# Generic Assembler
# User Manual

# Contents

# Overview

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

There are many different computer architectures, each with its own machine (and assembly) language. In order to successfully assemble an assembly source program for a particular computer architecture, several aspects of the architecture and its instruction set must be described. This includes the architecture's:

- register names with their respective encoding values.
- compatible instruction formats.
- assembly language structure.
- compatible mnemonics with their field encodings.
- compatible operand formats/arrangements for each respective mnemonic.
- endianness.
- minimum addressable unit.

The Generic Assembler reads in two inputs: (1) a specification of the computer architecture and its instruction set, and (2) a source program written in that assembly language. If successful the assembler will generate the object file. If errors exist within the specification file, this will be reported in output file spec_error_report.txt.

# Using Generic Assembler

## Program Execution

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

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

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 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).**

# Architecture 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 Program Execution). It describes:

- The name of the architecture.
- Register names with their respective encoding representations.
- Instruction formats (fields with bit lengths).
- An “assembly operand tree” which describes the arrangement and structure of operands the assembler can expect from the assembly source code.
- Mnemonics supported by the architecture, including relevant information such as field encodings and the expected operand arrangement for each mnemonic.
- 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 structure, 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 this manual.

**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:

MIPS
```

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

```
Architecture:

MIPS RISC           ; valid



Architecture:

MIPS
RISC                ; not valid (two lines)
```

# Registers

The purpose of this section within the specification file is to declare the architecture's processor register names (token representations) and assign them to their respective binary representations 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:

$t0  8H
$t1  9H
$t2  10H
...
```

The first line assigns register name `$t0` to hex value 8 (1000 in binary).
The second line assigns register name `$t1` to hex value 9 (1001 in binary).

- `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 all the architecture's instruction formats which are needed for instruction encoding.

**Each instruction format (or component) must be given a name. Each field within the instruction is also given a name and a respective bit length. The names given to both the instruction and its fields are ultimately up to the user, but must remain consistent with other sections within the specification file so that assembly source instructions may be assembled properly (this is described in detail in MnemonicData).**

This section may declare 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:

R-type : opcode(6) rs(5) rt(5) rd(5) shamt(5) func(6)

I-type : opcode(6) rs(5) rt(5) immediate(16)

J-type : opcode(6) address(26)

syscall : call(32)

...
```

The first instruction format specified, assigns instruction format/component with name `R-type` to six fields (`opcode`, `rs`, `rt`, `rd`, `shamt` and `func`). `opcode` field is 6 bits long; `rs`, `rt`, `rd` and `shamt` are all 5 bits long; `func` is 6 bits 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/or 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 assembly operand tree describes the arrangement and structure 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).

**Before assembly, the software analyses each operand within the source assembly instruction using the tree, and establishes what type of operand it is.** This is best described with an example:

***Available AssemblyOpTree notation (Explained later)***

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

***Input Format***

```
AssemblyOpTree:

<node> : <expression>
```

***Example Input***

```
AssemblyOpTree:

statement : mnemonic operand operand operand
```

```
operand : immediate
operand : register
operand : memory

immediate : INT

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

memory : immediate(register)

mnemonic : "ADD"
```

The above example 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` is of the form:

  `mnemonic operand operand operand`.

- An `operand` may be an `immediate`, `register` or `memory` operand.

- An `immediate` operand is an integer (`INT`).

- A `register` operand may be `$t1` or `$t2`.

- A `memory` operand is of the form `immediate(register)`.

- A `mnemonic` may be `ADD`.

**The software will parse each source assembly instruction against the AssemblyOpTree. When parsing, it takes one operand at a time from the source assembly instruction and analyses that operand with the tree.**

**It is assumed by the assembler that operands are separated by one or more spaces or commas within the source assembly instruction.** The operands highlighted in red for each instruction below are the tokens analysed (one at a time) against the tree:

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

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

ADD $t1, $t1, 5
```

**The software takes an operand and works its way logically down the tree via the root expression (`statement` in the example above) until it reaches a "leaf" expression (i.e., an expression token which is NOT an exact match with another node defined within the tree).** For example:

```
statement : mnemonic operand operand operand

operand : immediate
operand : register
operand : memory

immediate : INT                         ; leaf expression

register : "$t1"                        ; leaf expression
register : "$t2"                        ; leaf expression

memory : immediate(register)            ; leaf expression

mnemonic : "ADD"                        ; leaf expression
```

**The leaf expression reached is then used for comparison with the source assembly operand.**

**If successful, the software will establish each operand type, and if the assembly line is consistent.** For example:

```
ADD $t2, $t1, $t1
; consistent with tree, mnemonic register register register
```

```
ADD $t1, $t1, 3($t2)
; is consistent, mnemonic register register memory

ADD $t1, $t1, 5
; is consistent, mnemonic register register immediate

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)

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 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 within a root node expression, 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 tree shown 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`.

Wildcards can ONLY be applied to node references within the root node expression. The root node refers to the first node declared within the AssemblyOpTree section (`statement` in the example above).

- The "root" node (`statement` in the MIPS example) **must be the first node specified within the AssemblyOpTree section**. **Its expression should describe the general structure of the source assembly instructions**. For example:

```
AssemblyOpTree:

statement : mnemonic operand*
operand : immediate
...

; valid, root token statement is the first line within
; AssemblyOpTree section and describes how assembly
; instructions consist of a compulsory mnemonic followed
; by zero or more operands.
```

If this was instead:

```
AssemblyOpTree:

operand : immediate
statement : mnemonic operand*
...

; not valid, assembly instructions consisting only of a
; single immediate operand will pass as valid
```

**The order in which nodes are declared within the AssemblyOpTree (other than the root node) is not important.**

NOTE: The root node does not have to be `statement`. This is up to the user.

- **An AssemblyOpTree node must be a single token. It may contain any combination of letters and/or numbers.** For example:

```
register : "$t1"    ; valid
```

```
$register : "$t1"   ; not valid (symbol)

regi ster : "$t1"   ; not valid (not single token)
```

- If the root node expression contains several nodes 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 whitespace. Commas at the beginning or end of an operand are omitted before analysing with the AssemblyOpTree**. Therefore leaf expressions (and literals) defined within the AssemblyOpTree should never **begin** 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 leaf expressions (and literals) which begin or end with a comma (i.e., `register : ",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 root node expression tokens (which refer to nodes further defined in the tree)**. For example (from the MIPS tree shown previously):

```
statement : mnemonic operand*

; valid as wildcard has been applied to node in root node
; expression
```

```
statement : mnemonic operand operand operand

operand : immediate*

; not valid as wildcard has not been applied to a node
; within the root node expression
```

- **There should be no whitespace in an expression other than the root node expression**. As the root node expression should describe the general pattern of operands within a source assembly instruction, it is assumed by the assembler that all child nodes refer to a single operand. Therefore any whitespace within a child node expression is not valid, for example:

```
statement : mnemonic operand operand operand

operand : memory

memory : immediate (register) ; not valid

memory : immediate(register)       ; valid
```

- Leaf expressions which have other nodes embedded within it, must have these embedded nodes 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 node references within an expression to specify syntax. For example instead of:

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

  Could use instead:

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

  Both mean that a `register` in the assembly program can be specified as `$t1` or `$t2`.

- 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

immediate : INT

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

memory : immediate(register)

mnemonic : "ADD"
```

The assembly code:

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

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

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

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

```
statement : label? mnemonic operand*
...
operand : LABEL
...
label : LABEL
```

This describes how an assembly statement may begin with 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**, 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.

# MnemonicData

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” encodings for that mnemonic (i.e., static field encodings which are the same for that mnemonic regardless of what format/arrangement the operands for that mnemonic may take, such as the opcode).

- one or more operand formats/arrangements the operands may take for that mnemonic. For each operand arrangement declared, the user must also specify:

- what field (if any) a respective operand populates (e.g., the field the encoding value for a respective register populates, or a respective immediate value etc).

- any “local” encodings for the given operand arrangement (i.e., static field encodings which are unique to this arrangement of operands for that mnemonic).
-
- the instruction component/s (defined in section InstructionFormat) which compose the full instruction needed for encoding.

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

***Line Input Format***

```
MnemonicData:

<mnemonicName>
     <globalFieldEncodings>

     <operandFormat>
          <operandFieldEncodings>
          <localFieldEncodings>
          <instructionFormat>
```

```
    <operandFormat>
        ...

<mnemonicName>
    ...
```

***Example Input***

```
MnemonicData:

ADD
    opcode=000000B, func=100000B                    ; tab
                                                    ; empty line
    mnemonic register, register, register           ; tab
        mnem rd rs rt                               ; 2 tabs
        shamt=00000B                                ; 2 tabs
        R-type                                      ; 2 tabs
```

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

- Regardless of the format of the mnemonic the opcode (`opcode` field) is always encoded with binary `000000` and field `func` is always encoded with binary `100000` ("global" field encodings with respect to the `ADD` mnemonic):

```
ADD
    opcode=000000B, func=100000B

    mnemonic register, register, register
        mnem rd rs rt
        shamt=00000B
        R-type
```

- The operand format the `ADD` instruction may take is `mnemonic register, register, register` **(these tokens, without the commas, must be defined within the AssemblyOpTree)**:

```
ADD
     opcode=000000B, func=100000B

     mnemonic register, register, register
          mnem rd rs rt
          shamt=00000B
          R-type
```

- For operand format `mnemonic register, register, register`, the operand field encodings (needed to encode this instruction) are `mnem rd rs rt`:

```
ADD
     opcode=000000B, func=100000B

     mnemonic register, register, register
          mnem rd rs rt
          shamt=00000B
          R-type
```

So given the assembly line:

```
     ADD  $t2, $t1, $t1
   ; mnem rd   rs   rt
```

This means that for the `R-type` instruction, the `rd` field will be populated with the encoding value for `$t2`. The `rs` field will be populated with the encoding value for the first `$t1` register and the `rt` field will be populated with the encoding value for the second `$t1` register.

NOTE: `mnem` is used only as a placeholder as `mnem` is not defined as a field within instruction `R-type` (in the InstructionFormat section).

- `ADD` operand format `mnemonic register, register, register`, has "local" encoding `shamt=00000B` (i.e., an encoding which is unique to this operand format of the `ADD` mnemonic):

```
ADD
     opcode=000000B, func=100000B

     mnemonic register, register, register
          mnem rd rs rt
          shamt=00000B
          R-type
```

- The instruction format (defined in section InstructionFormat), used to encode `ADD` operand format `mnemonic register, register, register` is `R-type`.

```
ADD
     opcode=000000B, func=100000B

     mnemonic register, register, register
          mnem rd rs rt
          shamt=00000B
          R-type
```

`R-type` **must be defined within InstructionFormat section**:

```
InstructionFormat:

R-type: opcode(6) rs(5) rt(5) rd(5) shamt(5) func(6)

I-type: opcode(6) rs(5) rt(5) immediate(16)

J-type: opcode(6) address(26)
```

**Notice that all fields needed to encode instruction R-type have been defined within the global field encoding, local field encodings or operand field encodings**:

```
ADD
     opcode=000000B, func=100000B

     mnemonic register, register, register
          mnem rd rs rt
          shamt=00000B
          R-type
```

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

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

- `globalFieldEncodings` line must begin with a tab, and is of the format `<fieldName>=<value><B/H/I>`.

  Similarly to the way registers are assigned encoding values (in the Registers section), `B` indicates the value is binary; `H` indicates the value is hexadecimal; `I` indicates the value is an integer. Multiple global opcodes must be separated by a comma (`,`). For example:

```
opcode=000000B, func=100000B       ; valid
opcode=000000B                     ; valid
opcode=1FH                         ; valid
```

  If no global field encodings exist for the mnemonic, then simply omit line:

```
ADD
                          ; no global field encodings
     mnemonic register, register, register
          mnem rd rs rt
          opcode=000000B, func=100000B, shamt=00000B
          R-type
```

- A mnemonic may have several compatible operand formats. These must be separated by one or more empty lines, here is an example taken from x86 `ADD` instruction:

```
ADD
     op=000000B
                                ; empty line
```

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

This means that the ADD mnemonic has two valid operand formats:

```
mnem reg8, reg8
mnem reg32, reg32
```

**These tokens, without the separator commas, must be defined within the AssemblyOpTree.**

**The 3 lines which succeed each format is data unique to that format.** Remember that `op=000000B` is a global encoding for ADD and so applies to all operand variations of ADD.

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

```
ADD
     op=000000B

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

     mnem reg32, reg32
          mnem rm reg
          d=0B, s=1B, mod=11B
```

```
          opcode mod-r/m
                              ; empty line
ADC
     op=010011B

     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 commas are "ignored".

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

```
ADC
     op=000000B

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

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

- `operandFieldEncodings` must start with two tabs. They "map" operands within the assembly source line to instruction fields in which they are to populate.

  **Commas (used as separators) must not be specified here (this is already done within the operandFormat line above it).** Each token within `operandFieldEncodings` is "mapped" to the corresponding operand in the assembly source line.

  To give a more complex example, given the input:

```
MnemonicData:

ADD
     op=000000B

     mnem reg32, baseIndScale
          mnem reg [base+index*4]
          d=1B, s=1B, ss=10B, mod=00B, rm=100B
          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 `operandFieldEncodings` (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.

**The syntax of the operands (not separator commas) 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 there instead; it isn't important what this placeholder is, along as it is a single token consisting of letters and numbers only).**

**If a mnemonic has no operands** and so no fields in which operands are encoded then "`--`" should be used to indicate this to the assembler. For example:

```
syscall
     op = 1100B

     mnem
          --              ; no operand field encoding
          --              ; no local field encodings
          syscall
```

- `localFieldEncodings` must start with two tabs and is of the same format as `globalFieldEncodings`, `<fieldName>=<value><B/H/I>`. Multiple local field encodings must be separated by a comma (`,`). **If there are no local field encodings** then simply put "`--`" after the two tabs. For example:

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

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

- `instructionFormat` line 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. **These instruction names must be defined within InstructionFormat section**. For example:

```
MnemonicData:

ADD
     op=000000B

     mnem reg32, baseIndScale
```

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

InstructionFormat:

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

Then the entire instruction used to encode `ADD` operand format `mnem reg32, baseIndScale` is:

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

# 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 tokens “big” or “little” (not case sensitive).

- **Big** endian stores the most significant value in the sequence at the lowest storage address.

- **Little** endian stores the least significant value in the sequence at the lowest storage address.

# 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 token consisting only of an int variable. This int variable must be greater than 0.

# Assembly Source Code 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 Program 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:**

```
.data

; data here

.text

; assembly code here
```

# .data

The .data section of the assembly file provides various directives for reserving storage space for variables. It can be used to reserve as well as initialize data in one or more minimum addressable units.

The following are some examples:

```
.data

myInteger 6MAU 100
smallInteger 2MAU 4

storageAlloc 5MAU
reserve 10MAU

MyString .ascii "Hello, World"
goodbyeStr .ascii "Bye"
```

---

`myInteger 6MAU 100`

This allocates 6 minimum addressable units (as specified in MinAddressableUnit section within the specification file) for encoding of integer (100). This integer will be converted to its binary equivalent and stored within the object file as a hexadecimal value. The **address offset** is assigned to variable `myInteger`.

---

`storageAlloc 5MAU`

Here storage space of 5 minimum address units is **reserved for uninitialised data (this is zero initialised within the object file**). The **address offset** is assigned to variable `storageAlloc`.

```
------------------------------
```

```
MyString .ascii "Hello, World"
```

Each character in string `Hello, World` is converted to its ASCII value. Again this is stored within the object file in hexadecimal. The **address offset** is assigned to variable `MyString`.

# .text

**The text section (.text) must begin with:**

```
.text

; the assembly code must go here

ADD $t1, $t1, $t1
AND $t1, $t1, $t3
...
```

**The assembler analyses each assembly instruction on a line-by-line basis, and attempts to assemble it via the specification file that was provided. If successful the object code will be stored within the object file in its hexadecimal form, adjacent to its address (again in hexadecimal form).** For example:

```
0:      48 65 6C 6C 6F 2C 20 57 6F 72 6C 64
c:      24 02 00 04
10:     3C 01 10 01
```

# Output Files

The software will output two files: “spec_error_report.txt” and “object_code.txt”.

**If errors are found in the specification file then these will be shown within spec_error_report.txt.**

If there are no visible errors within the specification file, then the assembler 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_code.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_code.txt.**