IDENTIFIER SIZE_SPEC "="
BITFIELD_SPEC .

INSTRUCTION_DECLARATION = "inst" IDENTIFIER SIZE_SPEC "("
INSTRUCTION_ARGLIST ")" "{" INSTRUCTION_SPEC "}" .
INSTRUCTION_ARGLIST = INSTRUCTION_ARG { "," INSTRUCTION_ARG } .
INSTRUCTION_ARG = IDENTIFIER ":" TYPE_SPEC .
INSTRUCTION_SPEC = INSTRUCTION_SPEC_FIELD { ","
INSTRUCTION_SPEC_FIELD } .
INSTRUCTION_SPEC_FIELD = IDENTIFIER "=" BITFIELD_SPEC [ "," ] .

TYPE_SPEC = TYPE_NAME SIZE_SPEC .
TYPE_NAME = "register" | "immediate" .

SIZE_SPEC = "[" EXPRESSION "]" .


COMMAND = COMMAND_STATEMENT ";" .
COMMAND_STATEMENT = SET_COMMAND | INCLUDE_COMMAND .

SET_COMMAND = "set" IDENTIFIER "=" EXPRESSION .
INCLUDE_COMMAND = "include" STRING .


IDENTIFIER = LETTER { ALPHACHARACTER } .
EXPRESSION = IDENTIFIER
```

```
           | NUMBER
           | "(" EXPRESSION ")"
           | EXPRESSION OPERATOR EXPRESSION .
OPERATOR = "-" | "+" | "/" | "*" | "|" | "&" | "^" | "~"  | "<<" |
">>" .


UPPERCASE_LETTER = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" |
"I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" |
"T" | "U" | "V" | "W" | "X" | "Y" | "Z" .
LOWERCASE_LETTER = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
"i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" |
"t" | "u" | "v" | "w" | "x" | "y" | "z" .
LETTER = UPPERCASE_LETTER | LOWERCASE_LETTER .
ALPHACHARACTER = LETTER | DECIMAL_DIGIT | "_" .

NUMBER = DECIMAL_NUMBER | HEXADECIMAL_NUMBER | OCTAL_NUMBER |
BINARY_NUMBER .
DECIMAL_NUMBER = DECIMAL_DIGIT { DECIMAL_DIGIT } .
HEXADECIMAL_NUMBER = "0x" HEXADECIMAL_DIGIT { HEXADECIMAL_DIGIT } .
OCTAL_NUMBER = "0o" OCTAL_DIGIT { OCTAL_DIGIT } .
BINARY_NUMBER = "0b" BINARY_DIGIT { BINARY_DIGIT } .


DECIMAL_DIGIT = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
"8" | "9" .
HEXADECIMAL_DIGIT = DECIMAL_DIGIT
                  | "a" | "b" | "c" | "d" | "e" | "f"
                  | "A" | "B" | "C" | "D" | "E" | "F" .

STRING = """" { ANY_CHAR } """" .
```

1.1 Code example

```
include "registers.mya";

bitfield Reg[4]

bitfield Opcode[8] {
    imm[1]
    op[7]
}

register r0[32] = Reg{0}
register r1[32] = Reg{1}
register r2[32] = Reg{2}
```

```
register r3[32] = Reg{3}
register r4[32] = Reg{4}
register r5[32] = Reg{5}
register r6[32] = Reg{6}
register r7[32] = Reg{7}

# Internal rules to avoid errors.
set INSTRUCTION_MAX_SIZE = 16;
set INSTRUCTION_MIN_SIZE = 16;

# Assembly example: mov r1, r2
inst mov[16](arg1: register[32], arg2: register[32]) {
    opcode = Opcode {
        imm = 0b0,
        op = 0x00,
    }, # It's equivalent to: Opcode{0}
    reg1 = Reg{arg1},
    reg2 = Reg{arg2},
}
```

2 Declarations

2.1 Bitfield

Bitfields are the representation of how the values (like registers) are specified on the machine code of the ISA. The syntax to declare a bitfield is:

```
bitfield <name>[<size>]

OR

bitfield <name>[<size>] {
    <field-list>
}
```

- `<name>` should start with an uppercase letter followed by any combination of `[a-z][A-Z][0-9]_` characters.
- `<size>` is a literal number that specifies the size of the bitfield in bits.
- `<field-list>` is a list of bitfield's field names and sizes[1].

2.1.1 Examples

```
bitfield Reg[4]
```

It's a bitfield named `Reg` with 4 bits size.

```
bitfield Opcode[8] {
    imm[1]
    op[7]
}
```

It's a bitfield named `Opcode` with 8 bits size and 2 fields:

1. `imm` (1 bit size) is the first bit of the bitfield `Opcode`.
2. `op` (7 bits size) are the next 7 bits of the bitfield `Opcode`.

[1] The sum of all field sizes should be equal to bitfield's size.

2.2 Register

Registers of the ISA can be declared specifiying the bitfield where the register code is set. The syntax to declare a register is:

```
register <name>[<size>] = <bitfield-specification>
```

- `<name>` should start with a letter followed by any combination of `[a-z][A-Z][0-9]_` characters.
- `<size>` is a literal number that specifies the size of the register in bits.
- `<bitfield-specification>` is the specification to what bitfield is used to set this register code and what value is set on this bitfield to specify the usage of this register.

2.2.1 Examples

```
bitfield Reg[4]

register r2[32] = Reg{2}
```

It's a 32 bit register named `r2` where they code is set on a `Reg` bitfield, and it's code is `2`.

```
bitfield Reg[4] {
    size[1]
    code[3]
}

register rdx[64] = Reg {
    size = 1,
    code = 2,
}
```

It's a 64 bit register named `rdx` where they code is set on a `Reg` bitfield, and the bitfield's fields are set to `size = 1` and `code = 2` respectivaly. It's equivalent to `Reg{10}`.

2.3 Instructions

ISA's instructions are declared specifying it's arguments and machine code format. The syntax is:

```
inst <name>[<size>](<arglist>) {
    <instruction-specification>
}
```

- `<name>` should start with a letter followed by any combination of `[a-z][A-Z][0-9]_` characters.
- `<size>` is the size in bits of the instruction.
- `<arglist>` is a command separated list of arguments that the instructions expects.
- `<instruction-specification>` is a comma separated list of bitfields in the instruction.

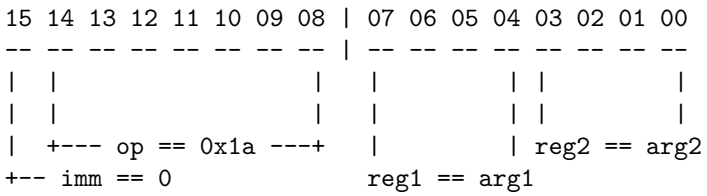
2.3.1 Examples

```
bitfield Reg[4]

bitfield Opcode[8] {
    imm[1]
    op[7]
}

register r0[32] = Reg{0}
register r1[32] = Reg{1}
...
register r15[32] = Reg{15}


inst mov[16](arg1: register[32], arg2: register[32]) {
    opcode = Opcode {
        imm = 0b0,
        op = 0x1a,
    },
    reg1 = Reg{arg1},
    reg2 = Reg{arg2},
}
```

This specify an instruction named `mov`, 16 bits size, that expects two 32 bit registers as arguments. On the assembly perspective, this instruction looks like:

```
mov <reg32>, <reg32>
```

The machine code format is specified on the body of the instruction, where it's uses a sequence of one `Opcode` bitfield and two `Reg` bitfields. Having the format like:



Example:

```
Assembly:         mov r2, r10
Hex machine code: 1a 2a
Bin machine code: 00011010 00101010
```

3 Commands

Commands are executed at parse-time. The generic syntax to a command is:

```
<keyword> <command-specific-syntax> ;
```

All commands ends with a semicolon.

3.1 include

The `include` command includes another module content on the same position where it's command is used. The included module is parsed and executed in the same time at the `include` command is executed. The syntax is:

```
include "<module-path>";
```

- `<module-path>` is the relative or absolute path to module's file to include, using `/` as directory separator. A path starting with `/` means an absolute path, where it's start on the current filesystem root. Relative paths will be relative starting from a common path where it's considered the "current working directory", and not relative from where the module is.

3.1.1 Examples

```
include "modules/registers.mya";
include "config.mya";
include "/etc/mya/modules/common.mya";
```

3.2 set

The `set` command sets the value of a global variable, that could be used on any expression. The syntax is:

```
set <name> = <value>;
```

- `<name>` is the name for the variable to change/create. Should start with a letter followed by any combination of `[a-z][A-Z][0-9]_` characters.
- `<value>` is any valid expression to be evaluate as the variable's value.

3.2.1 Examples

```
set A = 2;
set B = A + 5;
```