Generic Assembler
User Manual

Contents

Overview

An assembler is a program that reads in a source program written in an assembly language, and translates it to the corresponding machine language.

There are many different computer architectures, each with its own machine (and assembly) language.

The Generic Assembler reads in two inputs: (1) a specification of the computer architecture and assembly language, and (2) a source program written in that assembly language. The software then outputs the corresponding machine language result and an analysis of both input files (for error checking purposes).

Using Generic Assembler

Download

The Generic Assembler is a java application, consisting of several java files contained within a directory. These are available from <link>.

Program Execution

The Generic Assembler reads in two text files (.txt):

1. An architecture and assembly language specification file.
2. An assembly language source program.

To run the software follow these steps:

1. Navigate to the software's root directory:

```
cd Generic-Assembler/
```

2. Put both the specification and assembly program files here.

3. Execute Generic Assembler, from command line, specifying both filenames (specification file first, assembly file second):

```
java -cp bin/  Main <nameOfSpecFile>  <nameOfAssemblyFile>
```

For example:

```
java -cp bin/  Main spec.txt assembly.txt
```

Input Files

The program takes as input two text files (.txt); an architecture (and assembly language syntax) specification file, and an assembly source code file.

Each file is analysed on a **line-by-line** basis.

Comments may be added to either file. The start of a comment is indicated by a semi-colon (;) and everything thereafter (within the same line) is assumed to be a comment. For example:

```
ADD cl, al     ; This is a comment
```

Comments are ignored by the assembler.

Empty lines (or lines containing only whitespace) are also ignored by the assembler. (There is one exception to this rule, see MnemonicData).

Specification File

This file must be delivered to the software as a text file (.txt) and is the first input file specified (as previously described, see Execution). It describes:

- The name of the architecture.
- Register names with their respective encodings.
- Instruction formats (fields with bit lengths)
- An "assembly operand tree" which describes the arrangement and "types" of operands the assembler can expect from the assembly source code.
- Mnemonic's supported by the architecture, including relevant information such as opcode encodings and the expected operand formats that each mnemonic supports.
- The endian ("big" or "little")
- The minimum addressable unit (the minimum size of memory that can be addressed individually)

Sections

In order for the user to describe an architecture and assembly language syntax, the specification file is split up into 7 sections. All sections are required for the software to function properly, and **the format of each section must be respected**.

The order in which each section appears within the specification file is not important and is ultimately up to the user or the creator of the file.

All sections have a section "header" (all of which end with a colon, and none of which are case sensitive), followed by the appropriate information (in the correct format). This is described in detail in the categories below.

The general format of the specification file is:

```
architecture:
```

```
; architecture data here

registers:

; registers data here

instructionformat:

; instruction format data here

assemblyoptree:

; assemblyoptree data here

mnemonicdata:

; mnemonic data here

endian:

; endian data here

minaddressableunit:

; minimum addressable unit data here
```

Architecture

The purpose of this section within the specification file is simply to name the computer architecture being described.

Input Format

```
architecture:

<architectureName>
```

Example Input

```
architecture:

x86
```

- `architectureName` must only be one line. For example:

```
        architecture:

        Intel x86           ; valid



        architecture:

        Intel
        x86                 ; not valid
```

Registers

The purpose of this section within the specification file is to assign register names (string representations) to their respective values in either binary, hexadecimal or integer form.

It may contain an arbitrary number of register names and values.

Input Format

```
registers:

<registerName>   <value><B/H/I>
```

Example Input

```
registers:

eax    000B
ecx    001B
edx    010B
...
```

The first line assigns register name `eax` to binary encoding `000`.
The second line assigns register name `ecx` to binary encoding `001` etc.

- `registerName` and `value` must be separated by one or more spaces.
- `registerName` must be a single token only, containing any combination of letters, numbers or symbols.
- `<value>B` means `value` is a binary value:

```
     al   000B       ; binary value 000
```

`<value>D` means `value` is a hexadecimal value:

```
     di     7H       ; hex value 7 (111 in binary)
```

`<value>I` means `value` is an integer value:

```
     esi    6I       ; integer value 6 (110 in binary)
```

InstructionFormat

The purpose of this part of the specification file is to specify the architecture's instruction formats. Each instruction format (or component) must be given a name. Then its fields are specified along with each field's respective bit length.

It may contain an arbitrary number of instruction formats, and each instruction format may contain an arbitrary number of fields.

Input Format

```
instructionformat:

<instructionName> : <fieldName>(<bitLength>)
```

Example Input

```
instructionformat:

opcode : op(6) d(1) s(1)

mod-r/m : mod(2) reg(3) rm(3)

sib : ss(2) index(3) base(3)

displacement : dis(32)

...
```

The first instruction format specified assigns instruction component with name `opcode` to three fields (`op`, `d` and `s`). `op` field is 6 bits long, both `d` and `s` fields are 1 bit long.

- If an instruction defined has more than one field, then these fields must be separated by one or more spaces:

```
opcode : op(6) d(1)s(1)       ; not valid
opcode : op(6)d(1) s(1)       ; not valid
opcode : op(6) d(1) s(1) ; valid
```

- There must be no whitespace between the `fieldName`, `bitLength` and enclosing brackets:

```
opcode : op(6) d(1 ) s(1)     ; not valid
opcode : op (6) d(1) s (1)    ; not valid
opcode : op(6) d(1) s(1) ; valid
```

- `instructionName` must be a single token consisting of any combination of letters, numbers or symbols (except colon):

```
opcode.L : op(6) d(1) s(1)    ; valid
opcode L : op(6) d(1) s(1)    ; not valid
```

- `instructionName` and its fields must be separated by a colon:

```
opcode  op(6) d(1) s(1)       ; not valid
opcode : op(6) d(1) s(1) ; valid
```

- `fieldName` must be a single token. Consisting only of letters and numbers:

```
opcode : op(6) d(1) s(1) ; valid
opcode : op.L(6) d(1) s(1)    ; not valid
```

- `bitLength` must be an integer only.

AssemblyOpTree

The *assemblyOpTree* describes the arrangement and types of operands within the assembly language. This section can contain an arbitrary number of lines and must be consistent with the structure of the assembly language source code (within the provided assembly source file). The software analyses each operand within the source assembly instruction with the tree, and establishes what type of operand it is. This is best described with an example:

Available assemblyOpTree notation

`<label>*`	zero or more <label>
`<label>+`	one or more <label>
`<label>?`	optional <label> (zero or one)
`INT`	an integer
`HEX`	a hex variable
`LABEL`	an assembly program “label”

Format

```
assemblyOpTree:

<label> : <label/syntax/literal>
```

Example

```
statement : mnemonic operand operand operand

operand : immediate
operand : register
operand : memory

immediate : INT
```

```
register : $reg
reg : "t1"
reg : "t2"

memory : immediate(register)

mnemonic : "ADD"
```

The above example is very crude, but it introduces a basic *assemblyOpTree*. This describes a very primitive subset of MIPS assembly language. The *assemblyOpTree* above describes an assembly language with the following properties:

- An assembly `statement` can take the form `mnemonic operand operand operand`.
- An `operand` can be an `immediate`, `register` or `memory` variable.
- An `immediate` variable is an integer (`INT`).
- The syntax of a `register` is of the form `$reg`.
- A `reg` can be `t1` or `t2`.
- A `memory` variable can be of the form `immediate(register)`.
- A `mnemonic` can be `ADD`.

The software will parse each source assembly line against the *assemblyOpTree* and establish each operand type, and if the assembly line is consistent. For example:

```
ADD $t2, $t1, $t1

; consistent, mnem register register register

ADD $t2, $t1, $t1, $t1

; not consistent (four operands)

ADD $t2, $t1

; not consistent (only two operands)

ADD $t2, $t1, $t3
```

```
; not consistent (t3 not defined in assemblyOpTree)

ADD $t1, $t1, 3($t2)

; is consistent, mnem register register memory

ADD $t1, $t1, 5

; is consistent, mnem register register immediate

MOV $t1, $t1, 5

; not consistent (MOV not defined in assemblyOpTree)

ADD $t2, t1, 5

; not consistent (t1 does not possess $ symbol)

ADD $t1, $t1, 4[$t0]

; not consistent (has squared brackets instead of curved)
```

Many different types of assembly instructions may have an arbitrary number of operands, it will therefore be time consuming to list every single conceivable combination of the number of operands, i.e.:

```
statement : mnemonic operand
statement : mnemonic operand operand
statement : mnemonic operand operand operand
...
```

Therefore a wildcard may be used to describe this instead. If the `statement` expression in the MIPS example above was replaced with:

```
statement : mnemonic operand*
```

Then this would allow any number of operands (or none at all) to be expressed after the `mnemonic`. Similarly if the expression was:

```
statement : mnemonic operand+
```

Then there must be at least one operand present after the `mnemonic`.

- The "root" term (`statement` in the MIPS example) **must be the first line specified** (i.e., the first *assemblyOpTree* line in the *assemblyOpTree* section).
- A assemblyOpTree label (to the left of the colon) must be a single token. It may contain any combination of letters and/or numbers. For example:

  ```
  register : $reg      ; valid

  $register : $reg     ; not valid (symbol)

  regi ster : $reg     ; not valid (not single token)
  ```

- If there are several labels (to the right of the colon) then they must be separated by one or more spaces:

  ```
  statement : mnemonic operand operand operand
  ```

- It is assumed that operands within a source assembly instruction are separated by one or more spaces or commas. **These are omitted before analysing with the *assemblyOpTree***, therefore literals defined within the *assemblyOpTree* should never **start** or **end** with one or more commas. For example:

  ```
  ADD $t2, $t1, $t3
  ADD $t2, ,,$t1,  , $t3
  ```

  The strings highlighted in red are the ones evaluated by the *assemblyOpTree*. So literals which begin or end with a comma (i.e., `register : ,$reg,` or `reg : ",t1"`) will never result in a match as its not possible for an operand (which is being evaluated by the *assemblyOpTree*) to begin or end with a comma.

- **Wildcards** (`*`, `+`, `?`) **can only be applied to *assemblyOpTree* labels which are not "leaf" expressions** (i.e., labels which have a further definition within the *assemblyOpTree*). For example (from the MIPS tree shown previously):

  ```
  statement : mnemonic operand*

  ; valid as operand further defined
  ; (operand : register etc)

  register : $reg*

  ; not valid as $reg not further defined within
  ; assemblyOpTree, only reg is (reg : "t1" etc)

  operand : register*

  ; valid as register further defined
  ; (register : $reg)
  ```

- **There should be no whitespace in what is intended to be a single operand**, either in its specification in the *assemblyOpTree* or in its use in the assembly program. For instance (in the MIPS example) a register operand is of the form:

  ```
  register : $reg
  ```

  However, if it was instead specified as:

  ```
  memory : $ reg
  ```

  Then the software will treat this as two separate operand "entities" and will evaluate the assembly line expecting two separate operands (a `$` followed by a space then `reg`.)

  NOTE: This could still work but it makes things more complicated (especially when completing other sections), so it is advisable to adhere to this constraint.

- Nested single operands must have operands separated by a symbol (or symbols) and this should be consistent with the assembly source code (no spaces remember!):

  ```
  memory : immediate(register)
  ```

```
; valid

memory : immediateregister

; not valid, no symbol separation

memory : immediate#register#

; valid

memory : immediate#register

; valid

memory : immediate(register+register)

; valid

memory : (immediate((register)+register))

; valid

memory : immediate(register + register)

; not valid, spaces exist
```

- Symbols may be added to labels to specify syntax:

```
register : $reg
reg : "t1"
reg : "t2"
```

This means that a *`register`* in the assembly program can be specified as *`$t1`* or *`$t2`*.

NOTE: Could use instead:

```
register : "$t1"
register : "$t2"
```

This means the same thing effectively.

- An `INT` term must be an integer.
  A `HEX` term must be a hexadecimal value.

  For example, given the *assemblyOpTree*:

  ```
  statement : mnemonic operand*

  operand : immediate
  operand : register
  operand : memory

  register : $reg
  reg : "t1"

  mnemonic : "ADD"

  immediate : INT

  ...
  ```

  The assembly code:

  ```
  ADD $t1, 3          ; is valid (mnem register
                      ; immediate).
  ```

  If the `immediate` *assemblyOpTree* line was changed to `immediate : #INT` then:

  ```
  ADD $t1, #3         ; is valid (mnem register
                      ; immediate).
  ```

- It is assumed by the assembler that any labels **marking a relocation point will be the first operand specified in an instruction** (i.e., **`loop`** `ADD ecx, eax`). This operand **must** be declared within the *assemblyOpTree* with the keyword **`LABEL`**. For example:

```
statement : label? mnem op*
...
op : LABEL
..
label : LABEL
```

This describes how an assembly statement may contain an optional label. `statement` would therefore be valid for instructions with or without a label at the beginning.

A label within an assembly instruction must be a single token consisting of letters only. For example:

```
loop ADD bh, al      ; loop is a valid label

loop1 ADD bh, al     ; loop1 is not valid
```

If an instruction (such as a jump instruction for example) uses **a label as an operand** (`op` in the example *assemblyOpTree* given above), then the user must remember to specify that an operand may also be a `LABEL`.

- Literals (`mnemonic : "ADD"`*)* can be in inverted commas for readability, although this is not necessary.

To conclude this section, here is an example of a very small (and incomplete) subset of the x86 assembly *assemblyOpTree*, which encapsulates the guidelines expressed :

```
statement : label? mnem op*

op : adrMode

adrMode : reg
adrMode : imm
adrMode : dirMem
adrMode : dirReg
adrMode : baseInd
adrMode : baseIndDisp
adrMode : scale
adrMode : baseIndScale
adrMode : label

reg : reg8
reg : reg16
reg : reg32

reg8 : "al"
reg8 : "cl"
reg8 : "dl"
reg8 : "bl"
reg8 : "ah"
reg8 : "ch"
reg8 : "dh"
reg8 : "bh"

reg16 : "ax"
reg16 : "cx"
reg16 : "dx"
reg16 : "bx"
reg16 : "sp"
reg16 : "bp"
reg16 : "si"
reg16 : "di"

reg32 : "eax"
reg32 : "ecx"
reg32 : "edx"
reg32 : "ebx"
reg32 : "esp"
reg32 : "ebp"
reg32 : "esi"
reg32 : "edi"

imm : INT

dirMem : [HEX]

dirReg : [reg]
```

```
baseInd : [reg+reg]

baseIndDisp : [reg+imm]

scale : scaleOne
scale : scaleTwo

scaleOne : [imm+reg*1]
scaleTwo : [imm+reg*2]

baseIndScale : baseIndScaleFour

baseIndScaleFour : [reg+reg*4]

mnem : "ADD"
mnem : "MOV"

label : LABEL
```

<u>MnemonicData</u>

This section exists to specify all the relevant data needed to populate an instruction via its respective mnemonic. For a given mnemonic it specifies:

- any "global" opcodes for that mnemonic (i.e., opcodes which are the same for that mnemonic regardless of what format the operands for that mnemonic may take).

- the different operand formats the operands may take for that mnemonic.

- instruction fields (from *instructionFormat*) for a particular operand format which are needed for encoding the instruction for that mnemonic.

- "local" opcodes which are unique to a particular operand format of that mnemonic.

- the instruction component/s (from *instructionFormat*) which compose the entire instruction for a particular operand format of that mnemonic.

Having an **apt *assemblyOpTree*** is crucial so that this section may function as expected.

Line Input Format

```
mnemonicdata:

<mnemonicName>
     <globalOpcodes>

     <operandFormat>
          <instructionFieldLabels>
          <localOpcodes>
          <instructionFormat>

     <operandFormat>
          ...
```

```
<mnemonicName>
    ...
```

Example Input

```
mnemonicdata:

ADD
    op=000000                       ; one tab
                                    ; empty line
    mnem reg8, reg8                 ; one tab
        mnem reg rm                 ; two tabs
        d=1, s=0, mod=11            ; two tabs
        opcode mod-r/m              ; two tabs
                                    ; empty line
    mnem reg32, reg32
        mnem rm reg
        d=0, s=1, mod=11
        opcode mod-r/m
                                    ; empty line
ADC
    op=010011

    mnem reg16, reg16
        ...
```

The example above (taken from x86 ADD instruction incidentally), declares mnemonic `ADD` and `ADC`. `ADD` mnemonic has the following properties:

- Regardless of the format of the mnemonic the opcode (`op` field in the instruction) is always `000000` (a "global" opcode with respect to the `ADD` mnemonic):

  ```
  ADD
      op=000000

  mnem reg8, reg8
  ```

```
        mnem reg rm
        d=1, s=0, mod=11
        opcode mod-r/m

mnem reg32, reg32
        mnem rm reg
        d=0, s=1, mod=11
        opcode mod-r/m
```

This means that regardless of whether the operand format for mnemonic `ADD` is `mnem reg8, reg8` or `mnem reg32, reg32`, `op` field is always encoded as `000000`.

- The two formats the `ADD` instruction may take are `mnem reg8, reg8` and `mnem reg32, reg32` (these terms **must** be defined within the *assemblyOpTree*):

```
ADD
        op=000000

mnem reg8, reg8
        mnem reg rm
        d=1, s=0, mod=11
        opcode mod-r/m

mnem reg32, reg32
        mnem rm reg
        d=0, s=1, mod=11
        opcode mod-r/m
```

These may be "viewed" as two separate "blocks". The 3 double tabbed lines under each format is data unique to the format above it, and which is needed when encoding the respective operand format. For example:

```
ADD
        op=000000

mnem reg8, reg8                  ; unique data for op format
        mnem reg rm              ; mnem reg8, reg8
        d=1, s=0, mod=11
        opcode mod-r/m

mnem reg32, reg32
        mnem rm reg
        d=0, s=1, mod=11
        opcode mod-r/m
```

- For operand format `mnem reg8, reg8`, the instruction field labels (needed to encode this instruction) are `mnem reg rm`:

```
ADD
        op=000000

mnem reg8, reg8
        mnem reg rm
        d=1, s=0, mod=11
        opcode mod-r/m

mnem reg32, reg32
        mnem rm reg
        d=0, s=1, mod=11
        opcode mod-r/m
```

This means that the first `reg8` will be used to populate the `reg` field within the instruction. The second `reg8` will be used to populate the `rm` field within the instruction.

NOTE: `mnem` is used only as a placeholder as `mnem` is not defined as an instruction field (in the *instructionFormat* section).

- Operand format `mnem reg8, reg8`, has "local" opcodes `d=1, s=0, mod=11` (i.e., opcodes which are unique to this operand format of the `ADD` mnemonic and which are needed for the instruction encoding).

```
ADD
        op=000000

mnem reg8, reg8
        mnem reg rm
```

```
          d=1, s=0, mod=11
          opcode mod-r/m

mnem reg32, reg32
          mnem rm reg
          d=0, s=1, mod=11
          opcode mod-r/m
```

- The instruction components (defined in section *instructionFormat*) used to encode `ADD` operand format `mnem reg8, reg8` are `opcode mod-r/m` (i.e., `opcode` and `mod-r/m`).

```
ADD
          op=000000

mnem reg8, reg8
          mnem reg rm
          d=1, s=0, mod=11
          opcode mod-r/m

mnem reg32, reg32
          mnem rm reg
          d=0, s=1, mod=11
          opcode mod-r/m
```

This means that the entire instruction needed to encode this operand format is:

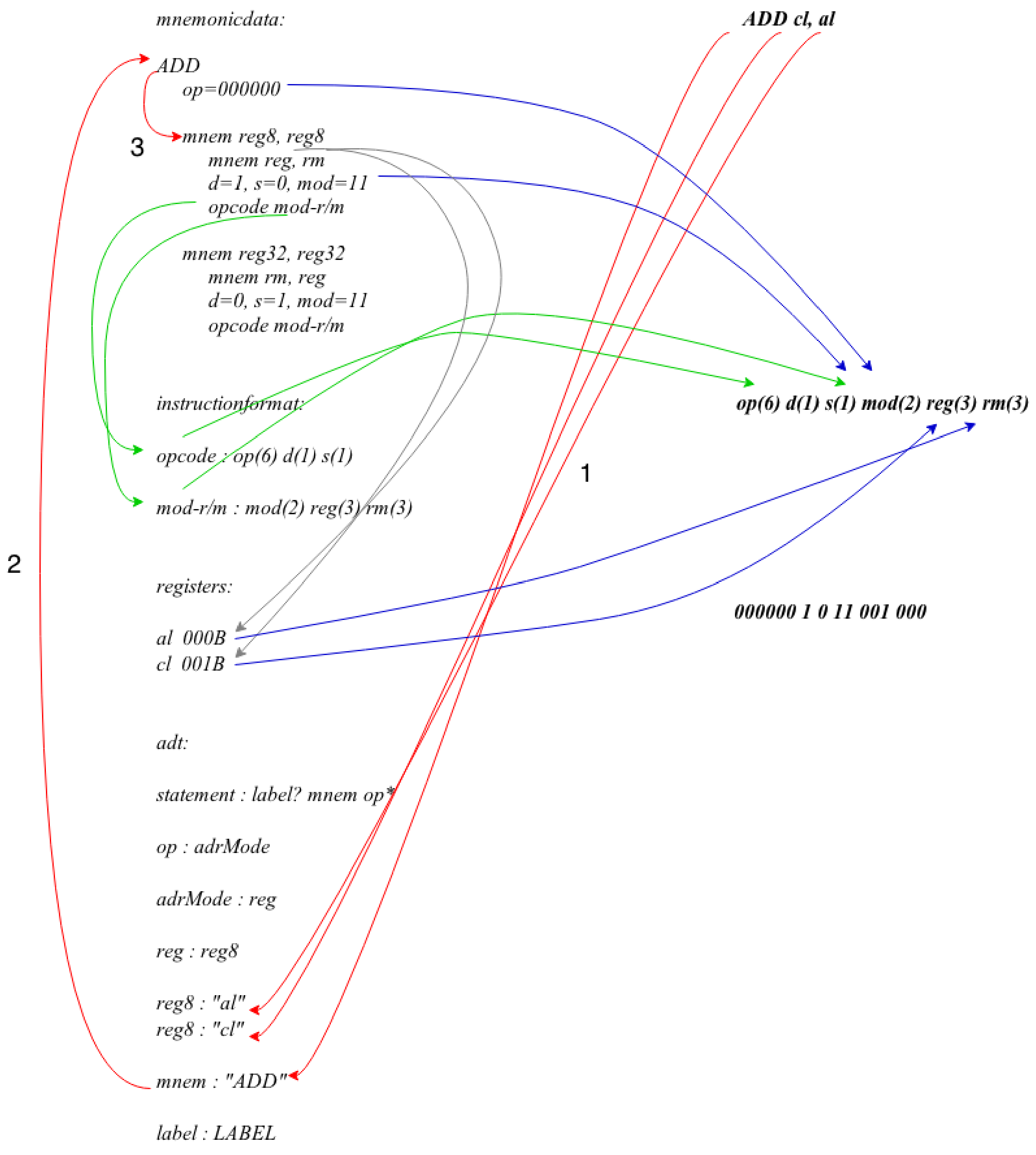
```
instructionformat:

opcode : op(6) d(1) s(1)
mod-r/m : mod(2) reg(3) rm(3)
```

Or:

```
op(6) d(1) s(1) mod(2) reg(3) rm(3)
```

To convey this visually:



As you can see from the diagram above, assembly instruction `ADD cl, al` will be evaluated to be of operand format `mnem reg8 reg8` via the *assemblyOpTree* (1). This will then be matched to mnemonic `ADD` (2) and then of `ADD` format `mnem reg8, reg8` (3).

The instruction encoding format is specified to be `opcode mod-r/m` (i.e., `op(6) d(1) s(1) mod(2) reg(3) rm(3)`) this is shown via the green lines.

These fields are then "populated" from either the global opcodes, local opcodes and opcodes which are matched directly from the assembly instruction, i.e., `reg` field is first `reg8` (register `cl`) and `rm` field is second `reg8` (register `al`). The values of registers `cl` and `al` will be taken from the registers section as shown by the blue lines.

- `mnemonicName` line must have no whitespace at the beginning. It must be a single token which may consist of any combination of letters, numbers or symbols. For example:

  ```
  ADD         ; legit
  ADD.L       ; legit
  ADD L       ; not legit (2 tokens)
  ```

- `globalOpcodes` line must begin with a tab, and is of the format `<fieldName>=<value>`. Multiple global opcodes must be separated by a comma (`,`). If there are no global opcodes then this line may be simply omitted. Examples:

  ```
       op=001001, d=0, s=0           ; legit

       op=000000                     ; legit
  ```

  If no global opcodes exist for mnemonic, then simply omit line:

  ```
  ADD
                            ; no global opcodes present
       mnem reg8, reg8
            mnem reg rm
            op=000000, d=1, s=0, mod=11
            opcode mod-r/m

       mnem reg32, reg32
            mnem rm reg
            op=000000, d=0, s=1, mod=11
            opcode mod-r/m
  ```

- Separate `operandFormats` must be separated by one or more empty lines:

```
ADD
     op=000000
                         ; empty line
     mnem reg8, reg8
          mnem reg rm
          op=000000, d=1, s=0, mod=11
          opcode mod-r/m
                         ; empty line
     mnem reg32, reg32
          mnem rm reg
          op=000000, d=0, s=1, mod=11
          opcode mod-r/m
```

- Separate mnemonic declarations must also be separated by one or more empty lines:

```
ADD
     op=000000

     mnem reg8, reg8
          mnem reg rm
          d=1, s=0, mod=11
          opcode mod-r/m

     mnem reg32, reg32
          mnem rm reg
          d=0, s=1, mod=11
          opcode mod-r/m
                    ; empty line
ADC
     op=010011

     mnem reg16, reg16
          ...
                    ; empty line
AND
```

```
        ...
```

- `operandFormat` line must start with a tab. Not only does it specify the format of the operands via the *assemblyOpTree* terms, but must be **consistent with the syntax of the line with the inclusion of expected comma's**. Remember that before *assemblyOpTree* analysis separator comma's are "ignored".

  For example, instruction `ADD cl, al` will be evaluated by the *assemblyOpTree* to be of format `mnem reg8 reg8`. Use of commas and spaces within the `operandFormat` line must be consistent with the syntax of the source assembly line. For example:

  ```
      ADD cl, al                  ; assembly source line

      mnem reg8, reg8             ; is valid
          mnem reg rm
          d=1, s=0, mod=11
          opcode mod-r/m

      mnem reg8,, reg8            ; not valid
          mnem reg rm
          d=1, s=0, mod=11
          opcode mod-r/m
  ```

- `instructionFieldLabels` must start with two tabs. They "map" operands within the assembly source line which are directly needed for the instruction encoding. For example:

  ```
  ADD
      op=000000

  mnem reg8, reg8
      mnem reg rm
      d=1, s=0, mod=11
      opcode mod-r/m

  ...

  instructionformat:
  ```

```
opcode : op(6) d(1) s(1)
mod-r/m : mod(2) reg(3) rm(3)

...

registers:

al   000B
cl   001B
```

Given assembly line:

```
     ADD cl, al
```

and

```
     mnem reg8, reg8
          mnem reg rm
```

Then register `cl` is mapped to encode the `reg` field within the instruction, and register *`al`* is mapped to encode the *`rm`* field within the instruction.

NOTE: Comma's (used as separators) must not be specified here (as they are in the operandFormat line above). Each token in `instructionFieldLabels is` "mapped" to the corresponding operand in the assembly source line. **Each token within `instructionFieldLabels` must consist of letters and numbers only.**

To give a more complex example, given the specification:

```
mnemonicdata:

ADD
     op=000000

     mnem reg32, baseIndScale
          mnem reg [base+index*4]
          d=1, s=1, ss=10, mod=00, rm=100
          opcode mod-r/m sib


registers:

ecx    001B
ebx    011B
edi    111B


instructionformat:

opcode : op(6) d(1) s(1)
mod-r/m : mod(2) reg(3) rm(3)
sib : ss(2) index(3) base(3)


assemblyOpTree:

statement : label? mnem op*

op : adrMode

adrMode : reg
adrMode : baseIndScale

reg : reg32

reg32 : "ecx"
reg32 : "ebx"
reg32 : "edi"

baseIndScale : [reg32+reg32*4]

mnem : "ADD"

...
```

So given assembly line:

```
ADD ecx, [ebx+edi*4]
```

and `instructionFieldLabels` (with `operandFormat`):

```
mnem reg32, baseIndScale
     mnem reg [base+index*4]
```

Then register `ecx` will be used to populate the `reg` field within the instruction.
Register `ebx` will be used to populate the `base` field within the instruction.
And register `edi` will be used to populate the `index` field within the instruction.

NOTE: The syntax of the **operands** should remain consistent (i.e., use of symbols). And tokens which are not used for direct encoding (such as `mnem` and `4` in the example above should have a placeholder in place; it isn't important what this placeholder is, along as it is a single token).

- `localOpcodes` must start with two tabs and is of the same format as `globalOpcodes`, `<fieldName>=<value>`. Multiple local opcodes must be separated by a comma (`,`). **If there are no local opcodes** then simply put "`--`" after the two tabs. For example:

```
ADD
     opcode=000000, func=100000, shamt=00000

      mnem reg, reg, reg
          mnem rd rs rt
          --                    ; no local opcodes
          R-type
```

- `instructionFormat` must start with two tabs. This line lists the instruction names (defined in instructionFormat section) which compose the entire instruction. This can be one or more instruction names, separated by one or more spaces.

Endian

This section exists to specify the endianness.

Input Format

```
endian:

<big/little>
```

Example Input

```
endian:

big
```

- This must consist of only one line, limited to the strings “big” or “little” (not case sensitive).

MinAddressableUnit

This section exists to specify the minimum addressable unit of the architecture in **bits**.

Input Format

```
minaddressableunit:

<int>
```

Example Input

```
minaddressableunit:

8
```

- This must consist of only one line, limited to a single string consisting only of an int variable.

Assembly File

This file must be delivered to the software as a text file (.txt) and is the second input file specified (as previously described, see Execution). This file contains the source assembly code.

As with most assembly source code files. it contains a .data and .text section.

The general format of the specification file is:

```
section.data

<data>

section .text
     global main    ; tab
main:

<assemblyCode>
```

- The data section //**TODO**
- The text section (.text) must begin with:

  ```
  section .text
       global main    ; tab
  main:

  ; the assembly code must go here

  ADD $t1, $t1, $t1
  AND $t1, $t1, $t3
  …
  ```

Output Files

The software will output two files; “error.txt” and “object.txt”.

If errors are made in the specification file then these will be shown within error.txt.

If there are no visible errors within the specification file, then the assembly will attempt to assemble each instruction within the source assembly file. If errors occur during the assembly process then these will be reported in object.txt.

If no errors are found either in the specification file or in the assembly process then the object code will be presented in object.txt.