Generic Assembler Manual

Nishad Mathur

March 5, 2016

Contents

1 General Overview

1.1 About the Assembler

In brief, the purpose of this program is to provide a re-configurable assembler.

It provides 2 mechanisms through which this may be accomplished, firstly, you may use the YAML [1](a textual serialization format) configuration format in conjunction with the command line tool or you may use the system in a library format, configuring the assembler in code using it in a DSL (Domain Specific Language) like format. This manual is concerned primarily with the prior and will not cover the DSL format in any great detail.

1.2 About the Manual

I have tried to keep the manual fairly informal and brief, listing out the important information and adding (hopefully) illustrative examples along the way. After reading the 5000 page ARM architectural review manual I've gotten thoroughly sick of the dry manuals provided there, so this is my (admittedly poor) response to that.

I don't intend that you read the whole manual in one go, it may be worth skimming over the errors list 5 and the file-structure 4 and looking over them as you need them. The examples should illustrate the concepts in use if that is your preferred learning style.

There is a full example in the appendix which could be an interesting starting point. It is a mash-up of several architectures, so isn't useful on its own but it should give you a useful starting point and an idea of how it should be structured.

2 Running the Command-line Tool

#TODO update the CLI then update this document. Add help, output file, listings, etc.

Running the tool via command-line is a fairly trivial exercise the basic format of the tool is to use "`java -jar generic-assembler --input banana.asm --config lanugag`
which will run the tool, parse the configuration and then output the compiled file to standard-out. If no '–input' or '–output' parameters are defined, then standard-in and standard-out are used for the input and output respectively.

–help Provides a help listing of all commands, similar to this.

–input Provides the path to an assembly file to compile.

–output Provides the path to a destination file for the compiled file

–listings Outputs the listings instead of the assembly file

–config Provides the path to the configuration file.

```
// Ussembly!
// The impossible has happened, now it falls
// upon you to make sure the world knows it...
// Prepare the world!

hell:
    freeze water
    put pig0, #wings[0]
    goto #hell

animal:
    .data 0xDEADBEEF
```

Figure 1: The file (which will control the world, colloquially known as our assembly language file)!

3 Tutorial - uSemmbler for uSembly

The language is very simple, it has 3 operands, ”freeze”, ”put” and ”goto” as well as a special command called ”.data”. Not much you can do with it, but describe a lovely summers day in hell. It also has a couple of registers, called pig0 to pig 3 and water.

3.1 Getting Started

First things first, we need a couple of files to start with; lets call them `ussembly.yaml` for the assembly description file and `test.asm` for the assembly file.

Then go ahead any copy the contents of figure 1 into the `test.asm` file.

```
configuration:

references:

instructions:
```

Figure 2: The skeleton configuration file.

Next, copy the contents of figure 2 into the `ussembly.yaml` and save that.

Then we can go ahead and try that using the command `jav -jar assembler --config uSsembly.`
(it should throw up an error).

3.2 Filling in the Configuration Block

So lets go ahead and flesh out that configuration block the error was complaining about. But nefore we start, I will explain what exactly it is: the configuration block is where all the global settings go, the settings which don't fit else where or effect everything.

```
configuration:
  line can start with label: true
  label bit size: 4
  word size: 16
  label regex: "^(\\w+)"
  argument separator: ",\\s*"
  comment regex: "//.*"
```

Figure 3: The Configuration Block

I'll quickly go through an explain each bit for you:

line can start with label asks whether or not a line with an instruction on it can start with a label.

label bit size the maximum number of bits a label address can take up (you can reduce this on a case by case basis).

word size declares how long a word is (this is used to calculate offset sizes for labels).

label regex asks for a regular expression in which the first group captures a label's identifier.

argument separator asks for a regex which is used to split the arguments of an instruction up.

comment regex captures a whole comment into a regular expression.

There are also, some optional options, relating to endianess, but I will leave those be for now.

Now the assembler knows how big a word is, how big a label is, what a comment looks like, what an argument looks like and what a label looks like and honestly, that's all it needs to know here.

3.3 Filling in the References Block

Now it should be complaining that the references block is missing or malformed (mostly missing in this case), so lets get started on filling that in.

Lets start with the registers then, shall we? The first part of this block should look like this:

```yaml
references:
  - name: register
    kind: meta
    references:
      - name: filthy capitalist pigs
        kind: mapped
        size: 4
        range:
          format string: "pig{0}"
          start index: 0
          start literal: 0
          count: 4
      - name: things
        kind: mapped
        size: 4
        mappings:
          water: 0x4
```

Figure 4: The References Block - Registers

The gist of this one is that we have a bunch of registers, all of which can be refereed by their collective name of ”register” (or if your feeling a little bored, ”filty capitalist pigs”). Under this group are two sub groups ”things” and ”filthy capitalist pigs”, where there is one register under things (called water, with the value of 0x4) and 4 registers under piggies (call pig0 with the value of 0x0 through to pig3 with the value 0x3).

If you’re curious, the format string option expects a format string where the first item is the replaced value see the java MessageFormat docs [3] for more details.

Then we add the literals(figure 5): the numbers (like 1, 2, 100...) and the hex (like 0x0, 0x3, 0xFFFF...) literals. Here we have the literals grouped under a group called ”literal” (you could call it plebs if you prefer, the name is arbitrary) and we have 2 types of literal underneath it, hex literals and integer literals. Their options are fairly self, explanatory (but if not check

```
- name: literal
  kind: meta
  references:
    - name: hexadecimal
      kind: literal
      literal type: HEXADECIMAL
      literal size: 4
      validation regex: "0x\\d+"
      extraction regex: "0x(\\d+)"

    - name: int4
      kind: literal
      literal type: INTEGER
      literal size: 4
      validation regex: "\\d+"
      extraction regex: "(\\d+)"
```

Figure 5: The References Block - Literals

out section 4).

```
- name: label
  kind: label
  size: 4
  validation regex: "^#\\w+$"
  extraction regex: "\\w+"
```

Figure 6: The References Block - Label

This ones (figure 6) just simple, if you need to reference a label, then this describes their format, a string of characters which starts with a '#'.

This one (figure 7) is paradoxically both fairly simple and a little tricky to understand (these names may change in the future). So this is what each bit means:

source before offset controls whether the default byte order for this has the source (lhs) output before the offset (rhs) or not.

regex has 2 capture groups, the first for the source, the second for the offset.

```
  - name: memory
    kind: indexed
    source before offset: true
    regex: "(.*?)\\[(.*?)\\]"
    valid left hand types:
      - label
    valid right hand types:
      - literal
```

Figure 7: The References Block - Memory Access

valid [left—right hand types] declares which types (i.e. the references we defined earlier) can be embedded on each side.

And that is each of the bits. I will put the whole thing at the bottom, if you've had any issues, but it should be straight forward to put these pieces together.

3.4 Instruction Block

And *finally* we have the instructions them selves.

Before we actually get started on this, I'm going to go ahead and specify these properly for you.

freeze has 1 argument, a register. It's binary form has these value:

1. 4 bits identifier: 0x1
2. 4 bits register
3. 8 bits padding: 0x00

put has 2 arguments, a register and a memory access with a label and a literal offset.

1. 4 bits identifier: 0x2
2. 4 bits destination register
3. 4 bits label
4. 4 bits literal offset

goto has 1 argument, a label

1. 4 bits identifier: 0x1

2. 4 bits padding: 0x00
3. 4 bits label
4. 4 bits padding: 0x00

.data has 1 or 2 arguments, each is a literal

1. 4 bits literal

3.4.1 put

So lets start of with the conceptually simplest one ”put”. This one is straight forwards and is implemented as in figure 8. You just put the sentinel using ”byte sequence” and specify how many bits long it is. Then the arguments are extracted and then synthesized in the order in which they are defined here.

```
  - name: put
    byte sequence: 0x02
    size: 4
    arguments:
      destination: register
      value: memory
```

Figure 8: The Instructions Block - Put

3.4.2 freeze

This instructions is probably the most complicated of the 4 to implement. Mostly, because of those padding requirements. So this one differs in that you declare each literal or argument and the number of bits it takes up.

3.4.3 .data

This one is *really* easy. It doesn’t have any sentinels, or anything, it just has 1 catch, you can have 1 or 2 arguments. The solution though, is simple, remember how we had multiple registers under the same name? Well we can do the same here, with a meta block. Check out figure 10 to see how i solved it.

```
  - name: freeze
    arguments:
      thing to freeze: register
    byte sequence:
      - literal: 0x1
        size: 4
      - path: thing to freeze
        size: 4
      - literal: 0x0
        size: 8
```

Figure 9: The Instructions Block - Freeze

3.4.4 Goto

This instruction should be very similar to freeze, so I'm going to use this as an opportunity to let you try this you self, and then come back and check it. I've put my version (not the only answer!) into figure 11.

3.5 Running it!

Now you should have it all put together and ready to go. Go ahead and try run it using: `jav -jar assembler --config uSsembly.yaml --input test.asm --listings`

If it all goes well you should get something out at the end which looks like figure 12. If it doesn't look like that, then go ahead and look in the co

```
- name: .data
  kind: meta
  instructions:
    - name: .data1
      byte sequence: 0x0
      size: 0
      arguments:
        data0: literal
    - name: .data2
      byte sequence: 0x0
      size: 0
      arguments:
        data0: literal
        data1: literal
```

Figure 10: The Instructions Block - Data

```
- name: goto
  arguments:
    label: label
  byte sequence:
    - literal: 0x3
      size: 4
    - literal: 0x0
      size: 4
    - path: label
      size: 4
    - literal: 0x0
      size: 4
```

Figure 11: The Instructions Block - Goto

```

```

Figure 12: The Assembled File

4 File Structure

First things first; the most important thing to understand about the configuration format is that it is just plain old YAML, so any of the clever things you can do there, you can do here so don't forget that fact and use it where you find it helpful. For example you can use anchors and aliases to deduplicate blocks of code.

Secondly, I've tried to give helpful error messages in the assembler, but errors in your configuration file can manifest in weird, wonderful and not entirely obvious ways, so remember to keep an eye out for that.

The basic gist of the file its self is that it is a YAML file (hence the '.yaml' extension) which is passed into the assembler, loaded and built into an assembler configuration. If you find it limiting, you can always extend it yourself, using the assembler as a library, but that isn't covered in this manual (your best bet there is to check out 'com.nishadmathur.main').

The file its self is composed of four top level blocks: 'configuration', 'references', 'instructions' and (optionally) 'instruction formats'.

4.1 Common Information

The intent with this configuration was to maintain consistent formats, names and levels of flexibility throughout and as a consequence there are several conventions (for the format and YAML) which are helpful to know when creating a custom configuration.

4.1.1 YAML

YAML (a recursive acronym, YAML Ain't a Markup Language)[1] is a human readable serialization/markup format optimized for readability, consistency and simplicity. It is format is very convention driven and many of those are enforced by the parser. The more fundamental of which are explained below.

4.1.2 Configuration Conventions and Requirements

Not all of these are hard requirements, but they are strongly suggested and erring from them is not thoroughly tested.

2 Space Indentation is strongly suggested format, tabs are not supported for indentation and using 2 spaces per indent level aligns neatly with the array syntax, so is the suggested depth.

Dictionary keys and values for example names, such as path and the path value may contain an arbitrary name, including spaces and punctuation, and this is the suggested format as it improves readability and expressiveness of the language.

Naked Strings are the preferred form of string, avoid using the enclosing quotes except where necessary. Certain strings may require escaping or embedding in a string literal (see the YAML documentation for further details [2]).

Space Between Sections is a suggestion which more clearly delineates each large block from one another allowing the user to better skim over the format.

Use Multiline Dictionaries and avoid dictionary literals. It is also preferred to start the dictionary on the same line as the list (e.g. `- key: value`), if the dictionary is nested in a list.

4.2 Configuration

```
configuration:
  line can start with label: true
  label bit size: 16
  word size: 16
  label regex: "^(\\w+)"
  argument separator: ",\\s*"
  comment regex: ";.*"
```

This block sets up the global configurations for the assembler:

line can start with label : This switch controls whether or not the first character of the line is defined as a label.

label regex : This regular expression is used to identify and extract a label from a line, it is called on every line and the first capture group in the regex should contain the labels name.

argument separator : The instructions are split on this regular expression, so for example 'jmpf cond label' would be split on `"\^(\\w+)"` into the components 'cond' and 'label'.

comment regex : This expression should have a single capture group which matches on a comment.

word size : This #TODO find out wtf this is used for.

label bit size : This is used to control the default size of a label reference.

4.3 References

The references section defines the types which are essentially analogous variable types.

```
references:
  - name: literal
    kind: meta
    references:
      - name: literal4
        kind: meta
        references:
          - name: hexadecimal
            kind: literal
            literal type: HEXADECIMAL
            literal size: 4
            validation regex: "0x\\d+"
            extraction regex: "0x(\\d+)"

          - name: int4
            kind: literal
            literal type: INTEGER
            literal size: 4
            validation regex: "\\d+"
            extraction regex: "(\\d+)"

  - name: registers
    kind: meta
    references:
    - name: general registers
      kind: meta
      references:
      - name: 64bit general registers
        kind: mapped
        size: 4
        range:
          format string: "[X|x|r]{0}"
          start index: 0
          start literal: 0
          count: 32

    - name: special registers
      kind: mapped
      size: 4
      mappings:
        PC: 0
        SP: 0
        WSP: 0
        ELR: 0

  - name: label
    kind: label
    size: 16
    validation regex: "^\\w+$"
```

Meta : Following on from the type analogy, meta types are akin to an interface, every child type (transitively) conforms to the type. For example, literal4 is a literal and int4 is a literal4 and a literal.

```
references:
  - name: literal
    kind: meta
    references:
      - name: literal4
        kind: meta
        references:
          - name: hexadecimal
            kind: literal
            literal type: HEXADECIMAL
            literal size: 4
            validation regex: "0x\\d+"
            extraction regex: "0x(\\d+)"

          - name: int4
            kind: literal
            literal type: INTEGER
            literal size: 4
            validation regex: "\\d+"
            extraction regex: "(\\d+)"
```

Mapped : This type exists for named constants, for example for registers. As a note, there are two approaches to defining a mapped type.

```
references:
  - name: registers
    kind: meta
    references:
    - name: general registers
      kind: meta
      references:
      - name: 64bit general registers
        kind: mapped
        size: 4
        range:
          format string: "[X|x|r]{0}"
          start index: 0
          start literal: 0
          count: 32

    - name: special registers
      kind: mapped
      size: 4
      mappings:
        PC: 0
        SP: 1
        WSP: 2
        ELR: 3
```

The first is where you define the names of a mapping as a regular expression and the literal values associated with each directly. The alternative approach is to define a 'range' block, where you define the base case for the format-string and the literal value and the number of constants that are defined. Each range mapped value has the value incremented by one.

One point to note, the format string option expects a format string where the first item is the replaced value [3].

Indexed : Index is a compound type designed specifically for label + offset references but can be re-purposed for other binary operations. The field 'source before offset' defines whether the second field (the offset field) goes first in the binary form, by default it is placed after, but this reverses it. 'valid left hand types' and 'valid right hand types' allow you to list out which types are valid in the left and right hand fields respectively.

```
  - name: memory
    kind: indexed
    source before offset: true
    regex: "(.*?)\\[(.*?)\\]"
    valid left hand types:
      - literal16
      - label
    valid right hand types:
      - registers
```

Label : This essentially defines the users label reference, which differs from the label declaration (defined in the configuration block). This for example may be used to reference a location for a jump instruction ('jmp #loop') and the offset is later calculated and the label is substituted by the binary form of the offset value.

```
  - name: label
    kind: label
    size: 16
    validation regex: "^\\w+$"
    extraction regex: "\\w+"
```

Literal : This is the other primitive type for the assembler, this one is used for literal references, for example you would use it for your integer type '1' or your hexadecimal type '0x7d' (among others).

For The 'literal type' field, you must select a type from 'BINARY', 'HEXADECIMAL' or 'INTEGER'. Each of which corresponds to the obvious formats, base 2, base 16 and base 10. These are converted directly to their binary form, so '0xB' would convert to '1011'. Binary follows the standard 2s compliment format and binary is a direct conversion. #TODO look into replacing this with a base command instead.

```
references:
- name: int4
  kind: literal
  literal type: INTEGER
  literal size: 4
  validation regex: "\\d+"
  extraction regex: "(\\d+)"
```

4.4 Instructions

Instructions are the bread and butter for an assembler, hence defining the instruction correctly properly is key to producing a working assembler. There are 2 primary concepts at play here instruction normal instruction definitions and meta instructions. 'Normal' instruction definitions encompass an instruction which is selected solely on its name (although it is still type checked) and 'Meta' instructions are instructions which can have multiple different forms, where the layout depends on the types of its arguments as well as its name.

```yaml
- name: add
  byte sequence: 0x00
  size: 4
  arguments:
    destination: register
    lhs: register
    rhs: register

- name: mov
  kind: meta
  instructions:
    - name: mov rr
      byte sequence:
        - path: destination
          size: 5
        - literal: 0
          size: 11
        - path: source
          size: 5
        - literal: 0x2A8
          size: 11
      arguments:
        destination: 64bit general registers
        source: 64bit general registers

    - name: mov rl
      byte sequence:
        - path: destination
          size: 5
        - path: literal
          size: 16
        - literal: 0x694
          size: 11
      arguments:
        destination: 64bit general registers
        literal: literal

- name: .data
  byte sequence: 0x00
  size: 0
  arguments:
    data: literal32
```

4.5 Normal Instructions

Normal instructions will make up 80-90% of the instructions which you define. They are selected for using their name and their type information is then validated (if there is a mismatch an error is thrown). There are 2 (or really 3, but that will be explained in 4.6).

Byte Sequence Literal is the simpler form an instruction definition. It is designed for the most trivial use case of instruction literal followed by arguments (using their default sizes and formats). For this form 'byte sequence' is defined by passing a literal value and then 'arguments' is a dictionary composed of name and type pairs where name is name used for help and error messages and type is used to type-check and generate the binary values of the arguments.

```
 - name: add
   byte sequence: 0x00
   size: 4
   arguments:
     destination: register
     lhs: register
     rhs: register
```

Byte Sequence List is deigned to build up instructions which require an arbitrary (but fixed size) format and argument layout. You can place literals, arguments and sections of arguments in any order for the instruction.

The byte sequence definition here is a list of dictionaries where each dictionary defines either a literal value or a reference to an argument.

Literal is defined simply by providing the value using the field 'literal' as literal and providing the size of the segment (as a number of bits) as an integer.

Path permits you to reference any argument and (if supported) any subsection of that argument. The path is recursively resolved and will return the value at that path, or raises an error if it cannot be resolved. Currently, only the 'memory' reference type supports nested resolution and it has the fields: 'source' and 'offset' which reference the first and second fields of a reference, for example "address.offset" to reference the offset field of the address argument. For the path segment, the size is optional.

```
- name: mov rl
  byte sequence:
  - path: destination
    size: 5
  - path: literal
    size: 16
  - literal: 0x694
    size: 11
  arguments:
    destination: 64bit general registers
    literal: literal
```

4.6 Instruction Format

Instruction formats are designed for the purpose of deduplicating the byte sequence list for the **Byte Sequence List** format of the 'normal' instructions. You define the name of the instruction format and then, in the same form as in the byte sequence list, you list out the segments of the byte sequence. The paths here are resolved in the context of their instruction which references them.

```
instruction formats:
  compare and branch:
    - literal: 0xA2
      size: 8
    - path: label
      size: 19
    - literal: 0x01
      size: 1
    - path: cond
      size: 4
```

One additional feature that this supports in instructions is the 'aliases' block, where instruction paths can be defined to point to an existing argument, for example if there is a naming mismatch between the format and the argument names, or if you wish to refer to a subsection of an argument, e.g. the offset.

It may be preferable to use aliases instead of renaming the arguments as it may give far more helpful names and fields in the automatically generated help.

```
instruction formats:
  compare and branch:
    - literal: 0xA2
      size: 8
    - path: label
      size: 19
    - literal: 0x01
      size: 1
    - path: cond
      size: 4

-- This is just an example.
instructions:
  - name: jmpf
    instruction format: compare and branch
    aliases:
        cond: condition
        label: destination
    arguments:
      destination: label
      condition: register
```

5 Errors and Explanations

There are a multitude of errors types in this system but in general and they are each designed for use in a specific context, some are specifically for use during the Configuration parsing phase, some are internal errors which should never occur during normal operation and lastly some are raised during the assembly file, parsing, assembling or output stage.

One important consideration to keep in mind when hunting down as error is that certain types of mistakes in the configuration stage can manifest them selves as errors during the assembly phase, for example if your regular expressions fail to capture the expression they may manifest as run-time errors in the assembler. So, be careful about that.

The various errors are enumerated below in a non exhaustive list (some error types are currently not in use and are excluded).

Lastly, if none of this is of any help, this is a JVM program and you always have the nuclear option of using a JVM debugger to step through the program directly. This program's source should be available and if not, it employs no obfuscation techniques, so feel free to de-compile it an inspect it that way. As a note, it is a Kotlin program so some of the decompiled code is non-idiomatic in java.

5.1 Configuration Parse Errors

Due to the design of configuration parser, line level granularity of errors is not available.

IncompleteDeclarationParserError indicates one of two things. Either that you are missing or have misnamed one of the three primary blocks ('instructions', 'references' or 'configuration') or that you have referenced an instruction format which doesn't match any of your declared formats. In both cases check the spelling of both formats.

MalformedDeclaration indicates that the format of your declarations doesn't match the form that the configuration parse expects. See 4 for details.

InvalidOption is almost a catch all error type for anything for which a more specific error does not exist. But its primary purpose is for declaring missing fields in the configuration blocks where it highlights exactly which field is missing and from which block.

5.2 Internal Errors

These errors should ideally never be raised during normal operations, but if they are, contact me and I would appreciate it if you could attach the stack trace and any documents required to reproduce the error.

InvalidImplementation is raised if a module (internal or an extension) has incorrectly implemented some API.

5.3 Assembler Errors

These are the errors you see during if an error is encountered when trying to lex, parse or assemble the actual assembly language file. These errors should be annotated with the line numbers, to allow you to easily track down an error and identify its cause.

InstructionParseError is raised if an instruction is correctly identified but its arguments cannot be parsed correctly into an expected form. It should display the acceptable forms for the instruction.

DataSourceParseError occurs when a reference type fails to be parsed correctly, for example if a (potentially recursive) type such as indexed references cannot match on one of the nested types, e.g. cannot match on the offset for a[0xx1].

LineParseError generally occurs when there is text on the line which is un-handled, for example if you have a comment regular expression which does not capture the whole comment block.

UndeclaredLabelError is raised if you reference a label for which the assembler cannot find the declaration. Generally check spellings here.

IncorrectTypeError occurs when an instruction is identified and its arguments are fully parsed, but they do not match the types expected for that instruction, fro example if an integer is passed instead of a register constant.

AbstractInstructionInstantiationError is the meta-instruction counterpart to 'InstructionParseError' and 'IncorrectTypeError'. It is raised if an instruction is identified as a meta-instruction but does not match the type of any of the candidate instructions.

PathResolutionError is raised if a path from an instruction declaration cannot be resolved. For example if you reference an argument which doesn't exist, or if there is a spelling error, or if for example if you reference a nested path and make an error there.

Ideally this should have been raised during the configuration parsing phase but currently it is raised during the assembly phase and this may change in the future.

6 Frequently Asked Questions and Common 'Gotchas'

I have an error, what should I do about it? RTFM. 5 specifically discusses the errors, general conventions surrounding them, normal causes and debugging tips.

What about cyclic references? Currently, the system has no system for cyclic references or forward declarations. The best option for dealing with those is through the usage of meta-references as container types or carefully avoiding circular references where possible.

#TODO write about default endianess

7 Appendices

7.1 Complete ”test.asm”

```
// Ussembly!
// The impossible has happened, now it falls
// upon you to make sure the world knows it...
// Prepare the world!

hell:
    freeze water
    put pig0, #wings[0]
    goto #hell

wings:
    .data 0xD, 0xE
    .data 0xA, 0xD
    .data 0xB, 0xE
    .data 0xE, 0xF
```

7.2 Full Example

```
configuration:
  line can start with label: true
  label bit size: 16
  word size: 16
  label regex: "^(\\w+)"
  argument separator: ",\\s*"
  comment regex: ";.*"

references:
  - name: literal4
    kind: meta
    references:
      - name: hexadecimal
        kind: literal
        literal type: HEXADECIMAL
        literal size: 4
        validation regex: "0x\\d+"
        extraction regex: "0x(\\d+)"

      - name: int4
        kind: literal
        literal type: INTEGER
        literal size: 4
        validation regex: "\\d+"
        extraction regex: "(\\d+)"

  - name: literal16
    kind: meta
    references:
      - name: hexadecimal
        kind: literal
        literal type: HEXADECIMAL
        literal size: 16
        validation regex: "0x\\d+"
        extraction regex: "0x(\\d+)"

      - name: int4
        kind: literal
        literal type: INTEGER
        literal size: 16
        validation regex: "(\\+|-)?\\d+"
        extraction regex: "((\\+|-)?\\d+)"

  - name: register
    kind: meta
    references:
      - name: general purpose register
```

References

[1] Clark C Evans. Yaml.

[2] Clark C Evans. Yaml specification.

[3] Oracle inc. Java javadoc - message format.