moviAsm

Manual

Version 1.4.0 / 2018-06-28

Table of Contents

Copyright and Proprietary Information Notice

Movidius Ltd.
1730 South El Camino Real, Suite 200
San Mateo, CA 94402
http://www.movidius.com/

1. Introduction

1.1. Tool Overview

The Movidius Assembler (`moviAsm`) is the **Software Component** which is responsible for producing binary files in the specified format, acording to the latest **ISA** specifications.

1.2. Conventions used in this document

Words encapsulated between `<` and `>` are not keywords, but a description of an expected value. E.g. `<parameter>` shows that a parameter is required.

With `{ }` we denote a series of possible values. With `|` we separate the values of the series.

With `[ ]` we denote an optional parameter.

The command line arguments given to `moviAsm` can be separated by whitespace (spaces or tabs).

1.3. Usage example

`moviAsm` is used from the command line in the following manner:

```
moviAsm [<option> ...] <filename> [<filename> ...]
```

The order of the arguments is not important.

1.3.1. Input

`moviAsm` takes as input text files containing the assembler mnemonics, their parameters, comments, and preprocessor directives described in this document.

1.3.2. Output

`moviAsm` is able to produce outputs in the following formats:

- `.o` - Object file
  - A binary elf file representing the binary form of the input source file. It contains all the sections that would compose a valid `ELF` file.
- `.prf` - Assembly profiling information (triggered by the `-profile` flag)
  - A text file containing profiling information (number of registers and units used per instruction; instruction size; 5 cycle average instruction size; min, max, average for this metrics; etc.)

1.4. Arguments

An <option> starts with a - character and may be any of the assembler switches. The case of the letters is significant. The arguments should not be duplicated in the command line, except for -o when multiple input files are specified.

1.4.1. Available Arguments

Table 1. Available Arguments

Argument	Description	Notes
-version	Print the Tools package version	
-h[elp]	Print the **how to use** message	
-err[:<fileName>]	Output errors to a text file. Filename is moviAsmError.err or <fileName>.err if one is specified.	Only once
-cv:<chipVersion>	The chip version for which to assemble. Available versions: ma1100, ma1300, ma2100, ma2150, ma2155, ma2450, ma2455, ma2480, ma2485, ma2x5x	Only once
-v[erbose]	Display more verbose messages	There is only one extra level of verbosity
-L	Optimize labels adresses (align to 128 bits). Uses Slot promotion mechanism.	Only once
-o:<fileName>	The output file name	(Recommended) Use imediately after specifying the input file
-pp	Output the preprocessed asm file	Only once
-ccwarnings	Enable calling convention warnings	Only once
-i:<path>	Include <path> to the include directories	
-D<symbolName>[:<symbolValue>]	Define a preprocessor symbol on the command line; this symbol is visible for all the assembled files	
-list	Print lines while assembling; Shows opcode and ASM code for each line (or pipe)	Only once
-a	Disable the analyzer	
-debug	Debug enabled (output a .log file, which can be used for debugging)	Only once

Argument	Description	Notes
	`-showWarnings:<number>`	Limit the number of warning/info messages of a type (e.g. unaligned target) printed out. The `<number>` 'th warning prints that warning, then reports that other similar warnings are suppressed.
	`-nowarn[:<warning>]`	Disable non critical warnings. If `<warning>` is specified disable that warning group or specific warnings. Valid options: • `low` • `<warningID>`
	`-keepOutput`	Keep output file despite errors. Correctness of such files is not guaranteed.
Only once	`-noSlotAllPromo`	Disable slot promotion through `SLOT_ALL` for label alignment
Only once	`-noSPrefixing`	No **"S."** prefixing for shave sections
Only once	`-noCompressedNOPs`	Disable `NOP` compression in delay slots
Only once	`-noFinalSlotCompression`	No `NOP` compression in the **final** delay slot
Only once	`-g`	Enable debug information for hand written assembly
Only once	`-strictCstyleComments`	Throw error and fail assembly if non C-style comments used
	`-Werror[=asm|analyzer]`	Treat (groups of) warnings as errors

2. Slot Promotion

Slot promotion mechanism requires advancing to a bigger slot encoding (SLOT2 < SLOT4 < SLOT_ALL + padding if needed) in order to obtain the necessary alignment [1: Alignment using this mechanism can be achieved with -L command line argument or with .lalign preprocessor directive for the next opcode]. If alignment cannot be achieved with this mechanism, the opcode is reverted to its original form.

2.1. Specific chip version slot promotion

- **ma2100** - promotion to SLOT_ALL is disabled (HW limitation)
- **ma2150**, **ma2155**, **ma2450** - promotion to SLOT_ALL is limited (HW limitation):
  - should not surpass a limit of 31 bytes for the whole pipe (including padding)
  - slot header should not go over a boundary limit (aka slot header should not have an address ending in 0xD, 0xE, 0xF)
- **ma2480**, **ma2485** - no known limitation

2.2. Examples

- promotion from SLOT2 to SLOT4 for ma2100/ma2x5x/ma2x8x

```
0000000E:01:                        07 : NOP
0000000F:07: <label>:
0000000F:07:            000003000002A9 : LSU0.LDIL I0, 0x0
           :                            : || LSU1.LDIH I0, 0x0
```

is promoted into

```
0000000E:02:                      00AE : NOP
00000010:07: <label>:
00000010:07:            000003000002A9 : LSU0.LDIL I0, 0x0
           :                            : || LSU1.LDIH I0, 0x0
```

- promotion from SLOT2 to SLOT_ALL
  - no promotion for ma2100 (disabled slot promotion) or ma2x5x (goes over a 16-byte boundary)
  - promotion for ma2x8x (no restrictions)

```
0000000D:01:                        07 : NOP
0000000E:07: <label>:
0000000E:07:            000003000002A9 : LSU0.LDIL45 I0, 0x0
           :                            : || LSU1.LDIH I0, 0x0
```

is promoted into

```
0000000D:03:                    00000F : NOP
00000010:07: <label>:
00000010:07:            000003000002A9 : LSU0.LDIL I0, 0x0
           :                            : || LSU1.LDIH I0, 0x0
```

- promotion from `SLOT2` to `SLOT_ALL` with padding
  - no promotion for `ma2100` (disabled slot promotion)
  - with padding for `ma2x5x`/`ma2x8x` (no restrictions)

```
0000000C:01:                        07 : NOP
0000000E:07: <label>:
0000000E:07:            000003000002A9 : LSU0.LDIL45 I0, 0x0
           :                            : || LSU1.LDIH I0, 0x0
```

is promoted into

```
0000000C:04:                  0000001F : NOP
00000010:07: <label>:
00000010:07:            000003000002A9 : LSU0.LDIL I0, 0x0
           :                            : || LSU1.LDIH I0, 0x0
```

- promotion for compressed `NOP` s

```
0000000D:01:                        20 : SLOT_IDC[2] (NOP 3)
0000000E:07: <label>:
0000000E:07:            000003000002A9 : LSU0.LDIL I0, 0x0
           :                            : || LSU1.LDIH I0, 0x0
```

`NOP  3` is split into `NOP  2` and a slot promoted `NOP`

```
0000000D:01:                        10 : SLOT_IDC[1] (NOP 2)
0000000E:02:                      00AE : NOP
00000010:07: <label>:
00000010:07:            000003000002A9 : LSU0.LDIL I0, 0x0
           :                            : || LSU1.LDIH I0, 0x0
```

3. Programming Guidelines

The general structure of an `.asm` file should be:

```
.version <versionNumber>
    // set, macro, include, other preprocessor directives
.data
    ...
.code
    ...
.end
```

The (optional) `.version` directive at the beginning of the file signalizes the code version to the assembler file. The version string is in format: `MM.mm.bb.rrrr` (e.g.: 00.40.10):

- `MM`: major release
- `mm`: minor release
- `bb`: build number
- `rrrr`: revision number

The user may use some defines and includes before a `.data` directive. The user may use them at any place in the code, but it is more useful to define them in the beginning of the file.
The following are data and code sections.
An **example** is given below:

```
//-----------------------------------------
//The code is for ISAAC version 1.1
//-----------------------------------------
.version 00.40.10.2
.data initSection 0x40000000
        vFrameSize:
            .int 0x10, 0x10, 100, 100
.data colorSection 0x90101000
        vColorTable:
            .byte 0xFF, 0xFF, 0xFF  //white
            .byte 0x00, 0x00, 0x00  //black
.data frameSection 0x10008000
vBmpPointer:
            .incbin "frame1.bin"
.code entryPoint 0x1D000000
    LSU0.LDIL I0, vColorTable       //load pointer to colors
        || LSU1.LDIH I0, vColorTable
    LSU0.LDIH I1, vBmpPointer       //load bmpPointer
        || LSU1.LDIH I1, vBmpPointer
    //......
.end
```

As seen in the above example, an `.asm` file may contain sections which may be categorized in two main sections: **data sections and code sections**. Each of the sections may have an optional `<sectionName>` and an optional `<sectionDefaultAddress>`.

In the **data section**, the user may insert data which is downloaded to the target when the user loads the file into the target system (`moviSim`, `FPGA` or `ASIC`). The data section may contain images, test data, etc.

The **code section** contains the lines of code which form the executable segments that are downloaded to the target.

Each of the sections may also contain preprocessor directives, which are described in Section Preprocessor.

The code is organized in pipes (`VLIW` instructions) and preprocessor directives. A pipe may contain one or more operations, labels, or additional comments. Operations in a pipe are delimited with the `||` operator. A pipe may be described on a single line (operations separated by `||`) or on multiple lines. A line that starts with `||` and the first non-whitespace characters are considered as being part of the previous pipe. A line cannot end in `||`. It is recommendable to use the second version, as errors are reported using the line the error is found on, and the user may insert additional comments to each operation, not just to the entire pipeline. A pipe may also contain comments. A comment starts with the `//` characters. C style comments should be used for documenting the code using the doxygen tool.

4. Programming Conventions

4.1. Whitespaces

Whitespaces are any sequence of spaces and tabs. They are treated by the assembler as one space. This allows labels, mnemonics and parameters alignment in order to increase the readability of the source code.
For **example**, assembling:

```
LSU0.LDIL I1, 0x0000
    || LSU1.LDIL I2, 0x0002
```

is equivalent to assembling:

```
LSU0.LDIL           I1      0x0000
    || LSU1.LDIL    I2      0x0002
```

4.2. Comments

Comments start with the // character. Any sequence of characters following // character is ignored by the assembler.
For **example**, assembling:

```
 LSU0.LDIL I1, 0x0000
    || LSU1.LDIL I2, 0x0002 //end of line comment
//line comment
LSU0.LDIL I3, 0x0003
    || LSU1.LDIL I4, 0x0004 //end of line comment
```

is equivalent to assembling:

```
LSU0.LDIL I1, 0x0000 || LSU1.LDIL I2, 0x0002
LSU0.LDIL I3, 0x0003 || LSU1.LDIL I4, 0x0004
```

C-style multiline comments are also supported:

```
/*
* This can be a doc comment
*/
```

Also in-line comments can be used:

```
LSU0.LDIL /* aaa */I4, /* bbb */0x0000
```

4.3. Labels

Labels are any sequence of characters ending with a `:` character. The label definition line should not contain any operations. It must contain the label name followed by the `:` character and optionally a comment.
Defined labels may be used inside the code before and after their definitions. Labels are replaced with:

- their values for `LDIL` (lower 16 bits of the label address) and `LDIH` (higher 16 bits of the label address)
- the 14 bit offset between the current IP (Instruction Pointer) and the label's IP, for `BRA` operation

The absolute address of the labels is not known at assembly time (the linker knows the exact address of the label), thus the assembler considers, as a starting point, the address as being the default 0x1D000000: local code address (windowed address of the `SHAVE`).
A label is actually associated with an address. A label may point either to a `.data` section (in this case the label denotes variables and pointers), or to a .code section (pointer to the target code which is placed after the label definition).
Labels are useful for naming memory locations and the assembler schedules them in order to be resolved by the linker at link time.

Example:

```
EntryPoint:
    LSU0.LDIL I1, EndLabel
        || LSU1.LDIH I1, EndLabel
    BRU.JMP I1
    NOP 6
    ...
    ...
    ...
EndLabel:
    VAU.ADD.I32 V1, V2, V3
```

4.4. Symbols

Symbols defined in the assembler are processed as strings. They are case sensitive and, when used in expressions, the parser tries to evaluate them.

NOTE The symbols are replaced by the preprocessor of the assembler and can be inserted into expressions. Should not be confused with labels.

Example:

```
.set SymbolName 0xFFFF0000
    LSU0.LDIL I1, SymbolName
        || LSU1.LDIH I1, (SymbolName >> 16)
    //I1 will have the value 0xFFFF0000
```

4.4.1. Symbol Linkage

There are two types of symbols: **Local** and **Global**.

Symbols are **Global** by default and they are represented in the symbol table (ELF symtab).

Local symbols are split in two:

- Hidden
    - their names start with .L (e.g. .LfooBar)
    - they are not represented in the symtab
- Internal
    - their names start with .I (e.g. .IfooBar)
    - they are represented in the symtab with the .I prefix stripped (e.g. fooBar)

4.5. Numbers

The numbers are used when immediate values are required in the parameter list of the mnemonic, and they may be expressed in decimal (no prefix), hexadecimal (0x prefix) and binary (0b prefix).
For decimal numbers, an optional floating point conversion is possible. This is specified as follows:

- if the number is suffixed by F16, it is converted to its floating point format on 16 bits
- if the number is suffixed by F32, it is converted to its floating point format on 32 bits
- if the number has a decimal point, it is automatically converted to its FP32 value
- if the number doesn't have any suffix and it contains a decimal value, it is converted to its value on 32 bits. If the data representation of the number requires less bytes (e.g. .byte), the least significant bytes of the number are considered

The LDIL and LDIH operations load specified immediate values as follows:

- LDIL: always load the lower 16 bits of the specified value
- LDIH:
    - If the number can fit inside 16 bits, the number will be loaded as is
    - If the number cannot fit inside 16 bits, the upper 16 bits will be loaded

Example:

```
.set SymbolName1 0x41200000
.set SymbolName2 10F32
    LSU0.LDIL I1, SymbolName1
        || LSU1.LDIH I1, (SymbolName1 >> 16)
    LSU0.LDIL I2, SymbolName2
        || LSU1.LDIH I2, (SymbolName2 >> 16)
    //I1 and I2 will have the same values 0x41200000
```

4.6. Mnemonics

A mnemonic name or parameter may be expressed in any case (uppercase or lowercase). For the defined symbols, letter case is significant.
Parameters may be separated using any whitespace or the , character.
A special case of the mnemonics is NOP. The assembler also allows the mnemonic format:

```
NOP [n]
```

where n is the number of nops to be inserted in the current position. For example:

```
NOP 4
```

is the same as:

```
NOP
NOP
NOP
NOP
```

4.7. Vector Elements

The vector elements are specified for the operations that allow vector element selection. For these operations, the vector element must be specified in the following format:

```
<VRFRegister>.<vectorElement>
```

For operations that require SIMD vector elements (CPVCR, CPVRC), the vector element may be extended up to the 15th column and it must be expressed as a hex digit (0..9, A..F).

4.8. Assembler Aliases

Section 3 in the `SHAVE_ISA` document defines sufficient and precise information on the encoding of each opcode and the corresponding assembler. Frequently, there are more convenient ways of representing the same information. Here are the tables presenting these aliases:

Table 2. Condition code masks

What	Can use string instead of condition code masks

Applies to:	The `<imm3:msk>` field of the following instructions:
	`PEU.PC1I`
	`PEU.PC1S`
	`PEU.PC1C`
	`PEU.PCNS`
	`PEU.PCNC`
	`PEU.PCXS`
	`PEU.PCXC`
	`PEU.PVEC`
	`PEU.PEN4`
	`PEU.PEN8`

Details:	`NONE <-> 0x0`
	`EQ   <-> 0x1`
	`GT   <-> 0x2`
	`GTE  <-> 0x3`
	`LT   <-> 0x4`
	`LTE  <-> 0x5`
	`NEQ  <-> 0x6`

Example:	`PEU.PC1I EQ || IAU.ADD I1, I2, I3`

Table 3. TRF register addresses

What	Can use string instead of TRF register addresses

Applies to:	The `<trf:src>` and/or `<trf:dst>` fields of the following instructions:
	`CMU.CPZT`
	`CMU.CMTI`
	`CMU.CPIT`
	`CMU.CPTI`

Details:	P_GPR <-> 0x1f P_GPI <-> 0x1e P_SVID <-> 0x1c P_CFG <-> 0x1b G_GALOIS <-> 0x15 B_SREPS <-> 0x14 C_CMU1 <-> 0x13 C_CMU0 <-> 0x12 C_CSI <-> 0x11 F_AE <-> 0x10 I_AE <-> 0x0F V_ACC3 <-> 0x0D V_ACC2 <-> 0x0C V_ACC1 <-> 0x0B V_ACC0 <-> 0x0A S_ACC <-> 0x09 V_STATE <-> 0x08 S_STATE <-> 0x07 I_STATE <-> 0x06 B_RFB <-> 0x05 B_STATE <-> 0x04 B_MREPS <-> 0x03 B_LEND <-> 0x02 B_LBEG <-> 0x01 B_IP_0 <-> 0x00
Example	CMU.CPIT S_ACC I1

Table 4. Using segment names in absolute addressing

What	Can use string instead of segment number
Applies to:	The <imm3:seg> field of the following instructions: LSUx.LDA LSUx.STA LSUx.PFA
Details:	WINDOW_A <-> 0x0 WINDOW_B <-> 0x1 WINDOW_C <-> 0x2 WINDOW_D <-> 0x3 SLICE_LOCAL <-> 0x4 SHAVE_LOCAL <-> 0x4 (for Myriad 2 or higher)
Example:	LSU0.LDA32 I16, WINDOW_A, 0xC

Table 5. FL_IMM numbers

What	Can use string instead of FL_IMM number
Applies to:	The <FL_IMM> field of the following instructions: SAU.ADD VAU.ADD SAU.SUB VAU.SUB SAU.MUL VAU.MUL

Details:	`0.5    <-> 0x0`
	`1.0    <-> 0x1`
	`2.0    <-> 0x2`
	`3.0    <-> 0x3`
	`SQT2   <-> 0x4`
	`RQT2   <-> 0x5`
	`PI     <-> 0x6`
	`E      <-> 0x7`
	`10.0   <-> 0x8`
	`128.0  <-> 0x9`
	`255.0  <-> 0xA`
	`-0.5   <-> 0x10`
	`-1.0   <-> 0x11`
	`-2.0   <-> 0x12`
	`-3.0   <-> 0x13`
	`-SQT2  <-> 0x14`
	`-RQT2  <-> 0x15`
	`-PI    <-> 0x16`
	`-E     <-> 0x17`
	`-10.0  <-> 0x18`
	`-128.0 <-> 0x19`
	`-255.0 <-> 0x1A`

Example	`SAU.ADD.F32 S1, S2, PI`

Table 6. Operation aliases

What	Can use string instead of FL_IMM number

Details	BRU.RPI I31 <-> BRU.RPI I31, 0x0 BRU.NOP [n] <-> NOP[n] SAU.MUL.U8F I31, I31, I31 <-> SAU.ILMULH.U8 I31, I31, I31 VAU.MUL.U8F V31, V31, V31 <-> VAU.ILMULH.U8 V31, V31, V31 IAU.FINSJ I31, I31, I31 <-> IAU.FINS I31, I31, I31 IAU.FEXTI I31, I31, 0x1F, 0xF <-> IAU.FEXT.I32 I31,I31,0x1F,0xF IAU.FEXTJI I31, I31, I31 <-> IAU.FEXT.I32 I31, I31,I31 IAU.FEXTU I31, I31, 0x1F, 0xF <-> IAU.FEXT.U32 I31, I31, 0x1F, 0xF IAU.FEXTJU I31, I31, I31 <-> IAU.FEXT.U32 I31, I31, I31 LSUx.LDI64.H V31, I31, I31 <-> LSUx.LD64.H V31, I31, I31 LSUx.LDI64.L V31, I31, I31 <-> LSUx.LD64.L V31, I31, I31 LSUx.LDI128.U8.F16 V31, I31, I31 <-> LSUx.LD128.U8.F16 V31, I31, I31 LSUx.LDI128.U8F.F16 V31, I31, I31 <-> LSUx.LD128.U8F.F16 V31, I31, I31 LSUx.LDI128.F16.F32 V31, I31, I31 <-> LSUx.LD128.F16.F32 V31, I31, I31 LSUx.LDI128.U16.U32 V31, I31, I31 <-> LSUx.LD128.U16.U32 V31, I31, I31 LSUx.LDI128.I16.I32 V31, I31, I31 <-> LSUx.LD128.I16.I32 V31, I31, I31 LSUx.LDI128.U8.U16 V31, I31, I31 <-> LSUx.LD128.U8.U16 V31, I31, I31 LSUx.LDI128.I8.I16 V31, I31, I31 <-> LSUx.LD128.I8.I16 V31, I31, I31 LSUx.LDI32R V31, I31, I31 <-> LSUx.LD32R V31, I31, I31 LSUx.LDI16R V31, I31, I31 <-> LSUx.LD16R V31, I31, I31 LSUx.LDI8R V31, I31, I31 <-> LSUx.LD8R V31, I31, I31 LSUx.LDI128.RGB565 V31, I31, I31 <-> LSUx.LD128.RGB565 V31, I31, I31 LSUx.LDI128.RGB1555 V31, I31, I31 <-> LSUx.LD128.RGB1555 V31, I31, I31 LSUx.LDI128.RGB4444 V31, I31, I31 <-> LSUx.LD128.RGB4444 V31, I31, I31 LSUx.LDI64 S31, I31, I31 <-> LSUx.LD64 S31, I31, I31 LSUx.LDI32 S31, I31, I31 <-> LSUx.LD32 S31, I31, I31 LSUx.LDI16 S31, I31, I31 <-> LSUx.LD16 S31, I31, I31 LSUx.LDI8 S31, I31, I31 <-> LSUx.LD8 S31, I31, I31 LSUx.LDI64 I31, I31, I31 <-> LSUx.LD64 I31, I31, I31 LSUx.LDI32 I31, I31, I31 <-> LSUx.LD32 I31, I31, I31 LSUx.LDI16 I31, I31, I31 <-> LSUx.LD16 I31, I31, I31 LSUx.LDI8 I31, I31, I31 <-> LSUx.LD8 I31, I31, I31 LSUx.LDI32.I16.I32 I31, I31, I31 <-> LSUx.LD32.I16.I32 I31, I31, I31 LSUx.LDI32.I8.I32 I31, I31, I3 <-> LSUx.LD32.I8.I32 I31, I31, I31 LSUx.LDI32.U16.U32 I31, I31, I31 <-> LSUx.LD32.U16.U32 I31, I31, I31 LSUx.LDI32.U8.U32 I31, I31, I31 <-> LSUx.LD32.U8.U32 I31, I31, I31 LSUx.STI64.H V31, I31, I31 <-> LSUx.ST64.H V31, I31, I31 LSUx.STI64.L V31, I31, I31 <-> LSUx.ST64.L V31, I31, I31 LSUx.STI128.U8.F16 V31, I31, I31 <-> LSUx.ST128.U8.F16 V31, I31, I31 LSUx.STI128.U8F.F16 V31, I31, I31 <-> LSUx.ST128.U8F.F16 V31, I31, I31 LSUx.STI128.F16.F32 V31, I31, I31 <-> LSUx.ST128.F16.F32 V31, I31, I31 LSUx.STI128.U16.U32 V31, I31, I31 <-> LSUx.ST128.U16.U32 V31, I31, I31 LSUx.STI128.I16.I32 V31, I31, I31 <-> LSUx.ST128.I16.I32 V31, I31, I31 LSUx.STI128.U8.U16 V31, I31, I31 <-> LSUx.ST128.U8.U16 V31, I31, I31 LSUx.STI128.I8.I16 V31, I31, I31 <-> LSUx.ST128.I8.I16 V31, I31, I31

Details	LSUx.STI32R V31, I31, I31 <-> LSUx.ST32R V31, I31, I31 LSUx.STI16R V31, I31, I31 <-> LSUx.ST16R V31, I31, I31 LSUx.STI8R V31, I31, I31 <-> LSUx.ST8R V31, I31, I31 LSUx.STI128.RGB565 V31, I31, I31 <-> LSUx.ST128.RGB565 V31, I31, I31 LSUx.STI128.RGB1555 V31, I31, I31 <-> LSUx.ST128.RGB1555 V31, I31, I31 LSUx.STI128.RGB4444 V31, I31, I31 <-> LSUx.ST128.RGB4444 V31, I31, I31 LSUx.STI64 S31, I31, I31 <-> LSUx.ST64 S31, I31, I31 LSUx.STI32 S31, I31, I31 <-> LSUx.ST32 S31, I31, I31 LSUx.STI16 S31, I31, I31 <-> LSUx.ST16 S31, I31, I31 LSUx.STI8 S31, I31, I31 <-> LSUx.ST8 S31, I31, I31 LSUx.STI64 I31, I31, I31 <-> LSUx.ST64 I31, I31, I31 LSUx.STI32 I31, I31, I31 <-> LSUx.ST32 I31, I31, I31 LSUx.STI16 I31, I31, I31 <-> LSUx.ST16 I31, I31, I31 LSUx.STI8 I31, I31, I31 <-> LSUx.ST8 I31, I31, I31 LSUx.STI32.I16.I32 I31, I31, I31 <-> LSUx.ST32.I16.I32 I31, I31, I31 LSUx.STI32.I8.I32 I31, I31, I3 <-> LSUx.ST32.I8.I32 I31, I31, I31 LSUx.STI32.U16.U32 I31, I31, I31 <-> LSUx.ST32.U16.U32 I31, I31, I31 LSUx.STI32.U8.U32 I31, I31, I31 <-> LSUx.ST32.U8.U32 I31, I31, I31 LSUx.SWZ4V [3333], [3333] <-> LSUx.SWZ4V [3333], [3333],[PPPP] LSUx.SWZV8 [77777777] <-> LSUx.SWZV8 [77777777], [PPPP]

5. Code Analyzer

The code analyzer is a tool used to check for violations in asm code at assembly time. It checks for possible read and write port sharing violations and for write operations on the same register file (writeback clash), and outputs messages for each possible error. For more information, refer to the Port Sharing section in the `SHAVE_ISA` file. The code analyzer is enabled by default. It may be disabled by passing a `–a` on the command line of the assembler.

Limitations:

```
This is a static code analyzer. When analyzing lines containing jump instructions, the program may issue incorrect
warnings after the jump as it does not analyze the target of the jump, but instead it just performs linear code
analyzing. The same is true about predicated operations: the assembler cannot know the result of the predication.
This can be done at runtime only.
```

6. Preprocessor

The purpose of the preprocessor is to prepare the code in order to be assembled. It includes features like powerful macro language, conditional assembly, object code formatting, etc.

6.1. Preprocessor Conventions

6.1.1. Keywords

The preprocessor keywords start with a `.` (dot) character and they are in lowercase.

6.1.2. Symbols

A symbol is a name which is given a value. The preprocessor allows symbol definition and re-definitions in order to allow the user to build a more flexible code.

In the code, when preprocessor meets a symbol, it searches for it in the internal symbol table, which is the place where symbols are stored. If the symbol is found, it is replaced by its value. If not, the preprocessor sends an error message.

A symbol can be added into the symbol table using the `.set` directive and removed from the table using the `.unset` directive.

The symbol names are case sensitive.

Both local and global symbols can be defined also through symbol arithmetic in the form of `<symbol> + <offset>`.

E.g.:

```
.label .Lsym + 0x4
.word sym + 13
```

6.1.3. String literals

String literals **must** be enclosed in double quotes.

6.1.4. Expressions

The preprocessor evaluates expressions.

An expression may contain operators and operands (numbers or preprocessor symbols).

An expression must be enclosed in round parentheses; e.g.: `(5 + 4 * 2)`.

When an expression is met, first the defined symbols are replaced and then the expression is evaluated.

The supported operators are defined similar to the C ones (this means that also the priority and meaning is the same as in C):

Table 7. Expression Operators

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
%	modulo (remainder)
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
!=, <>	not equal
<	bitwise left shift
>>	bitwise right shift
^	bitwise XOR
~	bitwise NOT
&	bitwise AND
|	bitwise OR
!	logical NOT
&&	logical AND
||	logical OR

Operator	Description
**	exponentiation

The priority of operations is defined as follows, in order from higher priority to lower:

Table 8. Operator Priority

Priority	Operators
14:	~, !, **
13:	*, /, %
12:	+, -
11:	<<, >>
10:	<, <=, >, >=
9:	==, !=, <>
8:	&
7:	^
6:	|
5:	&&
4:	||

The following operators have right associative: **, ~, !.

The order of evaluation may be changed using rounded parentheses.

6.2. Preprocessor Directives

6.2.1. Conditional Directives

The main purpose of these directives is to conditionally assemble one or more blocks of data.

6.2.1.1. .if

Syntax:

```
.if <absoluteExpression>
```

Description:

Marks the beginning of a section of code which is only considered part of the source program being assembled if the argument (which must be an absolute expression) is nonzero.

The end of the conditional section of code must be marked by `.endif`.

Optionally, the user may include code for the alternative condition, flagged by `.else`.

If the user has several conditions to check, `.elseif` may be used to avoid the nesting blocks if/else within each subsequent `.else` block.

6.2.1.2. .else

Syntax:

```
.else
```

Description:

`.else` is part of the support for conditional assembly. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

6.2.1.3. .elseif

Syntax:

```
.elseif <condition>
```

Description:

`.elseif` is part of the support for conditional assembly. It is shorthand for beginning a new `if` block that would otherwise fill the entire `.else` section.

6.2.1.4. .endif

Syntax:

```
.endif
```

Description:

Marks the end of a conditional block.

6.2.1.5. .ifdef

Syntax:

```
.ifdef <symbolName>
```

Description:

If the spesified symbol has been defined, the following section of code gets assembled.

6.2.1.6. .ifc

Syntax:

```
.ifc "<string1>", "<string2>"
```

Description:

Assembles the following section of code if the two strings are the same. The strings must be enclosed in double quotes. The string comparison is case sensitive.

6.2.1.7. .ifeq

Syntax:

```
.ifeq <absoluteExpression>
```

Description:

Assebles the following section of code if the argument is zero.

6.2.1.8. .ifeqs

Syntax:

```
.ifeqs "<string1>", "<strig2>"
```

Description:

Same as `.ifc`

6.2.1.9. .ifge

Syntax:

```
.ifge <absoluteExpression>
```

Description:

Assebles the following section of code if the argument is greather than or equal to zero.

6.2.1.10. .ifgt

Syntax:

```
.ifgt <absoluteExpression>
```

Description:

Assebles the following section of code if the argument is greather than zero.

6.2.1.11. .ifle

Syntax:

```
.ifle <absoluteExpression>
```

Description:

Assebles the following section of code if the argument is less than or equal to zero.

6.2.1.12. .iflt

Syntax:

```
.iflt <absoluteExpression>
```

Description:

Assebles the following section of code if the argument is less than zero.

6.2.1.13. .ifnc

Syntax:

```
.ifnc "<string1>", "<string2>"
```

Description:

Like `.ifc`, but the sense of the test is reversed.

6.2.1.14. .ifndef

Syntax:

```
.ifndef <symbolName>
```

Description:

Assembles the following section of code if the specified symbol has not been defined. Identical with `.ifnotdef`.

6.2.1.15. .ifnotdef

Syntax:

```
.ifnotdef <symbolName>
```

Description:

Assembles the following section of code if the specified symbol has not been defined. Identical with `.ifndef`.

6.2.1.16. .ifne

Syntax:

```
.ifne <absoluteExpression>
```

Description:

Assembles the following section of code if the argument is not equal to zero (in other words, this is equivalend to `.if`).

6.2.1.17. .ifnes

Syntax:

```
.ifnes <string1>, <string2>
```

Description:

Like .ifeqs, but the sense of the test is reversed. This assembles the following section of code if the two strings are not the same.

6.2.2. Object Directives

6.2.2.1. .code

Syntax:

```
.code [<sectionName>]
```

Description:

Mark the beginning of a code section. The segment ends with the `.end` directive or when the start of another section is met. If not specified, default `<sectionName>` is `.text`.

The optional argument `<sectionName>` denotes the name of the secion and it can be used for relocating different secion from the linker command line. If there are two or more section with the same name, they are concatenated, starting with the first one that is met during the link process.

The concatenation of the sections is done according to their alignment specifiers. Default alignment of the section is 1 byte (meaning that the sections are unaligned). The alignment may be useful for `.data` sections. For `.code` sections this may be dangerous as the padding is performed using zeroes - for alignment of the code, specific alignment specifiers should be used (which insert `ESC_PAD` slots).

6.2.2.2. .data

Syntax:

```
.data [<sectionName>]
```

Description:

`.data` tells the assembler to assemble the following statements onto the end of the data subsection numbered subsection (which is an absolute expression). If the subsection is omitted.it defaults to zero.

The optional argument `<sectionName>` denotes the name of the section and it can be used for relocating different sections from the linker command line. If there are 2 or more sections with the same name, they are concatenated, starting with the first one that is met during the link process.

The concatenation of the sections is done according to their alignment specifiers. Default alignment of the section is 1 byte (meaning that the sections are unaligned). The alignment may be useful for `.data` sections. For `.code` sections this may be dangerous as the padding is performed using zeroes – for alignment of the code specific alignment specifiers should be used (which insert `ESC_PAD` slots).

6.2.2.3. .end

Syntax:

```
.end
```

Description:

`.end` marks the end of the assembly file. The assembler does not process anything in the file past the `.end` directive. The current section is dumped into the object file and is closed.

6.2.2.4. .section

Syntax:

```
.section <name> [, "<flags>"[, @<type>]]
```

Description:

Marks the beginning of a section. The segment ends either when another section directive is found, either at the end of the file (which may optionally be specified using the `.end` directive).

The argument `<name>` denotes the name of the section. If there are two or more section with the same name, they are concatenated during the link-editing process in the order they are found in the elf object file.

The concatenation of the sections is done by the link-editor according to their alignment specifiers. Default alignment of the section is 1 byte (meaning that the sections are unaligned).

The optional `<flags>` argument is a quoted string which may contain any combination of the following characters:

- `a` section is allocatable
- `w` section is writeable
- `x` section is executable

The optional `<type>` argument may contain one of the following constants:

- `@progbits` section contains data
- `@nobits` section does not contain data (i.e., section only occupies space)

6.2.2.5. .salign

Syntax:

```
.salign <numberOfBytes>
```

Description:

`.salign` (section align) may be used for aligning the current section to the specified number of bytes. The alignment is performed by inserting zeroes until the specified alignment is reached. When sections are concatenated (they contain the same name or the same address) an alignment is also performed. The `.salign` directive shouldn't be used for `.code` sections unless there are no other sections to concatenate with (with the same name or the same start address), as there are zeroes inserted for padding (0 is not a valid shave opcode). For aligning opcodes, other alignment directives should be used (e.g. `.lalign`, `.128`, etc.)

NOTE This directive is deprecated and will soon be removed.

Example:

```
.data testData 0x40000000
    byteData:
        .byte 0x10              // 0x40000000
    intData:
        .int 0x11223344         // 0x40000001
.data testData
    secondInt:
        .int 0x12345678         // 0x40000005
.data anotherData
.salign 4
    thirdInt:
        .int 0x87654321         // 0x4000000C
        .byte 0x10              // 0x40000010
.data anotherData 0x40000000
    fourthInt:
        .int 0x11111111         // 0x40000014
        .byte 0x20              // 0x40000018
.data regularData
.salign 4
    regularInt:
        .int 0x22222222         // 0x1C000000
        .byte 0x30              // 0x1C000004
    .data regularData
.salign 4
        anotherInt:
            .int 0x22222222     // 0x1C000008
            .byte 0x30          // 0x1C00000C
```

6.2.2.6. .align

Syntax:

```
.align <numberOfBytes>
```

Description:

`.align` specifies the alignment of the elements inside a section to the specified number of bytes for the next label/element. An element starts with a data storage preprocessor directive. The `.align` directive has 1 as the default value (8 bits, as it is the default value). Alignment is performed using 0s. Usually the `.align` directive is used to make sure the data pointed by a specified label is aligned to a specific number of bytes, in order to optimize the fetching of the data pointed by that label.

Example:

```
.data
    var1:                   // 0x1C000000
        .int  0x12345678    // 0x1C000000
        .byte 0x9A          // 0x1C000004
    var2:                   // 0x1C000005
        .int  0xBCDEF012    // 0x1C000005
        .align 4
    var3:                   // 0x1C00000C
        .int  0x3456789A    // 0x1C00000C
        .byte 0xBC          // 0x1C000010
    var4:                   // 0x1C000011
        .int  0xDEF01234    // 0x1C000011
        .byte 0x56          // 0x1C000015
        .align 4
    var5:                   // 0x1C000018
        .int  0x78ABCDEF    // 0x1C000018
        .byte 0xAA          // 0x1C00001C
        .align 4
    var6:                   // 0x1C000020
        .byte 0x01, 0x02    // 0x1C000020
        .byte 0x03          // 0x1C000022
    var7:                   // 0x1C000023
        .byte 0x04          // 0x1C000023
    ...
```

6.2.2.7. .lalign

Syntax:

```
.lalign
```

Description:

Label alignment can be useful when the user does not want to use the label optimization switch in the assembler command line invocation, which aligns all labels to 128 bits, but still wishes to align some particular labels to 128 bits. This can be used inside `.code` body only. For `.data` sections, the `.align` directive should be used.

Slot promotion is performed. This means that the previous opcode will be encoded using the next possible encoding (slot 4 or slot all), until alignment is reached. When reaching the slot all encoding, it will specify the number of bytes in the padding field of the header needed in order to attain the alignment. MoviAsm will also encode those bytes at the end of that previous opcode.

Example:

```
0000000B:05:          010200080E : CMU.CMII.I32 I0, I1
    // naturally, CMU.CMII would be encoded as 0x01020081 on slot2_A
    // instead, it is encoded on slot4_A to obtain the alignment
.lalign
0x00000010:10: <alignedOpcode>:    LSU0.LDIL I1, expectedAlignedOpcode
```

NOTE Expect only an increase in the code size, but not in the number of cycles.

6.2.2.8. .128

Syntax:

```
.128
```

Description:

Start a 128-bit block. This is a block which is executed much faster than the code outside this block. This can be useful for code that needs to be executed faster than the rest of the code. The penalty of this is that the operations inside this block cannot have an opcode greater than 128 bits. This restriction means that the user cannot use the entire opcode slot, so too many operations cannot reside in the same pipe. If the opcode exceeds 128 bits, an error is signalized and the opcode is not written to the output file.

The 128 bits block ends with a `.endb` directive.

6.2.2.9. .64

Syntax:

```
.64
```

Description:

Start a 64-bit block. This is a block which is executed much faster than the code outside this block. This can be useful if the user has code that needs to be executed faster than the rest of the code. The penalty of this is that the operations inside this block cannot have the opcode greater than 64 bits. This restriction means that the user cannot use the entire opcode slot, so too many operations cannot reside in the same pipe. If the opcode exceeds 64 bits, an error is signalized and the opcode is not written to the output file.

The 64-bit block ends with a `.endb` directive.

6.2.2.10. .endb

Syntax:

```
.endb
```

Description:

End a 128 or 64-bit block started with a `.128` or `.64` directive.

6.2.2.11. .noload

Syntax:

```
.noload
```

Description:

Mark the current section to occupy no space in the object file. This is usually the case of the bss sections (uninitialized data). Same as `.bss`.

6.2.2.12. .bss

Syntax:

```
.bss
```

Description:

Mark the current section to occupy no space in the object file. This is usually the case of the bss sections (uninitialized data). Same as `.noload`.

6.2.3. Section Directives

6.2.3.1. .byte

Syntax:

```
.byte <expressions>
```

Description:

`.byte` expects zero or more expressions, separated by commas. Each expression is assembled into the next byte.

6.2.3.2. .ascii

Syntax:

```
.ascii <string>[(,| )<string>...]
```

Description:

Expects a string, string literal or list of strings (literals).
It assembles the strings without trailing zero.
Strings with spaces inside must be delimited by quotes. When the string contains quotes inside it, quotes need to be prefixed by \.
moviAsm accepts the following escape sequences:

Table 9. Escape sequence

Escape Sequence	Description
\'	Single quote
\"	Double quote
\\	Backslash
\0	Null character
\a	Audible bell
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\xnnn	Hexadecimal number (nnn)

6.2.3.3. .asciiz

Syntax:

```
.asciiz <strings>
```

Description:

`.asciiz` is just like `.ascii`, but the string is followed by a zero byte. The `z` in `.asciiz` stands for `zero`.

6.2.3.4. .string

Syntax:

```
.string "str"
```

Description:

Same as `.asciiz`.

6.2.3.5. .hword

Syntax:

```
.hword <expressions>
```

Description:

This expects zero or more expressions, and emits a 16-bit number for each. This directive is a synonym for `.short`.

6.2.3.6. .short

Syntax:

```
.short <expressions>
```

Description:

`.short` is normally the same as `.word`.

6.2.3.7. .single

Syntax:

```
.single <floatNumbers>
```

Description:

This directive assembles zero or more floating point numbers, separated by commas. It has the same effect as `.float`.

6.2.3.8. .equ

Syntax:

```
.equ <symbol> <expression>
```

Description:

This directive sets the value of symbol to expression. It is synonymous with `.set`.

6.2.3.9. .float

Syntax:

```
.float <floatNumbers>
```

Description:

This directive assembles zero or more float32 numbers, separated by commas. The exact kind of floating point numbers emitted depends on how the preprocessor is configured. Numbers may be expressed in any base. For storing the actual value of the float numbers, the user has to use decimal point or the `F32` suffix.

6.2.3.10. .float32

Syntax:

```
.float32 <floatNumbers>
```

Description:

`.float32` is the same as `.float`.

6.2.3.11. .float16

Syntax:

```
.float16 <floatNumbers>
```

Description:

This directive assembles zero or more float16 numbers, separated by commas. The exact kind of floating point numbers emitted depends on how the preprocessor is configured. For storing the floating point values, use the `F16` suffix.

6.2.3.12. .int

Syntax:

```
.int <expressions>
```

Description:

Expect one or more expressions, of any section, separated by commas. For each expression, emit a number that, at run time, is the value of that expression.

6.2.3.13. .word

Syntax:

```
.word <expressions>
```

Description:

This directive expects zero or more expressions, of any section, separated by commas.

6.2.3.14. .long

Syntax:

```
.long <expressions>
```

Description:

`.long` is the same as `.int`.

6.2.3.15. .label

Syntax:

```
.label <labelName>
```

Description:

Construct a jumptable. A jumptable is a `.data` section which contains, instead of initialized data, an array of 32 bit elements which contain the values of the specified labels. A jumptable may be used for implementation of switch-case constructs. The linker fills the entries in the jumptables after the symbol resolution.

Example:

```
.data
jumpTable:
    .label case1
    .label case2
    .label case3
    ...
.code entryPoint 0x1D000000
start:
lsu0.ldil i0, jumpTable
                || lsu1.ldih i0, jumpTable
            iau.add i0, i0, i1          //assume that i1 = caseNo * 4
...
bru.bra i0
```

6.2.3.16. .fill

Syntax:

```
.fill <repeat> [, <size>[ , value]]
```

Description:

`<repeat>`, `<size>` and `<value>` are absolute expressions. This emits repeat copies of `<size>` bytes. `<repeat>` may be zero or more. `<size>` may be zero or more, but if it is more than 8, then it is deemed to have the value 8, compatible with other assemblers. If `<value>` cannot be represented on the specified number of bytes (specified by the `<size>` value), the least significant bytes are considered.
`<size>` and `<value>` are optional. If the second comma and `<value>` are absent, value is assumed zero. If the first comma and following tokens are absent, `<size>` is assumed to be 1.

6.2.3.17. .include

Syntax:

```
.include <filename>
```

Description:

This directive provides a way to include supporting files at specified points in the source program. The code from file is assembled as if it followed the point of the .include. When the end of the included file is reached, assembly of the original file continues. The user can control the search paths used with the `-i` command-line option. Quotation marks are not required around the file, as only one file per `.include` directive may be included.
If the included file is not found, an error message is generated.

6.2.3.18. .incbin

Syntax:

```
.incbin <filename>
```

Description:

The `.incbin` directive includes file verbatim in the current data section. The `<fileName>` may contain spaces. If the specified file cannot be found, an error message is generated. It is required for a label to be specified before this directive so that the included data can be addressed for future reference.

6.2.4. Symbol Directives

6.2.4.1. .set

Syntax:

```
.set <symbol> <expression>
```

Description:

Set the value of symbol to expression. This changes the symbol's value and type to conform to the expression. A symbol may not be `.set` many times in the same assembly.

If the symbol value contains an expression, it is evaluated at preprocessing time. If the expression is quoted, it is stored as a string (not evaluated).

6.2.4.2. .unset

Syntax:

```
.unset <symbol>
```

Description:

Undefines the specified symbol (removes it from the parser's symbol table). This allows the user to redefine symbols (same symbol name may be used with different values).

6.2.4.3. .unsetall

Syntax:

```
.unsetall
```

Description:

Undefines all the parser symbols defined up to a point in the code.

6.2.4.4. .size

Syntax:

```
.size <symbol>, <value>
```

Description:

Specifies the size of an existing symbol in the current file. Please note that the value can be calculated from an expression or from label arithmetic, but it could be also an actual number.

Example:

```
vc:
    .fill (8*8), 4, 0
    .size vc, . -vc
```

6.2.4.5. .type

Syntax:

```
.type <symbol>, <@function | @object>
```

Description:

Specifies the type of an existing symbol in the current file (the values supported are @function and @object).

Example:

```
z:
    .fill 1, 4, 0
    .type z, @object
    .size z, 4
```

6.2.4.6. .alias

Syntax:

```
.alias <new_symbol> <existing_symbol>
```

Description:

Defines a copy of a symbol existing in the current file. There are several combinations, although the most common ones will have the same symbol type (local to local/global to global). Please note that local symbols are marked with `.L` in the beginning of the name.

Example:

```
.alias .L_local_symbol  global_symbol // assuming that global_symbol is defined in the current file
.alias global_symbol  .L_local_symbol
```

6.2.4.7. .weak

Syntax:

```
.weak <symbol1>[, <symbol2>, ..]
```

Description:

This directive sets the weak attribute on the comma separated list of symbol names. If the symbols do not already exist, they will be created.

Example:

```
.weak add, main
```

6.2.5. Macro Directives

A macro is a rule or a pattern that specifies how a certain input sequence (often a sequence of characters) should be mapped to the output sequence according to a defined procedure. The following macro directives are supported for the current version of `moviAsm` preprocessor:

6.2.5.1. .macro

Syntax:

```
.macro <macname>
.macro <macname> <macargs ...>
```

Description:

The commands `.macro` and `.endm` allow the user to define macros that generate assembly output. The name of the macro is specified right after the keyword .macro. If the macro definition requires arguments, their names should be specified after the macro name, separated by commas. The argument `\@` maintains a counter of how many macros it has executed; you can copy that number to your output with `\@`, but only within a macro definition. The user can supply a default value for any macro argument by following the name with `=<deflt_value>`.
For example, these are all valid `.macro` statements:

```
.macro comm
```

Begin the definition of a macro called `comm`, which takes no arguments.

```
.macro plus1 p, p1
.macro plus1 p p1
```

Either statement begins the definition of a macro called `plus1`, which takes two arguments. Within the macro definition, `\p` or `\p1` are used to evaluate the arguments.

```
.macro reserve_str p1=0 p2
```

Begin the definition of a macro called reserve_str, with two arguments. The first argument has a default value, but not the second.
After the definition is complete, you can call the macro either as `reserve_str a,b` (with `\p1` evaluating to a and `\p2` evaluating to b), or as `reserve_str ,b` (with `\p1` evaluating as the default, in this case `0`, and `\p2` evaluating to b).

When the user calls a macro, argument values can be specified either by position or by keyword. For example, `sum 9,17` is equivalent to `sum to=17, from=9`.
If the user wants a parameter to take its default value, when the macro is called, `-` should be entered as its value.
For example, this definition specifies a macro `ZERO_IRF` that puts `0x0000` value into the specified range of registers in `IRF`:

```
.macro ZERO_IRF from=0 to=5
LSU1.LDIL I\from, 0x0000
.if (\to-\from)
ZERO_IRF (\from+1),\to
.endif
.endm
```

With that definition, `ZERO_IRF 0,5` is equivalent to this assembly input:

```
LSU1.LDIL I0, 0x0000
LSU1.LDIL I1, 0x0000
LSU1.LDIL I2, 0x0000
LSU1.LDIL I3, 0x0000
LSU1.LDIL I4, 0x0000
LSU1.LDIL I5, 0x0000
```

For a macro definition like:

```
.macro ldirf irf, value
lsu0.ldil i\irf, \value
            || lsu1.ldih i\irf, \value
.endm
```

An example of using the defined macro in a code sequence is presented below:

```
.data
    data1:
        .int 10, 20, 30
    data2:
        .int 20
.code
    start:
        ldirf 0, data1
        ldirf 1, data2
        ...
```

6.2.5.2. .endm

Syntax:

```
.endm
```

Description:

Exit early from the current macro definition.

6.2.5.3. .exitm

Syntax:

```
.exitm
```

Description:

Exit early from the current macro definition.

6.2.5.4. .purgem

Syntax:

```
.purgem <macroName>
```

Description:

Undefine the specified macro, so that later uses of the string are not expanded.

6.2.6. Repeat Directives

These directives allow the user to repeat, parametrically or not, a sequence of code. The current version of preprocessor supports the following repeat directives:

6.2.6.1. .irp

Syntax:

```
.irp <symbol> <values. . .>
```

Description:

Evaluate a sequence of statements assigning different values to symbol. The sequence of statements starts at the .irp directive, and is terminated by an .endr directive. For each value, symbol is set to value, and the sequence of statements is assembled. If no value is listed, the sequence of statements is ignored. To refer to a symbol within the sequence of statements, \symbol should be used. The values may be double quoted strings, too. No other preprocessor directive is allowed between .irp and <<__endr, .endr>>. If the user does not specify any values for the symbol, the sequence is ignored by the assembler. The symbol has visibility just inside the .irp block. The user may redefine a symbol outside the block. The user cannot nest multiple .irp directives. When using the symbol in an expression inside the block, the expression must be inside parentheses.
For example, assembling:

```
.irp param I1 I2 I3
LSU1.LDIL \param, 0x0000
.endr
```

is equivalent to assembling:

```
LSU1.LDIL I1, 0x0000
LSU1.LDIL I2, 0x0000
LSU1.LDIL I3, 0x0000
```

6.2.6.2. .rept

Syntax:

```
.rept <count>
```

Description:

Repeat the sequence of lines between the .rept directive and the next .endr directive <count> times.
For example, assembling:

```
.rept 3
IAU.ADD.U32S I1, I1, I1
.endr
```

is equivalent to assembling:

```
IAU.ADD.U32S I1, I1, I1
IAU.ADD.U32S I1, I1, I1
IAU.ADD.U32S I1, I1, I1
```

6.2.6.3. .repeat

Syntax:

```
.repeat count
```

Description:

Same as `.rept`.

6.2.6.4. .endr

Syntax:

```
.endr
```

Description:

`.endr` marks the end of `.irp` or `.rept` (.repeat) block. No other preprocessor directive is allowed between `.irp` (or `.rept` (.repeat)) and `.endr`.

6.2.7. Control Directives

6.2.7.1. .abort

Syntax:

```
.abort
```

Description:

This directive stops the assembly of the current file immediately. It may be used for interrupting the assembly of the current file when the code takes an undesired branch.

6.2.7.2. .line

Syntax:

```
.line <lineNumber>
```

Description:

Change the logical line number. `<lineNumber>` must be an absolute expression. The next line has that logical line number. It is a synonym for `.ln`.

6.2.7.3. .ln

Syntax:

```
.ln <lineNumber>
```

Description:

Change the logical line number. `<lineNumber>` must be an absolute expression. The next line has that logical line number. It is a synonym for `.line`.

6.2.7.4. .list

Syntax:

```
.list
```

Description:

Control (in conjunction with the `nolist` directive) whether or not assembly information is displayed during the assembling process. This information contains assembled lines, together with their offsets (e.g. `00000008: bru.swih 0x1f`).
These two directives maintain an internal counter (which is zero initially). .list increments the counter, and `nolist` decrements it. The display of the assembled line is ON if the counter is greater than zero.
By default, listings are disabled (counter = 0). The user can enable the listings using:

- the command line of `moviAsm` (with the `-list` command line switch). In this case, the initial value of the listing counter is one
- the `.list` directive inside the `.asm` code

6.2.7.5. .nolist

Syntax:

```
.nolist
```

Description:

See `.list` directive.

6.2.7.6. .print

Syntax:

```
.print {<string> | <symbol >}
```

Description:

The assembler prints string on the standard output during assembly. The string must be in double quotes.
If the string is empty, only a newline is printed.
If a symbol is specified, the value of that symbol is displayed. If the symbol is not defined at that point, its value is UNDEFINED.

6.2.7.7. .err

Syntax:

```
.err <errorMessage>
```

Description:

If the assembler assembles a .err directive, it prints an error message and does not generate an object file. This can be used to signal error in conditional code. The user must specify a message which is displayed as an error.

6.2.8. Editor Directives

These directives are ignored by the preprocessor. Their only purpose is to send some information to the editor.

6.2.8.1. .region

Syntax:

```
.region <regionName>
```

Description:

Marks the region which may be collapsed in the editor. The directive is ignored by the preprocessor. The region ends with an `.endregion` directive.

6.2.8.2. .endregion

Syntax:

```
.endregion
```

Description:

Mark the end of a .region block. This directive is ignored by the preprocessor. Its only purpose is for the editor.

6.2.9. Debug Directives

6.2.9.1. .file

Syntax:

```
.file [<fileno>,] "<filename>"
```

Description:

fileno is used to assign filenames to the .debug_line file name table.

If no fileno is given, the current file is assumed. Implicit value for fileno is 1.

The fileno operand must be a positive integer, to be used as an index entry in the .debug_line file name table.

For more info check the DWARF 2 standard.

6.2.10. Call Frame Information Directives

6.2.10.1. .cfi_startproc

Syntax:

```
.cfi_startproc
```

Description:

This directive marks the beginning of a function. This will typically be emitted by the compiler before the label marking the entry point to a function.

6.2.10.2. .cfi_endproc

Syntax:

```
.cfi_endproc
```

Description:

This directive marks the end of a function. This will be emitted after the last instruction of a function. All other CFI directives for a function will be somewhere between the `.cfi_startproc` and `.cfi_endproc` directives for that function.

6.2.10.3. .cfi_offset

Syntax:

```
.cfi_offset <register>, <offset>
```

Description:

The old value contained by `<register>` has been stored at `<offset>` from the current stack pointer. This will only be emitted in the prologue of a function. Currently only the frame pointer and the link register are mentioned by this directive. Registers are referenced by name.

6.2.10.4. .cfi_def_cfa_offset

Syntax:

```
.cfi_def_cfa_offset <offset>
```

Description:

The stack pointer has been adjusted by offset.

6.2.11. Other Directives

6.2.11.1. .version

Syntax:

```
.version <versionNumber>
```

Description:

This directive provides backward compatibility with the previous versions of the assembler. The string the user provides after the directive is considered version string. The `<versionNumber>` must be in the following format:

```
xx.xx.xx.x[x…]
```

where x is a digit.
The version string is the base for providing backward compatibility. Thus, if the user does not specify a `.version` preprocessor directive before any assembler line, an error is signalized and the assembler exits.
If the user specifies a valid `<versionNumber>`, the assembler checks the code and, if a compatibility problem is detected (a mnemonic or an operation latency was modified since the code version), it displays a warning each time the operation is met. In this case, the user should check each compatibility warning, modify possible errors and update the version string at the beginning of the code.

6.2.11.2. .csr

Syntax:

```
.csr
```

Description:

This directive, Code Size Reset, sets the `__CODE_SIZE__` to the current value of the Instruction Pointer, allowing the user to count the number of bytes occupied by a section of code. The specified zone starts with a `.csr` directive and ends with a `.print __CODE_SIZE__` directive.

6.2.11.3. .nowarn

Syntax:

```
.nowarn <number>
.nowarn low
.nowarn
```

Description:

This directive, used with an integer index, instructs `moviAsm` to print all the warning messages, except the message corresponding to that index. The warnings include the index number in their format:
`moviAsm: WARNING xx: ..`
The same directive also accepts the attribute `low`. This will stop printing warnings that present a low risk if not taken into consideration (e.g. Unaligned target might generate a stall). See table below for a list of all the warning messages.

The default behavior, with no parameter involved, is to stop printing all the warnings.

The table below doesn't represent the full list of warnings, but only some examples. Other warnings may be generated due to specific chip architecture being assembled for. Each of them are self-explanatory, having proper description of the warning in the displayed command line message.

Table 10. Warning messages, message id and priority

Message	Id	Priority
BRU.SWIH present in last 5 instructions	0	high
Invalid total-bytes specifier	1	low
Invalid debug keyword	2	low
LSU0.STVX/LDXV must be followed by a NOOP on LSU0 slot	3	high
LSU1.STVX/LDXV must be followed by a NOOP on LSU1 slot	4	high
PEU.PVEN8C will set 'invalid opcode' flag	5	high
Section attribute already specified	6	high
Unknown preprocessor directive used	7	critical
Too many PEU instructions in block %u.	8	low
Target instruction might generate a stall (> 128 bits)	9	low
Unaligned target instruction might generate a stall (> 128 bits)	10	low
Unaligned average target instruction block might generate a stall (>128 bits)	11	low
Average instruction block might generate a stall (>128 bits)	12	low
Printed <> extra delay slots before the end of the section	13	low
Instruction <> is conflicting	14	high
Unsupported .linkonce type. Using default value ('discard')	15	low
The align specifier inserted a NOP.	16	low
Specified location (0x..) for <> section is ignored in elf mode.	17	high
Cannot determine port clashes at line	18	low
.nowarn directive already in use at line	19	high
.nowarnend has no effect at line	20	high
Insufficient information to determine target label	21	low
Instruction <> is not supported on Myriad 2.1	22	high
SLICE_LOCAL is deprecated; please use SHAVE_LOCAL instead	23	low
Unknown .nowarn attribute	24	low
Out of range warning message index	25	low
Invalid .cfi_def_cfa_offset without .cfi_startproc	26	low
Invalid .cfi_endproc without .cfi_startproc	27	low
Invalid .cfi_offset without .cfi_startproc	28	low
Missing .cfi_endproc	29	low
Deprecated directive %s	30	low
Incomplete octal sequence	31	low
.END found in included file %s at line %d	32	low
Invalid line number in .loc directive	33	low
The analyzer is disabled for MA2480 and 2485!	34	low
Invalid instruction name %s	35	low

Message	Id	Priority
Missing end quote	36	low
Deprecated use of semicolon for comments	37	low
Deprecated command line switch %s	39	low

6.2.11.4. .nowarnend

Syntax:

```
.nowarnend
```

Description:

This directive instructs `moviAsm` to start printing warnings again.

6.3. Predefined Preprocessor Symbols

6.3.1. __CODE_VERSION__

This symbol contains the code version string.

6.3.2. __VERSION__

This symbol contains the assembler version string.

6.3.3. __DATE_TIME__

This symbol contains the current date and time.

6.3.4. __ERRORS__

This symbol contains the total number of errors.

6.3.5. __WARNINGS__

This symbol contains the total number of warnings.

6.3.6. __FILE_NAME__

This symbol contains the name of the current file.

6.3.7. __CODE_SIZE__

This symbol contains the code size in bytes. It can be used for statistics purposes using:

- `.print` preprocessor directive to display its value and
- `.csr` preprocessor directive to reset its value (set it to the current instruction pointer)

6.4. Preprocessor Summary

Category	Directive			
Conditional Directives	.if	.ifc	.ifdef	.ifeq
	.ifeqs	.ifge	.ifgt	.ifle
	.iflt	.ifnc	.ifndef	.ifne
	.ifnes	.ifnotdef	.else	.elseif
	.endif			
Object Directives	.align	.lalign	.salign	
	.bss	.code	.data	.section
	.end	.endb	.noload	.128
	.64			
Section Directives	.ascii	.asciiz	.byte	.single
	.equ	.fill	.float	.float16
	.float32	.word	.hword	.incbin
	.include	.int	.label	.string
	.long	.short		
Preprocessor Symbols Directives	.set	.unset	.unsetall	
Macros Directives	.endm	.exitm	.macro	.purgem
Repeat Directives	.endr	.irp	.repeat	.rept
Control Directives	.abort	.err	.list	.line
	.ln	.nolist	.print	
Editor Directives	.endregion	.region		
Debug Directives	.file			
Other Directives	.csr	.version		
Predefined Symbols	__CODE_VERSION__	__ERRORS__	__FILE_NAME__	__DATE_TIME__
	__VERSION__	__WARNINGS__	__CODE_SIZE__	

7. Profiler

If the assembler is run with the switch `–profile`, an output ASCII file with the extension `.prf` is created, containing profiling information.
The `.prf` file contains the following information:

- Number of `IRF`, `SRF` and `VRF` registers per instruction
- Opcode size per instruction
- Units in use per instruction (number and list of units)
- 5 cycle count instruction size (average size for 5 instructions)
- The minimum and maximum register number used for `IRF`, `SRF` and `VRF`
- The minimum, maximum and average values for:
    - Instruction size
    - 5 cycle count instruction size
    - Number of `IRF` registers per instruction
    - Number of `SRF` registers per instruction
    - Number of `VRF` registers per instruction
- Register file accesses (`Read`, `Write`, `Read`/`Write`) for each register
- Minimum number of registers required by the current application