# ASM32 Reference Manual

WOLFGANG ESCHENFELDER

October 20, 2025

# Contents

# 1

# The ASM32 Assembler

This document describes the use of the ASM32 Assembler for the VCPU-32. The ASM32 Assembly Language represents machine language instructions symbolically, and permits declaration of addresses symbolically as well. The Assembler's function is to translate an assembly language program, stored in a source file, into machine language.

As the concept for the VCPU-32 is heavily influenced by Hewlett Packards PA-RISC architecture, the following reference manual follows the structure of the PA RISC Assmebler manual.

For details about the VCPU-32 architecture please refer to

VCPU-32 System Architecture and instructions Set Reference

## 1.1 Assembler Features

The Assembler provides a number of features to make assembly language programming convenient. These features include:

- **Mnemonic Instructions:** Each machine instruction is represented by a mnemonic operation code, which is easier to remember than the binary machine language operation code. The operation code, together with operands, directs the Assembler to output a binary machine instruction to the object file.

- **Symbolic Addresses:** You can select a symbol to refer to the address of a location in virtual memory. The address is often referred to as the value of the symbol, which should not be confused with the value of the memory locations at that address.

- **Symbolic Constants:** A symbol can also be selected to stand for an integer constant.

- **Expressions:** Arithmetic expressions can be formed from symbolic constants, integer constants, and arithmetic operators. Expressions involving only symbolic and integer constants, defined in the current module, are called Symbolic Constants. They can be used wherever an integer constant can be used.

- **Storage Allocation:** In addition to encoding machine language instructions symbolically, storage may be initialized to constant values or simply reserved. Symbolic addresses and labels can be associated with these memory locations.

## 1.2   Program Structure

An assembly language program is a sequence of statements. Each statement contains up to four fields:

- Label

- Opcode or directive

- Operands

- Comments

The operands field cannot appear without an opcode field.
There are three classes of statements:

- **Instructions:** represent a single machine instruction in symbolic form.

- **Directives:** communicate information about the program to the Assembler or cause the Assembler to initialize or reserve one or more words of storage for data.

- **Comment** is starting with a ;. All text after the comment until end of line is ignored by the assembler.

The **label** field is used to associate a symbolic address with an instruction or data location, or to define a symbolic constant using the .EQU, .REG, .BYTE, .HALF, etc. directives. If a label appears on a line by itself, or with a comment only, the label is associated with the next address within the same subspace and location counter.

## 1.3   Symbols and Constants

Both addresses and constants can be represented symbolically. Labels represent a symbolic address except when the label is on an .EQU or .REG directive. If the label is on an .EQU or .REG directive, the label represents a symbolic constant.

The Label needs to end with a ":".

Symbols are composed of

- uppercase and lowercase letters (A-Z and a-z)

- underline

- decimal digits (0-9)

  A symbol needs to begin with a letter.

The Assembler considers uppercase and lowercase letters in symbols not distinct. The labels and mnemonics for operation codes, directives, and pseudo-operations can be written in either case.

The length of a symbol name is restricted to 32. The name of a symbol needs to be unique within a scope, This means it can not occur twice or more within an adressable range.

Integer constants can be written in decimal or hexadecimal notation, as in the C language. In addition it is possible to structure hexadecimal constants by underscores.

Example: 0x0123_4567_89ab.

## 1.4   Register and Register Mnemonics

The VCPU-32 features 3 types of registers:

- 16 general register R0 - R15

- 8 segment register S0 - S7

- 32 control register C0 - C31

  The PSW register is a maschine internal register which is not
  directly used in the ASM32 assembler.

  Data is loaded from memory into general registers and stored into memory from general registers. Arithmetic and logical operations are performed on the contents of the general registers.
  Some additional predefined register mnemonics are provided based on the standard procedure-calling convention.

### 1.4.1   General Register Mnemonics

| Register | Mnemonic | Description |
|----------|----------|---------------|
| R9       | ARG3     |               |
| R10      | ARG2     |               |
| R11      | ARG1     |               |
| R12      | ARG0     |               |
| R13      | DP       | Global Data   |
| R14      | RL       | Return Link   |
| R15      | SP       | Stack Pointer |

### 1.4.2   Segment Register Mnemonics

| Register | Mnemonic | Description     |
|----------|----------|-----------------|
| S5       | TS       | Task Segment    |
| S6       | JS       | Job Segment     |
| S7       | SS       | System Segment  |

## 1.5  Expressions

Arithmetic expressions are often valuable in writing assembly code. The Assembler allows expressions involving integer constants and symbolic constants. These terms can be combined with the standard arithmetic operators.

| Operator | Operation |
|----------|-----------|
| + | Integer addition |
| - | Integer subtraction |
| * | Integer multiplication |
| / | Integer division (result is truncated) |
| % | Modulo |
| L% | Left 22bit based on 32 bit value |
| R% | Right 10 bit based on 32 bit value |

## 1.6  Parenthesized Subexpressions

The constant term of an expression may contain parenthesized subexpressions that alter the order of evaluation from the precedence normally associated with arithmetic operators.

For example

        LDI R5, data5 + ( data7 - 4 ) * data0

## 1.7  Program structure

The structure of an assembler program for the ASM32 assembler has two flavours:

- **Standalone mode** The program consists of a GLOBAL area where all data and code sits within. There are no modules or functions allowed.

- **module mode** The program consists of 1-n MODULES with functions. In this case no GLOBAL is allowed.

## 1.8   Scope

In Standalone Mode there is only 1 scope level (program scope) allowed.

In Module Mode there are up to 3 scope levels

- Program Scope contains all information of a single source code file.

- Module Scope is the next level unit within a source file (Program Scope).

- Function Scope is the next level beyond the Module Scope.
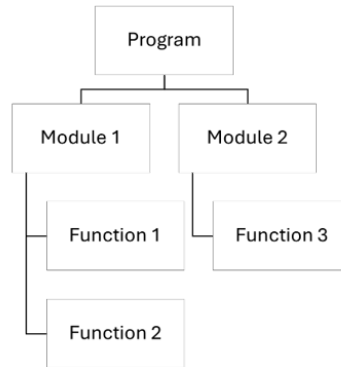


The program scope is created automatically based on the source file. It is also called **Assembly Unit.** Per assembly run one source file is processed.

The directives .MODULE and .FUNCTION determine the scope level within a program.

The symbol table is organized according to the scope level. If a new symbol is defined it must be unique within the current scope. Reference to the symbol walks along the scope tree upwards. E.g. is a symbol is referenced on function scope, it is searched on function scope level. If it is not found it is searched on module scope level. If it is not found on module level, it is searched on program scope level.

## 1.9   Assembler Directives

Assembler directives allow to take special programming actions during the assembly process. The directives begin with a period (.) to distinguish them from machine instruction opcodes.

The format of a directive is:

**Label: .DIRECTIVE [parameter]**

The bold directives are implemented in Version 1 (A.00.1.xx).

| | |
|---|---|
| **.ALIGN** | Aligns data address to the next address divisible by the number |
| **.BUFFER** | Reserves a buffer with size and initializes it to the given value |
| **.BYTE** | Reserves 8 bits (a byte) of storage and initializes it to the given value |
| **.CODE** | Defines the code address for the program loader or defines the following instructions as code. |
| **.DATA** | Defines the data address or defines the following directives as data. |
| **.DOUBLE** | Reserves 64 bits (a byte) of storage and initializes it to the given value |
| **.END** | Defines End of .PROGRAM, .GLOBAL, .MODULE, .FUNCTION |
| **.EQU** | Assigns an expression to an identifier |
| .EXPORT | Exports the name of a function |
| .FUNCTION | Start of Function scope |
| .FUNCTION MAIN | Defines the entry point of a program |
| .FUNCTION INIT | Defines the initialization routine entry point of a module |
| **.GLOBAL** | Defines the global area |
| **.HALF** | Reserves 16 bits (a byte) of storage and initializes it to the given value |
| .IMPORT | Imports all exported functions from a given module |
| .MODULE | Start of Module scope |
| **.REG** | Assigns a user defined name to a register |
| **.STRING** | Reserves a buffer initialized with a string |
| **.WORD** | Reserves 32 bits (a byte) of storage and initializes it to the given value |

### 1.9.1 ALIGN

**.ALIGN <parameter>**

- **Label:** n.a.

- **Parameter:** any number 2, 3, 4, etc. or 1k = 1024

- **Function:** the data address is aligned to the next address which is divisible by the number without a remainder. For the alignment 0x00 is written to the output. If the number is not a $2^n$, a warning is printed.

### 1.9.2 BUFFER

**<name:> .BUFFER <parameter>**

- **Label:** <name>:

- **Parameter:** size=<size in bytes,init=0xyy>

- **Function:** Reserves a buffer with size and initializes it to the given value. If no init value is given, the buffer is initialized to 0x00. The buffer is aligned to word boundary. For the alignment 0x00 is written to the output.

### 1.9.3 BYTE

**<name:> .BYTE <parameter>**

- **Label:** <name>:

- **Parameter:** Number in decimal or 0xyy. Range from -128 to 127

- **Function:** Reserves 8 bits (a byte) of storage and initializes it to the given number. If no value is given, the byte is initialized to 0x00.

### 1.9.4 CODE

**.CODE**

- **Label:** Name of .text section - .text.name

- **Parameter:** addr=0xyyyyyyyy,align=0xzzzzzzzz,entry.

- **addr**: starts a new segment and a section and defines the code 32-bit address for the program loader.

- **align:** defines the alignment and is checked with the parameter addr.

- **entry**: defines this CODE section as entry.

- **Instructions:** All following instructions (until next .CODE) belong to this segment/section The Instructions are located on the address given by addr=.

- **Only in standalone mode**: This directive is only allowed in the GLOBAL Section.

### 1.9.5   DATA

**.DATA**

- **Label:** Name of .data section - .data.name

- **Parameter:** addr=0xyyyyyyyy,align=0xzzzzzzzz,base=Rxx

- **addr**: starts a new segment and a section and defines the code 32-bit address for the data area in the GLOBAL section.

- **align**: defines the alignment and is checked with the parameter addr.

- **base**: defines the base register to be used in offset calculation. The programmer needs to ensure the base register is loaded with the right address before using a variable in this data section.

- **Data**: All following data definitions (until next .DATA) belong to this segment/section and addresses begin with 0.

- **Only in standalone mode**: This directive is only allowed in the GLOBAL Section.

### 1.9.6   DOUBLE

**<name:> .DOUBLE <parameter>**

- **Label:** <name>:

- **Parameter:** Number in decimal or 0xyy. Range from -9,22337E+18 to 9,22337E+18

- **Function:** Reserves 64 bits (a double word) of storage and initializes it to the given number. If no value is given, the double word is initialized to 0x00. The double word is aligned to double word boundary. For the alignment 0x00 is written to the output.

### 1.9.7   END

**<.END <parameter>**

- **Label:** n.a.

- **Parameter:** .GLOBAL, .MODULE, .FUNCTION

- **Function:** Defines End of .GLOBAL, .MODULE, .FUNCTION and of Program.

### 1.9.8   EQU

**<name> .EQU <parameter>**

- **Label:** <name>:

- **Parameter:** name for value or expression

- **Function:** Assigns an expression to an identifier. Nested EQU are allowed. e.g. A1: .EQU 5, A2: .EQU A1.

### 1.9.9 EXPORT

**.EXPORT \<parameter\>**

- **Label:** n.a.

- **Parameter:** \<name of FUNCTION\>

- **Function:** Makes the FUNCTION accessible by other MODULES.

- **Only in module mode**

### 1.9.10 FUNCTION

**\<name:\> .FUNCTION**

- **Label:** \<name\>:

- **Parameter:** n.a.

- **Function:** Defines a function. This directive is only allowed in the MODULE Section. AFUNCTION con only be called if the FUNCTION is defined before in the same modue or if the FUNCTION is imported from another MODULE.

- **Only in module mode**

### 1.9.11 FUNCTION MAIN

**\<MAIN:\> .FUNCTION**

- **Label:** MAIN:

- **Parameter:** n.a.

- **Function:** Defines the main function of the program. One function with label MAIN is required on program level. Data address for data inside the function is maintained on stack (SP).

- **Only in module mode**

### 1.9.12 FUNCTION INIT

**\<INIT:\> .FUNCTION**

- **Label:** INIT:

- **Parameter:** n.a.

- **Function:** Defines the init function of the module. One INIT function is required per module. The INIT function should be defined as the last function of the MODULE. At loading time of the exectuable program all INIT functions of all modules are exectuted before the MAIN function is called.

- **Only in module mode**

### 1.9.13   GLOBAL

**.GLOBAL**

- **Label:** n.a.

- **Parameter:** n.a.

- **Function:** Defines the global area. In this area only the following directives are allowed: BUFFER, BYTE, DOUBLE, HALF, WORD, CODE, DATA. The global can not combined in the same source file with modules and functions.

### 1.9.14   HALF

**<name:> .HALF <parameter>**

- **Label:** <name>:

- **Parameter:** Number in decimal or 0xyy. Range from -32768 to 32767

- **Function:** Reserves 16 bits (a byte) of storage and initializes it to the given number. If no value is given, the half word is initialized to 0x00. The buffer is aligned to half word boundary. For the alignment 0x00 is written to the output.

### 1.9.15   IMPORT

**.IMPORT <parameter>**

- **Label:** n.a.

- **Parameter:** <name of MODULE>

- **Function:** Makes all exported FUNCTIONS of the MODULE accessible. An imported FUNCTION can be called by <module name>.<function name>.

- **Only in module mode**

### 1.9.16   MODULE

**<name:> .MODULE**

- **Parameter:** n.a.

- **Function:** Defines a module. The instruction address is always set to 0x0000 (relative addressing). All instructions inside the module need to be in functions. Data address is in DP register.

- **Only in module mode**

## 1.9.17   REG

**<name> .REG <parameter>**

- **Label:** <name>:

- **Parameter:** register

- **Function:** Assigns a user defined name to a register. No nested REG are allowed.

## 1.9.18   STRING

**<name:> .STRING <parameter>**

- **Label:** <name>:

- **Parameter:** string data

- **Function:** string is terminated with 0x00. Length of string may not exceed line. The string is aligned to word boundary. For the alignment 0x00 is written to the output.

## 1.9.19   WORD

**<name:> .WORD <parameter>**

- **Label:** <name>:

- **Parameter:** Number in decimal or 0xyy. Range from -2147483648 to 2147483647

- **Function:** Reserves 32 bits (a word) of storage and initializes it to the given number. If no value is given, the word is initialized to 0x00.

## 1.10   Handling of large offsets

Based on the nature of the instructions, not all possible offsets can be handled directly in one instruction. E.g. the max offset in an arithmetic instruction is limited to 12 bit which gives an offset range of -2048 to 2047. For locations defined within a data section the base for the section is loaded into a basereg for this section. If a location is defined for example as a buffer with the len 5000, the location after the buffer can not be addressed by an offset. In this case the assembler generates an additional instruction.

Example:

```
ADD                R5, W1
    if the offset of W1 is > 2047
    the assembler generates an instruction
ADDIL R8,L%<offset>
    where R8 is the basereg for the data section containing W1
    and modifies the original ADD to
ADD      R5,R%<offset>(R1)
```

   **This works only for data offsets.**

Branch offsets must be handled by the programmer himself to ensure relocate-ability.

## 1.11   ASM32 Output file structure

The output of the ASM32 assembler is based on the standard ELF format. It is adapted to the requirements of the ASM32.

We use the **ELFIO library** from Serge Lamikhov-Center. The library can be found here:

$$https://github.com/serge1/ELFIO$$

ELFIO is a lightweight, header-only C++ library for reading and generating ELF (Executable and Linkable Format) binary files. It is completely standalone, requiring no external dependencies, and integrates seamlessly into any C++ project. Built to ISO C++ standards, ELFIO ensures compatibility across a wide range of architectures and compilers.

## 1.12   Example Program Standalone Mode

```
 1 ; ======================================
 2 ; Example program ASM32 Bootstrap Mode
 3 ; ======================================
 4 ;   Global area
 5 ; ======================================
 6
 7                     .GLOBAL
 8 CODE1:              .CODE addr=0x0000_0020,align=0x0000_0010,entry
 9                     ADCH    R1,R2(R3)
10
11 DATA1:              .DATA addr=0x0000_0030,align=0x0000_0010
12
13 BUFF:               .BUFFER size=5,init=0x99
14
15 StartReg:           .REG    R1
16 Foo_Bar:            .EQU    47
17
18 S_1:                .STRING "HALLO"
19 Word_01:            .WORD 0x0123_4567
20
21 DATA2:              .DATA addr=0x0003_0000,align=0x0001_0000
22
23 B1:                 .BYTE 0x01
24                     .ALIGN 16
25 H1:                 .HALF 0x0123
26 D1:                 .DOUBLE 0x0123_4567_89ab_cdef
27 AB_C1:
28                     ADCH    R1,R2(R3)
29                     ADD     R1,Foo_Bar
30                     ADC.O   StartReg,R2
31 X:                  ADD     R1,R2
32
33                     ADC.L   R2,-47
34                     B       AB_C1
35                     B       X
36
37 CODE2:              .CODE addr=0x0001_0000,align=0x0001_0000,
38
39                     ADCH    R3,R4(R5)
40                     ADD     R1,R2
41                     ADC.L   R2,-47
42                     ADD     R1,R2
43                     .END
           E:   Segment CODE1 overlaps with segment DATA1
```

```
+--------------------------------------------------------------------------------+
|                                 ELF FILE                                       |
+--------------------------------------------------------------------------------+
ELF Header

  Class:     ELF32
  Encoding:  Big endian
  ELFVersion: Current
  OS/ABI:    Linux
  ABI Version:0
  Type:      Executable file
  Machine:   Advanced Micro Devices X86-64 processor
  Version:   Current
  Entry:     0x20
  Flags:     0x0

Section Headers:
[ Nr ] Type             Addr       Size       ES   Flg Lk   Inf   Al    Name
[   0] NULL             0x00000000 0x00000000 0x00     0x00 0x000 0x00
[   1] STRTAB           0x00000000 0x00000041 0x00     0x00 0x000 0x01  .shstrtab
[   2] PROGBITS         0x00000030 0x00000014 0x00 WA  0x00 0x000 0x04  .data.DATA1
[   3] PROGBITS         0x00030000 0x00000011 0x00 WA  0x00 0x000 0x04  .data.DATA2
[   4] PROGBITS         0x00000020 0x00000020 0x00 AX  0x00 0x000 0x10  .text.CODE1
[   5] PROGBITS         0x00010000 0x00000010 0x00 AX  0x00 0x000 0x10  .text.CODE2
[   6] NOTE             0x00000000 0x00000038 0x00     0x00 0x000 0x01  .note
Key to Flags: W (write), A (alloc), X (execute),
              M (merge), S (strings), I (info),
              L (link order), O (extra OS processing required),
              G (group), T (TLS), C (compressed), E (exclude)
Program Headers:
[ Nr ] Type             VirtAddr   PhysAddr   FileSize   Mem.Size   Flags   Align
[   0] LOAD             0x00000030 0x00000030 0x00000014 0x00000014 RW      0x00000010
[   1] LOAD             0x00030000 0x00030000 0x00000011 0x00000011 RW      0x00010000
[   2] LOAD             0x00000020 0x00000020 0x00000020 0x00000020 R E     0x00000010
[   3] LOAD             0x00010000 0x00010000 0x00000010 0x00000010 R E     0x00010000

[0] .data.DATA1
[1] .data.DATA2
[2] .text.CODE1
[3] .text.CODE2

Note section (.note)
    No Name          Data size  Description
  [ 0] Created by ASM32 0x00000000 0x00000001
  [ 1] A.00.1.18     0x00000000 0x00000001

Section Data:
.shstrtab
[0x00000000] 0x00 0x2e 0x73 0x68 0x73 0x74 0x72 0x74 0x61 0x62 0x00 0x2e 0x64 0x61 0x74 0x61
[0x00000010] 0x2e 0x44 0x41 0x54 0x41 0x31 0x00 0x2e 0x64 0x61 0x74 0x61 0x2e 0x44 0x41 0x54
[0x00000020] 0x41 0x32 0x00 0x2e 0x74 0x65 0x78 0x74 0x2e 0x43 0x4f 0x44 0x45 0x31 0x00 0x2e
[0x00000030] 0x74 0x65 0x78 0x74 0x2e 0x43 0x4f 0x44 0x45 0x32 0x00 0x2e 0x6e 0x6f 0x74 0x65
.data.DATA1
[0x00000000] 0x99 0x99 0x99 0x99 0x99 0x00 0x00 0x00 0x48 0x41 0x4c 0x4c 0x4f 0x00 0x00 0x41
[0x00000010] 0x01 0x23 0x45 0x67
.data.DATA2
[0x00000000] 0x01 0x01 0x23 0x00 0x23 0x45 0x67 0x4f 0x00 0x01 0x23 0x45 0x67 0x89 0xab 0xcd
[0x00000010] 0xef
.text.CODE1
[0x00000000] 0x44 0x49 0x00 0x23 0x44 0x49 0x00 0x23 0x40 0x40 0x00 0x2f 0x44 0x54 0x00 0x21
[0x00000010] 0x40 0x44 0x00 0x21 0x44 0xa3 0xff 0xd1 0x80 0x3f 0xff 0xfb 0x80 0x3f 0xff 0xfd
.text.CODE2
[0x00000000] 0x44 0xc9 0x00 0x45 0x40 0x44 0x00 0x21 0x44 0xa3 0xff 0xd1 0x40 0x44 0x00 0x21
.note
[0x00000000] 0x00 0x00 0x00 0x11 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x43 0x72 0x65 0x61
[0x00000010] 0x74 0x65 0x64 0x20 0x62 0x79 0x20 0x41 0x53 0x4d 0x33 0x32 0x00 0x00 0x00 0x00
[0x00000020] 0x00 0x00 0x00 0x0a 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x41 0x2e 0x30 0x30
[0x00000030] 0x2e 0x31 0x2e 0x31 0x38 0x00 0x00 0x00

Segment Data:
Segment # 0
[0x00000000] 0x99 0x99 0x99 0x99 0x99 0x00 0x00 0x00 0x48 0x41 0x4c 0x4c 0x4f 0x00 0x00 0x41
[0x00000010] 0x01 0x23 0x45 0x67
Segment # 1
[0x00000000] 0x01 0x01 0x23 0x00 0x23 0x45 0x67 0x4f 0x00 0x01 0x23 0x45 0x67 0x89 0xab 0xcd
[0x00000010] 0xef
Segment # 2
[0x00000000] 0x44 0x49 0x00 0x23 0x44 0x49 0x00 0x23 0x40 0x40 0x00 0x2f 0x44 0x54 0x00 0x21
[0x00000010] 0x40 0x44 0x00 0x21 0x44 0xa3 0xff 0xd1 0x80 0x3f 0xff 0xfb 0x80 0x3f 0xff 0xfd
Segment # 3
[0x00000000] 0x44 0xc9 0x00 0x45 0x40 0x44 0x00 0x21 0x44 0xa3 0xff 0xd1 0x40 0x44 0x00 0x21
```

# 2

# Implementation Status

This temporary chapter documents the actual implementation status of the ASM32 Assembler.

**Version 2 is the standalone program mode.(A.00.2.xx) with proper handling of additional instructions generated by the assembler**

**Implemented in Version 1**

- Standalone program

- all VCPU32 instructions

- Code-generation

- proper handling underscore in names and addresses

- handling of .CODE and .DATA

- filling structure of ELF file

# 3

# Assembler Processing

The assembler processes the source code within the following steps:

- Main program
- Lexer
- Parser
- Codegen

## 3.1   Main Program

The main program establishes the overall environment, opens the source file, reads the source line and passes into the lexer. It builds a tree structure SRCNode global program source line as a base for printing source lines, binary instructions and error messages. After the lexer the parser is called and finally the coding is copied to the ELF text sections. Data are copied during the parser to the corresponding ELF data sections.

Finally the sections are added to the required segments and the ELF output file is created.

The main program contains a couple of DEBUG Switches, which provide additional listings. Currently they are implemented as global variables which can be set to TRUE or FALSE.

- **DBG_TOKEN** Prints the list of tokens generated by the lexer.
- **DBG_PARSER** Prints details of the parsing process.
- **DBG_GENBIN** Prints details of codegen process.
- **DBG_SYMTAB** Prints the symbol table.
- **DBG_AST** Prints the abstract syntax tree.

- **DBG_SEGMENT** Prints the segment table.
- **DBG_SOURCE** Prints a list of address, maschine code, source-line number and source text and corresponding errors:
- **DBG_ELF** Dumps ELF File.

## 3.2  Lexer

The Lexer reads the source line and extracts the tokens and provides the tokenList. This is a linked list with all tokens, sourcline number and column. During processing the lexer converts **hexadecimal values to decimal** values and masks L% and R% values accordingly. The lexer also creates the **SRC** structure and populates with source code.

## 3.3  Parser

Before the Parser is started the structures for the ELF output are pre-pared. The Parser reads the list of tokens and generates the symbol table (SymNode) and the abstract syntax tree (ASTNode). The parser consists of mutliple subprocessing routines for label (ParseLabel), directives (ParseDirectives) and instructions (Parseinstruction).

All tokens are translated to uppercase to ensure a common naming inside symboltable etc.

## 3.4  ParseLabel

This is a simple routine which just moves a valid label token into a field named label. It will be processes with the ParseInstruction or ParseDirective routine.

If the label is called MAIN or INIT and the directive is .FUNCTION, it is checekd whether MAIN is unique across the source and if every module consists of a INIT function.

## 3.5  ParseDirective

The tokens identified as directives (starting with a .) are checked against a table dirCodeTab. If the directive is valid the appropriate processing takes place.

For each directive an entry in the symbol table is created. **MODULE** and **FUNCTON** directives and their corresponding end functions build scope levels withinn the symbol table tree.

**SymbolTable**

- scope type

- scope level -> program=1, module=2, function=3

- scope name

- label

- function -> directive (without .)

- value

- variable type

- linenr

- code address

- data address

- pointer to children nodes

- number of children nodes

**Scope types:**

- SCOPE_PROGRAM

- SCOPE_MODULE

- SCOPE_FUNCTION

- SCOPE_DIRECT

**Variable Type**

- Value

- Memory global

- Memory local

- Label

### 3.5.1  MODULE and FUNCTION processing

If a MODULE directive is detected, scopetype is set to SCOPE_MODULE, dataAdr is set to zero. If .ENDMODULE is detected, scope level is set back to SCOPE_PROGRAM. Analog processing for FUNCTION.

### 3.5.2   EQU and REG processing

For both directives it is checked if a directive with the same name
(label) is already defined on the same scope level. If not, then the
Symbol is inserted in the symboltable.

REG directives can not be nested, e.g. H1 .REG R7 and H2 .REG
H1 is not allowed.

EQU directives can be nested, e.g. H1 .EQU 556 and H2 .EQU H1.
In this case H1 must be defined prior to H2.

### 3.5.3   BYTE, HALF, WORD, DOUBLE, BUFFER, STRING processing

Depending of the scope level (module or function) the variable type is
set to Memory global (module level) or Memory local (function level).
It is checked if a directive with the same name (label) is already ex-
isting on the same scope level. If not, then the Symbol is inserted in
the symboltable. BYTE, HALF, WORD and DOUBLE are aligned
automatically by the assembler, e.g. WORD to word address.

BUFFER and STRING are aligned to WORD address.

If no init= parameter is provided, BYTE, HALF, WORD, DOU-
BLE and BUFFER are initialized to 0x00.

The values of the above variables copied to the ELF data section
at their corresponding address.

### 3.5.4   EXPORT and IMPORT processing

## 3.6 ParseInstruction

The tokens identified as instructions are checked against a table op-
CodeTab. If the opcoode is valid the further processing takes place in
base of the type of the opcode. An entry in the opCodeTab assigns
every opcode to an OpType. This Optype determines smae parsing
procedure for a goup of opcodes which have the same mnemonic struc-
ture.

For every group a dedicated parsing routine is in place. In these
routines every token is analyzed. if the token is an expected token an
entry into the abstract systax tree (AST) is created.

**AST**

- node type

- value in character of the entry

- numeric value of the entry if available

- linenr of the source line

- column number of the token

- scopelevel (program, module, function)

- name of the scope

- adress of the corresponging node of the symbol table for easier search function in the symbol table.

- operand type to define the type of the value (register, memorylocation, label, value)

- address of code

- binary word of the instruction derived from opCodeTab.

- pointer to children nodes

- number of children nodes

## 3.7    ParseExpression

Expressions are recursively parsed. **ParseExpression** calls **ParseTerm** and calculates plus, minus, or, xor operations.

   **ParseTerm** calls **ParseFactor** and calculates multiply, division, modulo, and operations.

   **ParseFactor** processes the numbers, substitutes the EQU or global/local offsets.

## 3.8    Codegen

The codegen routine reads the **AST** sequentially, calculates code address and populates the binary word of the instruction. The result is stored in the **SRC** structure

   The procedure GenBinOption checks the options by groups of instructions with the same options.

   For instructions using a memory location the memory type can be:

- Memory global (in module) offset is calculated as the data offset within the module and register R13.

- Memory local (in function) tbd

## 3.9 Big-Endian Little-Endian

The target architecture VCPU32 is a Big-Endian maschine. the host systems Windows and MAC are Little-Endian systems.

The binary maschine instruktions are already in big-endian format.

All multi-byte metadata fields in the ELF file, in the ELF header, program headers, section headers, symbol tables, relocation entries, dynamic sections, and auxiliary tables — are subject to big-endian encoding as specified in e_ident[EI_DATA]. The raw contents of sections (machine instructions, literal data, strings) are stored as-is and are not affected by ELF's endianness rule.