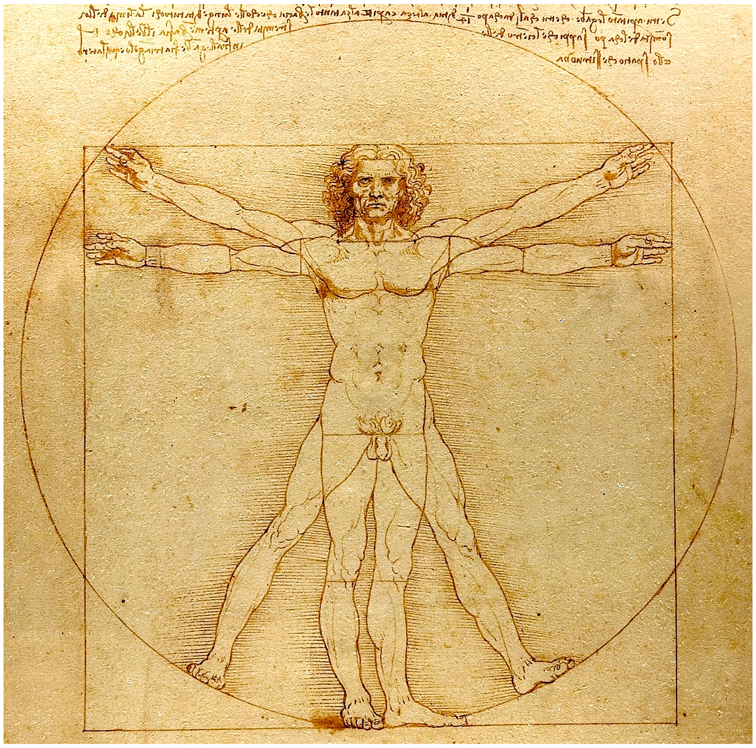
Da Vinci Assembler



RISC-V

Paolo Roberti

Contents

Chapter 1

Overview

The Da Vinci Assembler or DVASM™ is a set of `Java`® modules that implements a multi-architecture assembler framework. Any computer architecture can be supported by adding an architecture specific module.

When developing the Da Vinci framework module, it was decided to test it by also developing a module to support the RISC-V architecture, which is now distributed within the same `JAR` file as the framework module.

This manual is a reference manual that describes how to code RISC-V programs using the Da Vinci assembler.

Chapter 2

Introduction

The RISC-V is an open computer architecture that belongs to the RISC family of architectures. You can find the complete RISC-V ISA specifications at `https://riscv.org/technical/specifications`.

In this manual we are going to give a very short summary of the RISC-V ISA specification.

In RISC-V notation, register length is called XLEN. RISC-V support three different XLENs, XLEN=32,XLEN=64, and XLEN=128. There is also a version for 16 integers registers only. Optionally there can can exist 32 floating point register which can be 32, 64 or 128 bits wide and follow IEEE binary floating point standard.

Memory access is little endian.

ISA specifications is divided in basic ISAs and ISA extensions.

There are four ISA basic specifications which are:

- 32 integer registers with XLEN=32. It is named `RV32I`;
- 16 integer registers with XLEN=32. It is named `RV32E`;
- 32 integer registers with XLEN=64. it is named `RV64I`;
- 32 integer registers with XLEN=128. it is named `RV128I`;

DVASM™ support only the first three. It is important to note that these basic ISAs are really basic and do not include integer multiply and divide, floating point, control registers, supervisor instruction, etc.

There are also several extensions and there are provisions for chip designers to add "private" extensions.

DVASM™ supports all extension that have been "ratified", which means that everybody is committed to these extension without the possibility for additional changes. All ISA extensions that are currently "open", and some that are "frozen" are not supported, but all of them will be supported when they will become "ratified".

ISA extensions supported by DVASM™ are:

- "M" for integer multiplication and division;
- "A" for atomic instructions;
- "F" for 32 bit floating point instructions;
- "D" for 64 bit floating point instructions;
- "Q" for 128 bit floating point instructions;
- "ZICSR" for control register support;
- "ZIFENCEI" for instruction fetch fence;
- "C" for compressed instructions;
- "S" for supervisor instructions.

2.1 Implementation

DVASM™ RISC-V implementation has followed the opcode naming convention used in the RISC-V specification with the only difference that opcodes are case insensitive.

Coding of parameters have, in some cases, been extended or modified to enhance flexibility and consistency.

There are no reserved register names since DVASM™ framework does not provide for such a feature. However a macro is provided for defining symbols that correspond register names used in the specifications.

All macro listed in the specifications have also been implemented. An additional large number of macros have been added, from symbolic definition of registers, to function call linkage support, to algorithm implementation such as in memory sort, several type of linked list, AVL trees, hashing, etc.

Chapter 3

Coding RISC-V Programs

In this chapter we are going to describe where the DVASM™ coding rules of RISC-V machine instructions deviates from the coding rules in the ISA specifications.

3.1 SETENV Directive

As for all architectures supported by DVASM™ , RISC-V architecture is selected inside the code by using the `SETENV` directive.

The SETENV directive has four parameters that, for RISC-V, are coded as follows:

Arch. Name	Must be coded as `"RISCV"`
Arch. Options	It is a string containing the name of the base ISA name, followed by a colon, and by the list of ISA extensions used, separated by commas. An example is `"RV64I:a,c,d,m,n,zicsr,zifencei"`.
ABI	It is a string that contains the name of a valid ABI which must be compatible with the base ISA used, for example `"LP64D"`.
OS Name	It is a string containing the name of the Operating System being used, for example `"linux"`. If you are coding a new OS, just use the new OS name, and keep in mind that the name is used only by macros.

This is an example on how to use the `SETENV` directive and specify the RISC-V architecture:

```
//
//  We use RISC-V 64 bit arch with extensions normally used for Linux
//
        SETENV  "RISCV", "RV64I:a,c,d,m,n,zicsr,zifencei", "LP64D", "linux"
//
```

3.2 Memory Access

RISC-V is a load/store architecture with the addition of an extension for direct memory atomic machine instruction. All memory access in RISC-V iv via a displacement and a base register. Index registers are not used.

DVASM™ coding of memory access machine instructions, differs from the ISA coding specifications. The target register is always the first parameter. The second parameter contains the displacement and its sub-parameter contains the base register. This is true for both load and stores, even though, for store machine instructions, the memory address is the destination and not the source. This convention makes all memory access machine instructions, including atomic memory operation, consistent.

DVASM™ will also check that the memory displacement parameter is an expression, that resolves to either an `ABSOLUTE INTEGER` or a `DISPLACEMENT INTEGER`. Any other value type will be rejected as an error, including `OFFSET INTEGERs`

In this example, a 32 bit word is loaded from memory location pointed by base register `T1` and displacement `16`, into register `T2`. It is than stored back to memory location pointed by base register `T3` and displacement `24` are coded as follows:

```
BASEDEF                        // Include register definitions
LW      T2, 16 [T1]            // Load 32 bit word
SW      T2, 24 [T3]            // Store it back
```

3.3 Register to Register Machine Instructions

In RISC-V most register-to-register operations such as integer and floating point arithmetic, bit-wise integer logical operations, etc., use three register parameters. The first is the destination register, and the following two are the source registers.

When one of the source registers is also the destination register, the coder can specify only two parameters, the destination register, which is also one of the source register, and the other source register. For example to add values in register `T1` and `T2` and store the result in register `T1` the following standard code can be used:

```
BASEDEF                         // Include register definitions
ADD      T1, T1, T2             // Add two integers - standard way
```

or the short way as follows:

```
BASEDEF                         // Include register definitions
ADD      T1, T2                 // Add two integers - short way
```

The same applies to four register format opcodes. For this format the first register is the destination register and the following three registers are the source registers. When the opcode is encoded with three registers only, instead of four, the third source register (i.e. the missing register) is internally set to the destination register.

In this example the `FMADD.D` floating point opcode instructs the CPU to multiply the first two source registers, add the result to the third source register, and store the final result in the destination register, all using 64 bits floating point registers. The standard way to code `FMADD.D` is:

```
BASEDEF                         // Include register definitions
FMADD    F12, F21, F22, F12     // Mult-add doubles - standard way
```

while the short way is:

```
BASEDEF                         // Include register definitions
FMADD    F12, F21, F22          // Mult-add doubles - short way
```

3.4 The Floating Point Extensions

RISC-V defines three floating point extensions: 32 bit, 64 bit and 128 bit wide. All operations follow IEEE floating point standards.

For most floating point operation, RISC-V provides 3 bit in the opcode encoding, called the rounding mode, whose setting control the rounding of the result.

DVASM™ allows the coder to specify rounding mode, by providing a key-word parameter `RM=`. The default value for this parameter is `0b111` which instruct the CPU to use the default rounding mode set in the corresponding control registers. When other setting must be used, the coder can override the default by using `RM=`.

RISC-V macro `BASEDEF` containig mnemonic definition for all valid rounding modes supported.

Here is a simple example:

```
//
        BASEDEF                         // Include definitions
//
        FMULT.D F12, F13, F14           // Multiply using control register
                                        //  setting for rounding
//
        FMULT.D F12, F13, F14, RM=RM_RNE
                                        // Multiply and round to nearest
                                        //  as defined in BASEDEF macro
```

3.5 The C Extension

The RISC-V C extension allows the coder to use machines instructions which are encode into two byte length instead of the standard four bytes. This allows to generate shorter code. The RISC-V specification define all opcodes in the C extension to end with the suffix `".C"`. When the C extensions is specified in the `SETENV` directive, DVASM™ does not allow the coder to target C extension opcodes directly. Instead, it automatically convert standard four byte length opcodes to two byte length opcodes whenever a corresponding two byte length opcode is available. This action can be blocked by using the `C.SUSPEND` directive.

When the C extension is specified in the `SETENV` the conversion of standard opcodes to C extension opcodes is enabled by default. The coder can suspend or resume automatic conversion using the two directives `C.SUSPEND` and `C.RESUME`. When the directive `C.SUSPEND` is used, no automatic conversion takes place from that statement onward, until a `C.RESUME` directive is encountered, in which case, automatic conversion is resumed from that statement onward. Multiple alternating `C.SUSPEND` and `C.RESUME` can be used within the same source code.

In this example the C extension is enable just for a short loop code, to make sure that the whole loop fits in one or two cache lines:

```
//
        SETENV  "RISCV", "RV64I:a,c,d,m,n,zicsr,zifencei", "LP64D", "linux"
//                                      // RISC-V with C extension
//      C.SUSPEND                       // Disable C extension
//
//
//      Insert here code before loop
//
        C.RESUME                        // Enable C extension
//
        WHILE                           // Start loop
//
//          Insert loop code here
//
            WHILEEND                    // End of loop
//
        C.SUSPEND                       // Suspend C extension again
//
//      Insert code following loop here
//
```

3.6 The ALIGN Opcode

The `ALIGN` opcode is a special opcode used to align a machine instruction to the specified alignment values. The `ALIGN` opcode should be used only to align binary code. To align data, a data opcode with zero length and desired alignment should be used instead of the `ALIGN` opcode.

The `ALIGN` opcode generate a sequence of no-operation machine instructions, called a no-operation slide, so that the slide ends at an offset at the specified alignment. The alignment value must be a power of two, the lowest value being 4 and the highest value being 512.

The opcode also support a key-word parameter `BRCOND=`, which is used to generate a unconditional branch to the slide end, if the slide is too long. When the number of no-operation machine instructions in the slide exceeds the number specified in `BRCOND=` the branch is generated. When the value of `BRCOND=` is `0`, which is the default, no branch is generated regardless of the slide length.

In the following example, a small, CPU intensive loop is expected to run on a chip, with level 1 cache lines that are 64 byte long. `ALIGN` opcode is used to align the loop on 64 byte boundary. The `C.RESUME` directive is used to shorten the loop binary, so that it fits in as few cache lines as possible. The directive `C.SUSPEND` is used after the loop to generate 4 byte long machine instruction only, and it is preceded by the `ALIGN` opcode to force 4 byte alignment.

```
//
        SETENV  "RISCV", "RV64I:a,c,d,m,n,zicsr,zifencei", "LP64D", "linux"
//                                      // RISC-V with C extension
//      C.SUSPEND                       // Disable C extension
//
//
//      Insert here code before loop
//
        ALIGN   64, BRCOND=8            // Align loop on
//                                      //  cache line boundary
//                                      // Force a branch if the no-op
//                                      //  slide is longer than 8 no-ops
        C.RESUME                        // Enable C extension
//
        WHILE                           // Start loop
//
//          Insert loop code here
//
            WHILEEND                    // End of loop
//
        ALIGN   4                       // Force code to align
                                        //  on 4 byte boundary
        C.SUSPEND                       // Suspend C extension again
//
//      Insert code following loop here
//
```

3.7 Linkage Support Opcodes

DVASM™ RISC-V module provides three data opcodes, and the key-word parameter `COND=` for some load and store opcodes to assists coders in saving, at entry, and restoring, at exit, only registers that are being modified within the code. These are specialized extensions that are intended to be used within macros that generate entry code, exit code, and map the entry stack. See the `ENTRY` macro on page 55, the `EXIT` macro on page 58, and the `STACK` macro on page 59, on how these extension can be used.

3.7.1 Register Save Areas

Two data opcodes are provided to create save areas for register being updated in the code.

Opcode `REGSAVE` create a word in storage of length `XLEN`. If the target register being targeted is not being updated in the code, no storage area is created.

Opcode `FREGSAVE` does the same for floating point registers, except that the storage length can be 32, 64 or 128 bits, depending on the floating point extension being specified in the `SETENV` directive.

Opcodes `REGSAVE` and `FREGSAVE` should be used, for example, when mapping the register save area within a stack of a function being called using the standard ABI.

3.7.2 Load and Store Conditional Opcodes

The following load and store opcodes have a key-word parameter `COND=`. When set to `0`, the default, DVASM™ always generates code for the opcode. When set to `1`, DVASM™ generates code only if the target register is being updated within the code. The opcodes are:

`LW`	Load a 4 byte word into integer register.
`SW`	Store a 4 byte word from integer register.
`LD`	Load a 8 byte word into integer register.
`SD`	Store a 8 byte word from integer register.
`FLW`	Load a 4 byte word into floating point register.
`FSW`	Store a 4 byte word from floating point register.
`FLD`	Load a 8 byte word into floating point register.
`FSD`	Store a 8 byte word from floating point register.
`FLQ`	Load a 16 byte word into floating point register.
`SLQ`	Store a 16 byte word from floating point register.

The key-word parameter `COND=` is intended to be mainly used when saving registers when entering a function and restoring registers when exiting.

3.7.3 Clearing the Update Flag for all registers

DVASM™ keeps track of registers being updated by having a boolean variable for each register. When more that one function is coded in the same source file, there is a need to clear these variables in between the two functions. This is achieved by using the `CLRREGFLAGS` directive, which has no parameters.

3.8 Checking for Branch to Branch Condition

Sometimes when using control flow macros that implement `IF`, `ELSE`, `ENDIF`, etc., the code generated, under certain conditions, contains conditional branches to unconditional branches. DVASM™ , after generating the binary, inspects the code, and whenever it detects a conditional branch to an unconditional branch, it issue a warning for the conditional branch statement. However, DVASM™ will not try to optimize the code. That is the responsibility of the coder.

Chapter 4

Macros

DVASM™ RISC-V module comes with set of macros. These macros are intended to help coder write shorter and better code.

4.1 The DVASM.getenv Function

As described in the DVASM™ framework manual, the `JavaScript`® `DVASM.getenv` function can be used by macros to get environmental data. The function returns an array of strings of size 5. The first 4 strings contain the same exact values passed to the `SETENV` directive. The fifth string is architecture dependent, and it contains a set of `JavaScript`® assignments, semicolon separated, that assign values to predefined variables. This string is intended to be the target of the `JavaScript`® `eval` function, as in the following example:

```
eval(DVASM.getenv()[4]);
```

The following variable and its associated values are set:

`Variable Name`	`Value`
`abiNoRegs`	Number of general purpose registers, either 16 or 32
`abiNoFRegs`	Number of floating point registers, either 0 or 32
`abiXLen`	Size of general purpose registers in bits, either 32 or 64
`abiFLen`	Size of floating point registers in bits, either 32, 64 or 128

`abiWLen`	Size of general purpose registers in bytes, either 4 or 8
`abiFWLen`	Size of floating point registers in bytes, either 4, 8 or 16
`abiWID`	Suffix to be used for machine instructions based on general purpose register size, either `'W'`, for 32 bits, or `'D'` for 64 bits
`abiFWID`	Suffix to be used for machine instructions based on floating point register size, either `'S'`, for 32 bits, `'D'` for 64 bits, or `'Q'` for 128 bits

4.2 Macro Classification

The macros can be divided in the following categories:

- Macros defined in the RISC-V specifications;
- Symbol definition;
- Control flow;
- Linkage assist;
- External symbols access;
- Buffer handling;
- String handling;
- Heap sort;
- Linked lists;
- AVL trees;
- Hashing;
- Miscellaneous.

All these macros provide enough functionality to allow anybody to write a small OS kernel with ease.

4.3 Macros Defined in the RISC-V Specifications.

RISC-V specifications define several macros, such as `LA`, `LI`, etc., that complement the ISA specifications. DVASM™ implementation of these macros follow the RISC-V specification, and thus, are not documented here. Please refer to the RISC-V specification documents for more information.

4.4 Definition Macros

There is only one definition macro. The name of the macro is `BASEDEF`. It defines mnemonics for integer registers, floating points registers, floating point rounding modes and control registers. `BASEDEF` also invokes the framework definition macro `FRAMEWORKDEF`. The macro expansion of `BASEDEF`, in the listing, will show all the definitions in details.

4.5 Control Flow Macros

A set of macros is provided to allow coder to use structured programming constructs, such as `IF`, `ELSE`, `ENDIF`, etc.. These macros should be used together. Beside helping the coder with structured programming, these macros take advantage of DVASM™ indentation interface so that a program using these macros will be shown with the correct indentation in the output listing.

These are the control flow macros:

- `IF`
- `ANDIF`
- `ELSEIF`
- `ELSE`
- `ENDIF`
- `WHILE`
- `ENDWHILE`

- `DO`
- `ENDDO`
- `BREAK`
- `CONTINUE`

4.5.1 Branch Condition

In some of the control flow macros a condition (i.e. a Boolean expression) is either required or optional. The condition control the code execution. For example, if at execution time, the condition of the `IF` macro is true, the next machine instruction following the `macro` will be executed, otherwise a branch will be taken to the following `ELSE`, `ELSEIF` or `ENDIF` macro whichever comes first after the `IF` macro.

For macros `IF`, `ANDIF`, and `ELSEIF`, a condition is always required and it is the first positional parameter. For macros `WHILE` and `ENDWHILE`, a condition is optional, and it is specified as the key-word parameter `CONDITION=` which is defaulted to a null string.

The format of a condition is as follows:

- The condition parameter is a string;
- A condition contains comparisons of integer or floating point registers only;
- Comparison format is `Register_1 comparison_operator Register_2`;
- Comparison operators are expressed as `'=='`, `'!='`, `'<='`, `'>='`, `'<'` and `'>'`;
- Comparison operators can be qualified. When not qualified they indicated signed integer comparison. When they are qualified by `{U}` they indicate an unsigned integer comparison (only for `'>='`, `'<='`, `'>'` and `'<'`). When they are qualified by `{F}`, `{D}` or `{Q}` they indicate respectively 32 bit, 64 bit or 128 bit floating point comparison;
- More that one comparison can be specified using the Boolean operators `"&&"` for short circuit AND and `"||"` for short circuit OR;
- Nested round parenthesis are supported.

Whenever a condition is specified, two additional key-word parameters, `WReg=` and `Far=`, are available.

Parameter `WReg=` defaults to a null string, but must specify an integer work register when the condition includes comparison of floating point registers. The reason is that RISC-V ISA handle comparisons of floating point registers differently from integer registers. When comparing integer registers comparison and branching takes place in a single machine instruction. Instead when comparing two floating point registers, RISC-V sets an integer work register either to one, when true, or zero, when false. Branching is done by comparing the integer work register to register zero which is hardwired as binary zero.

Parameter `Far=` is a Boolean parameter and is defaulted to `FALSE` or `NO`. It should be set to `TRUE` or `YES` when the largest displacement of the conditional branch is larger than the maximum +/- 4096 bytes. In this case the macros generate two machine instructions: A conditional branch around the following unconditional branch with a 20 bit displacement.

It is a good practice to always enclose the condition string in either in round parenthesis or quotes. For example condition `R10==R11` should be coded as `(R10==R11)` or `!"R10==R11"`. In fact if `R10==R11` alone is used, the parser will interpret the first four characters `R10=` as key-word parameter.

Following are some examples. Test if register `R2` is bigger than register `R3`:

```
( R2 > R3 )
```

Test if register `R2` is bigger than register `R3` and `R4` less equal register `R5`:

```
( R2 > R3 && R4 <= R5 )
```

Test if register `F2` is bigger than register `F3` using 64 bit floating point comparison and if `R4` is less equal register `R5`. Use `R10` as work register to execute the floating point comparison:

```
( F2 {D}> F3 && R4 <= R5 ), WReg=R10
```

Test if either register `R2` is bigger than register `R3` and if `R4` is less equal register `R5` or register `R8` is greater than `R9` using unsigned integer comparison:

```
( ( R2 > R3 && R4 <= R5 ) || R8 {U}> {R9} )
```

Test if register `R2` is bigger than register `R3` and specify that the branch generated is to a out of range label.

```
        ( R2 > R3 ), Far= YES
```

4.5.2 IF and IF Related Macros

The `IF` macro is used to execute the action specified depending if the condition specified is true or false at execution time.

```
>--+---------+----IF---------------------------------------------------->
   |         |
   +--Label--+

>------condition,-----+------------------+--+----------------+---------->
                      |                  |  |                |
                      +--WReg=register,--+  +--Far=Boolean,--+

>---------+--THEN--------------------+---------------------------------><
          |                          |
          +--GOTO,------ID=label-----+
          |                          |
          +--BREAK-----+-------------+
          |            |             |
          |            +-,-ID=label--+
          |                          |
          +--CONTINUE--+-------------+
                       |             |
                       +-,-ID=label--+
```

`Label`	Optional
`Name`	`IF`
`Pos. Parameters`	
`condition`	Branch condition (see page 18).
`Key-word Parameters`	
`WReg`	Condition work register (see page 18).
`Far`	Boolean, to specify far branches (see page 18).
`ID`	Branch target when GOTO, BREAK or CONTINUED is used.

When the `THEN` action is specified the `IF` macro starts a `IF ... ENDIF` sequence. If the condition is true at execution time, statements following the `IF` macro are executed, otherwise a branch is taken to the next `ELSE`, `ELSEIF` or `ENDIF` macro, whichever comes first.

When the `GOTO` action is specified the `IF` macro becomes a direct branch to the label specified with the `ID=` key-word parameter if the condition is true at execution time.

When the `BREAK` action is specified, and the condition is true at execution time, the `IF` macro break out of the inner-most `WHILE` loop, or `DO` block. The key-word parameter `ID=` can be used to specify the label of an outer loop or block instead.

When the `CONTINUE` action is specified, and the condition is true at execution time, the `IF` macro reiterate the inner-most `WHILE` loop or `DO` block. The key-word parameter `ID=` can be used to specify the label of an outer loop or block instead.

The `ENDIF` macro terminate an `IF` clause macro.

```
>--+---------+---ENDIF--------------------------------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`ENDIF`

`Pos. Parameters`

`None`

`Key-word Parameters`

`None`

The `ANDIF` macro is code as follows:

```
>------------+---ANDIF-----condition------------------------------------>
   |         |
   +--Label--+

>------------------------+------------------+--+----------------+------><
                         |                  |  |                |
                         +--WReg=register,--+  +--Far=Boolean,--+
```

`Label`	Optional
`Name`	`ANDIF`
`Pos. Parameters`	
`condition`	Branch condition (see page 18).
`Key-word Parameters`	
`WReg`	Condition work register (see page 18).
`Far`	Boolean, to specify far branches (see page 18).

If the condition at execution time is true, the statements following the macro are executed, otherwise a branch is taken to the next `ELSEIF`, `ELSE` or `ENDIF` macro, whichever comes first. The `ANDIF` macro must reside inside a IF clause, and does not start a new IF clause.

The `ELSEIF` macro is coded as follows:

```
>--+---------+---ELSEIF---+--------------------------------+------------>
   |         |            |                                |
   +--Label--+            +--PreCode=[precode_statements]--+

>----------Condition,--+------------------+--+----------------+--------><
                       |                  |  |                |
                       +--WReg=register,--+  +--Far=Boolean,--+
```

`Label`	Optional
`Name`	`ELSEIF`
`Pos. Parameters`	
`condition`	Branch condition (see page 18).
`Key-word Parameters`	
`PreCode`	List of strings, where each string contains an assembler statement to be executed before testing the condition.
`WReg`	Condition work register (see page 18).
`Far`	Boolean, to specify far branches (see page 18).

The `ELSEIF` macro is the branch target of an `IF`, `ANDIF` or `ELSEIF` macro within the same IF clause, when the condition is false at execution time. The macro has a key-word parameter `PreCode=` which is a string list containing assembly statements. This statements are executed before the condition is tested for branching. If the condition at execution time is true, the statements following the macro are executed, otherwise a branch is taken to the next `ELSE`, `ELSEIF` or `ENDIF` macro, whichever comes first.

The `ELSE` macro is coded as follows:

```
>--+---------+---ELSE--------------------------------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`ELSE`

`Pos. Parameters`

`None`

`Key-word Parameters`

`None`

The `ELSE` macro is the branch target of an `IF`, `ANDIF` or `ELSEIF` macro when the condition is false at execution time. The `ELSE` macro must be the last macro of the same IF clause before the `ENDIF` macro.

This is an example on how to use control flow macros in a IF clause:

```
//
//          BASEDEF                         // Include definitions
//
            IF      ( R2 > R3 ), THEN       // Start IF clause
//
//              Next statement executed if R2 > R3
//
                IF      ( R5 == R6), THEN   // Nested IF clause
//
//                  Statements here are executed if
//                      R2 > R3 && R5 == R6
//
                    ENDIF                   // End of nested IF clause
//
//
            ANDIF   ( R3 < R4 )             // One more condition
                                            //  for outer IF clause
//
//              Statements here are executed if R2 > R3 && R3 < R4
//
            ELSEIF  PreCode=                /> Load constant 16 for
                    [!"    LI   R10, 16"]   />  condition comparison
                    ( R5 == R10 )           // Condition R5 == 16
//
//              Statements here are executed if
//                  ( R2 <= R3 ) || R3 >= R4 ) && R5 == 16 (constant)
//
            ELSE                            // Last macro before ENDIF
                                            //  within same IF clause
//
//              Statements following ELSE are executed if
//                  ( R2 <= R3 ) || R3 >= R4 ) &&
//                      R5 != 16 (constant)
//
                ENDIF                       // End of IF clause
//
```

4.5.3 WHILE and ENDWHILE macros

The `WHILE` macro is used to start a loop.

```
  >--+---------+---WHILE---+---------------------------------------+------>
     |         |           |                                       |
     +--Label--+           +--RepeatCode=[repeatcode_statements],--+

  >--+-----------------------------------------------------------------+-><
     |                                                                 |
     +--Condition=condition--+------------------+--+----------------+--+
                             |                  |  |                |
                             +-,-WReg=register--+  +-,-Far=Boolean--+
```

`Label`	Optional
`Name`	`WHILE`
`Pos. Parameters`	
`None`	
`Key-word Parameters`	
`condition`	Condition tested to check if to continue the loop (see page 18).
`RepeatCode`	List of strings, where each string contains an assembler statement to be executed at every iteration before testing the condition.
`WReg`	Condition work register (see page 18).
`Far`	Boolean, to specify far branches (see page 18).

The condition is tested the first time the loop is entered, and every time the loop is executed thereafter. When condition is not specified, this is equivalent to an infinite loop that can be exited via the `BREAK` macro, `IF` with the `BREAK` action or the `ENDWHILE` macro with a condition.

The `RepatCode=` key-word parameter is a string array that contains assembly statements to be executed at each iteration of the loop, except when entering the loop for the first time. The `RepeatCode` parameter can be used to code a loop functionally equivalent to the `for` loop in the `C` language.

The `ENDWHILE` macro ends a while loop started with macro `WHILE`.

```
>--+---------+---ENDWHILE------------------------------------------------>
   |         |
   +--Label--+

>--+------------------------------------------------------------------+-><
   |                                                                  |
   +--Condition=condition--+-------------------+--+----------------+--+
                           |                   |  |                |
                           +-,-WReg=register---+  +-,-Far=Boolean--+
```

`Label`	Optional
`Name`	`ENDWHILE`
`Pos. Parameters`	
`None`	
`Key-word Parameters`	
`condition`	Condition tested to check if to continue the loop (see page 18). assembler statement to be executed at every iteration before testing the condition.
`WReg`	Condition work register (see page 18).
`Far`	Boolean, to specify far branches (see page 18).

When the condition is true at execution time, the loop is exited. When condition is false, or is not specified the loop is iterated.

4.5.4 BREAK and CONTINUE Macros

Macro `BREAK` can be used to unconditionally break out of the inner-most `WHILE` loop or `DO` block.

```
>--+---------+---BREAK-----+------------+-----------------------------><
   |         |             |            |
   +--Label--+             +--ID=label--+
```

`Label`	Optional
`Name`	`BREAK`
`Pos. Parameters`	
`None`	
`Key-word Parameters`	
`ID`	Label of an outer `WHILE` loop or `DO` block, to be the target of `BREAK`.

When there is a need to break out of a loop or block that is not the inner-most, the `ID=` key-word parameter can be used to specified the label of the loop or block being target for break-out.

Macro `CONTINUE` can be used to unconditionally iterate the inner-most `WHILE` loop or `DO` block.

```
>--+---------+---CONTINUE--+------------+------------------------------><
   |         |             |            |
   +--Label--+             +--ID=label--+
```

`Label`	Optional
`Name`	`BREAK`
`Pos. Parameters`	
`None`	
`Key-word Parameters`	
`ID`	Label of an outer `WHILE` loop or `DO` block, to be the target of `CONTINUE`.

When there is a need to iterate a loop or block that is not the inner-most, the `ID=` key-word parameter can be used to specified the label of the loop or block being iterated.

This is an example of a `WHILE` loop using both the `BREAK` and `CONTINUE` macro:

```
//
//  This example shows how to code the equivalent of C code
//
//      for (int i= 0; i < 256; i++)
//
//  also using break and continue
//
//          BASEDEF                         // Include definitions
//
            LI      T0, 0                   // Set index register to zero
            LI      T1, 256                 // Load upper bound of loop
            WHILE                           />
                    RepeatCode=[!"     ADDI    T0, 1"] />
                    Condition= ( T0 < T1)   // Same as C for loop
//
//              Statements here
//
                IF ( T3 > T4), BREAK        // Break out on condition
//
//              More statements
//
                IF ( T3 <= T5 ), CONTINUE   // Iterate loop on condition
//
//              More statements
//
                ENDWHILE                    // End of loop
//
```

4.5.5 DO and ENDDO Macros

The `DO` and `ENDDO` macros can be used to create a block of code that can be the target of `CONTINUE` and `BREAK` macros.

The `DO` macro starts a DO block:

```
>--+---------+---DO-----------------------------------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`DO`

`Pos. Parameters`

`None`

`Key-word Parameters`

`None`

The `ENDDO` macro ends a DO block:

```
  >--+---------+---ENDDO-------------------------------------------------><
     |         |
     +--Label--+
```

`Label` Optional

`Name` `ENDDO`

`Pos. Parameters`

`None`

`Key-word Parameters`

`None`

4.5.6 Fixing Branch to Branch

When using control flow macros, it can happen that the macros generate conditional branches to unconditional branches. This is a typical example:

```
//
//          BASEDEF                         // Include definitions
//
            IF      ( R2 > R3 ), THEN       // Outer IF clause
//
//              Statements here
//
                IF      ( R5 == R6), THEN   // If condition is false
                                            // Code branch to unconditional
                                            // branch of ELSE
//
//                  Statements here
//
                    ENDIF                   // End of nested IF clause
//
            ELSE                            // First machine instruction
                                            // of ELSE is unconditional
                                            // branch to ENDIF
                                            // It is also the target of
                                            // a conditional branch in
                                            // the IF macro
//
//              Statements here
//
                ENDIF                       // End of IF clause
//
```

DVASM™ RISC-V module check for these instances, and issues a warning for each statement with a conditional branch to an unconditional branch.

It is up to the coder to fix the code for optimal performance. Normally this is achieved by the use of a DO block in conjunction with the `BREAK` macro. This is how the previous example can be re-coded for optimal performance:

```
//
//          BASEDEF                         // Include definitions
//
            DO                              // Start DO block
                IF      ( R2 > R3 ), THEN       // Outer IF clause
//
//                  Statements here
//
                    IF      ( R5 != R6), BREAK  // Test for negated condition
                                                // If condition is true
                                                //  break out of the DO block
//
//                  Statements here
//
                ELSE
//
//                  Statements here
//
                    ENDIF                       // End of IF clause
                ENDDO                       // End of DO block
//
```

4.6 Linkage Assist Macros

Several macros are provided to generate code to transfer execution to an entry point just like in a `C` function call or tail call. Macros starting with `CALL.*` implement several flavors of function calls, where a return from the target entry point is expected. Macros starting with `TAIL.*` implement several flavors of tail calls, where a return from the target entry point is not expected.

4.6.1 Common Parameters

`CALL.*` and `TAIL.*` macros share the following common parameters:

`EP`	Positional: It indicates the entry point symbol to which execution will be transferred. Typically `EP` is an external symbol. This parameter is used by all `CALL.*` and `TAIL.*` macros except for `CALL.REG`.
`Reg`	Positional: It indicates the register containing the full address of the entry point to which execution will be transferred. This parameter is used only by `CALL.REG`.
`ARG=`	Keyword: It is a list containing the name of the arguments passed during the call. It is defaulted to an empty list `[]`. See below for a detailed description. This parameter is used by all `CALL.*` and `TAIL.*` macros except for `CALL.PRIME`.
`SaveReg=`	Keyword: It is a list containing the the name of the registers that must be saved and restored before and after the transfer to the entry point. It is defaulted to an empty list. This parameter is only used by all `CALL.*` macros with the exception of `CALL.PRIME`. When using this parameter only temporary registers should be specified in the list, since, according to the ABI, non-temporary registers must be saved and restored by the callee when being modified.
`SaveFReg=`	Keyword: It is a list containing the the name of the floating point registers that must be saved and restored before and after the transfer to the entry point. It is defaulted to an empty list `[]`. This parameter is used by all `CALL.*` macros except for `CALL.PRIME`. When using this parameter only temporary

	registers should be specified in the list, since according to the ABI, non-temporary registers must be saved by the callee when modified, and restored upon return.
`Stack=`	Keyword: It specifies a symbol, typically an `MMAP`, that indicates the start of the stack. It is defaulted to the null string (i.e. no stack used). This parameter is used by all `CALL.*` and `TAIL.*` macros except for `CALL.PRIME`. This parameter must be specified when requesting to save registers, or when the parameter list does not fit in the parameter registers.

Passing arguments in the RISC-V architecture is done by using argument registers (i.e. A0, A1, etc.). When there are more arguments than argument registers, the overflowing arguments are stored at the bottom of the stack. Either way each argument must be loaded in the appropriate A type register and, for an overflowing argument, must be stored at the bottom of the stack. The same convention is followed when passing arguments that are floating point numbers, with the only difference being that floating point argument registers are used.

Each item in the `ARG=` argument list contains two words. The first word is the opcode used to load the argument into the argument register, and the second word is the opcode parameter indicating the source of the argument. The opcode is enclosed in curly brackets, and when the opcode start with the 'F' character, it is assumed that the argument is a floating point number.

For example if a call to a `C` function requires two arguments currently in registers `S3` and `S5`, the coding for the arguments will look like:

```
ARG=[ {ADDI}  0[S3},  {ADDI} 0[S5] ]
```

and the code generated by the macro to set the argument will look like:

```
ADDI        A0, 0[S3]       // Load first argument
ADDI        A1, 0[S5]       // Load second argument
```

Note that the macro decides what A type register to use for each argument.

Since most of the time arguments are already loaded in some registers, when the opcode is not specified for an argument, it is defaulted to `ADDI`, so that the previous example can be re-coded as:

```
ARG=[ 0[S3}, 0[S5] ]
```

to generate the same exact code.

4.6.2 CALL.FAR Macro

The `CALL.FAR` macro is used to branch to an entry whose relative offset from the branch code must be within +/- 2 gigabytes. It is the preferred CALL macro when branching to a an entry which is part of the executable. In this case it generates assembly code that is PIC (Position Independent Code). However if the entry sits in a dynamic library, the code generated is not PIC.

```
>--+---------+----CALL.FAR----EP---------------------------------------->
   |         |
   +--Label--+

>----------+-----------------+---+-------------------------+------------>
           |                 |   |                         |
           +--,-ARG=Arglist--+   +--,-SaveReg=SaveRegList--+

>----------+---------------------------+---+-----------------------+---><
           |                           |   |                       |
           +--,-SaveFReg=SaveFRegList--+   +--,-Stack=StackSymbol--+
```

`Label`	Optional
`Name`	`CALL.FAR`
`Pos. Parameters`	
`EP`	Entry point symbol.
`Key-word Parameters`	
`ARG`	List of arguments (see page 37).
`SaveReg`	List of registers to be saved/restored (see page 37).
`SaveFReg`	List of floating point registers to be saved/restored (see page 37).
`Stack`	Symbol indicating the start of the stack being used (see page 37).

In the following example the `printf` function is called to print the number stored in register `S4` using a format string:

```
//
//  Base register S0 is set as base register to the code
//  to address constants such as Fmt

        EXTERNAL    printf
        CALL.FAR    printf, ARG=[ {LA} Fmt, 0[S4] ],  />
                    SaveReg=[T0-4], Stack= myStack


Fmt     UTF8         "The number is: %ld\n\u0000"
```

Note that the macro will save and restore registers `T0, T1, T2, T3,` and `T4` before and after the call. Also the object generated will not be PIC if the C function `printf` is in a dynamic library and is not statically linked in the executable.

4.6.3 CALL.NEAR Macro

The `CALL.NEAR` is the same as `CALL.FAR`, except that the entry point relative offset must be not more than +/- 1 megabyte. It should only be used when the entry point is part of the executable, and the executable itself is not larger than 1 megabyte. In this case it generates code that is PIC (Position Independent Code). If the executable is larger than 1 megabyte, special instructions should be given to the binder, so that the `CALL.NEAR` code and the target entry point are as close as possible in the final executable. `CALL.NEAR` should never be used when the entry point belongs to a dynamic library since there is very high probability that the entry point will be out of range at load time.

The `CALL.NEAR` macro is coded as follows:

```
>--+---------+----CALL.NEAR---EP---------------------------------------->
   |         |
   +--Label--+

>----------+-----------------+---+-------------------------+------------>
           |                 |   |                         |
           +--,-ARG=Arglist--+   +--,-SaveReg=SaveRegList--+

>----------+---------------------------+---+-----------------------+---><
           |                           |   |                       |
           +--,-SaveFReg=SaveFRegList--+   +--,-Stack=StackSymbol--+
```

`Label`	Optional
`Name`	`CALL.NEAR`
`Pos. Parameters`	
`EP`	Entry point symbol.
`Key-word Parameters`	
`ARG`	List of arguments (see page 37).
`SaveReg`	List of registers to be saved/restored (see page 37).
`SaveFReg`	List of floating point registers to be saved/restored (see page 37).
`Stack`	Symbol indicating the start of the stack being used (see page 37).

4.6.4 CALL.LCL Macro

The `CALL.LCL` macro is the same as the `CALL.NEAR`, except that the target entry point is a local symbol in the current source code.

The `CALL.LCL` macro is coded as follows:

```
>--+---------+----CALL.LCL----EP---------------------------------------->
   |         |
   +--Label--+

>----------+-----------------+---+-------------------------+------------>
           |                 |   |                         |
           +--,-ARG=Arglist--+   +--,-SaveReg=SaveRegList--+

>----------+---------------------------+---+-----------------------+---><
           |                           |   |                       |
           +--,-SaveFReg=SaveFRegList--+   +--,-Stack=StackSymbol--+
```

`Label`	Optional
`Name`	`CALL.LCL`
`Pos. Parameters`	
`EP`	Entry point symbol.
`Key-word Parameters`	
`ARG`	List of arguments (see page 37).
`SaveReg`	List of registers to be saved/restored (see page 37).
`SaveFReg`	List of floating point registers to be saved/restored (see page 37).
`Stack`	Symbol indicating the start of the stack being used (see page 37).

4.6.5 CALL.GOT Macro

The `CALL.GOT` macro is used to branch to an entry without any limit to the relative offset from the branch code to the entry point. It always generates PIC code, but cannot be used when coding boot loaders or kernels since no loader is available to create the GOT at boot time.

The `CALL.GOT` macro is coded as follows:

```
>--+---------+----CALL.GOT----EP---------------------------------------->
   |         |
   +--Label--+

>----------+-----------------+---+-------------------------+------------>
           |                 |   |                         |
           +--,-ARG=Arglist--+   +--,-SaveReg=SaveRegList--+

>----------+---------------------------+---+-----------------------+---><
           |                           |   |                       |
           +--,-SaveFReg=SaveFRegList--+   +--,-Stack=StackSymbol--+
```

`Label`	Optional
`Name`	`CALL.GOT`
`Pos. Parameters`	
`EP`	Entry point symbol.
`Key-word Parameters`	
`ARG`	List of arguments (see page 37).
`SaveReg`	List of registers to be saved/restored (see page 37).
`SaveFReg`	List of floating point registers to be saved/restored (see page 37).
`Stack`	Symbol indicating the start of the stack being used (see page 37).

This is the `CALL.FAR` example using `CALL.GOT`:

```
//
//  Base register S0 is set as base register to the code
//  to address constants such as Fmt

        EXTERNAL    printf
        CALL.GOT    printf, ARG=[ {LA} Fmt, 0[S4] ],  />
                    SaveReg=[T0-4], Stack= myStack


Fmt     UTF8         "The number is: %ld\n\u0000"
```

4.6.6 CALL.PRIME and CALL.FAST Macros

The `CALL.PRIME` and `CALL.FAST` macros are an optimized version of the `CALL.FAR` macro, to be used when calling the the same entry multiple time within the same code, for example in a loop. When using the `CALL.FAR` macro two immediate machine instructions are used. The first load the high 20 bits of the relative address, and the second branch to the entry point by filling the missing low 12 bits, for a total 32 bits relative address. When using the `CALL.PRIME` the high 20 bits are loaded. Later, when using the `CALL.FAST` macro, the low 12 bits are added to the register used in the `CALL.PRIME` macro to generate the branch address. The use of `CALL.PRIME` and `CALL.FAST` saves one machine instruction, when calling the entry point repeatedly, as in a loop.

The `CALL.PRIME` macro is coded as follows:

```
>--PrimeLabel ----CALL.PRIME----EP,----PrimeReg------------------------><
```

`Label`	Required
`Name`	`CALL.PRIME`
`Pos. Parameters`	
`EP`	Entry point symbol.
`PrimeReg`	Register holding the partial address of entry point, upon completion.
`Key-word Parameters`	
`None`	

The `CALL.FAST` macro is coded as follows:

```
   >--+---------+----CALL.FAST---PrimeLabel,---PrimeReg-------------------->
      |         |
      +--Label--+

   >----------+-----------------+---+-------------------------+------------>
              |                 |   |                         |
              +--,-ARG=Arglist--+   +--,-SaveReg=SaveRegList--+

   >----------+---------------------------+---+-----------------------+---><
              |                           |   |                       |
              +--,-SaveFReg=SaveFRegList--+   +--,-Stack=StackSymbol--+
```

`Label`	Optional
`Name`	`CALL.FAST`
`Pos. Parameters`	
`PrimeLabel`	Name of label used for `CALL.PRIME` macro.
`PrimeReg`	Register holding the partial address of symbol targeted by this call.
`Key-word Parameters`	
`ARG`	List of arguments (see page 37).
`SaveReg`	List of registers to be saved/restored (see page 37).
`SaveFReg`	List of floating point registers to be saved/restored (see page 37).
`Stack`	Symbol indicating the start of the stack being used (see page 37).

where `PrimeLabel` is the symbol used as label for the corresponding `CALL.PRIME` macro, and `PrimeReg` is the register used to load the high 20 bits in the `CALL.PRIME` macro.

In the following example the C library function `rand()` is called multiple times using `CALL.PRIME` and `CALL.FAST`.

```
            EXTERNAL    rand
//
//  Random function has already be initialized using srand function
//
randPrime   CALL.PRIME  rand, S4    // Set S4 to high 20 bits
                                    //  of rand() relative offset
//
            LI          S3, 100     // Load number of random numbers
            LI          S2, 0       // Load iteration counter
//
            WHILE       ( S2 < S3 )
                CALL.FAST   randPrime, S4, Stack=myStack
                                        // Get random number in A0
                ADDI    S2, 1           // Increment counter
                ENDWHILE
```

4.6.7 CALL.REG Macro

The `CALL.REG` macro work like the `CALL.FAR` macro except that the entry point address is passed in a register. This macro always generate PIC object code.

```
>--+---------+----CALL.REG----Reg--------------------------------------->
   |         |
   +--Label--+

>----------+-----------------+---+-------------------------+------------>
           |                 |   |                         |
           +--,-ARG=Arglist--+   +--,-SaveReg=SaveRegList--+

>----------+---------------------------+---+-----------------------+---><
           |                           |   |                       |
           +--,-SaveFReg=SaveFRegList--+   +--,-Stack=StackSymbol--+
```

`Label`	Optional
`Name`	`CALL.REG`
`Pos. Parameters`	
`Reg`	Register holding the address of symbol targeted by this call.
`Key-word Parameters`	
`ARG`	List of arguments (see page 37).
`SaveReg`	List of registers to be saved/restored (see page 37).
`SaveFReg`	List of floating point registers to be saved/restored (see page 37).
`Stack`	Symbol indicating the start of the stack being used (see page 37).

In the following example the `C` library function `rand()` is called multiple times using `CALL.REG`. Function `rand` address is first loaded using macro `LA.GOT`:

```
                EXTERNAL    rand
//
//  Random function has already be initialized using srand function
//
                LA.GOT      S4, rand    // Load rand entry point address
//
                LI          S3, 100     // Load number of random numbers
                LI          S2, 0       // Load iteration counter
//
                WHILE       ( S2 < S3 )
                    CALL.REG    S4, Stack= myStack
                                            // Get random number in A0
                    ADDI    S2, 1           // Increment counter
                    ENDWHILE
```

4.6.8 TAIL.FAR Macro

The `TAIL.FAR` macro is used to tail branch to an entry point whose relative offset from the branch code must be within +/- 2 gigabytes. It is the preferred `TAIL` macro when branching to a an entry which is part of the executable. In this case it generates code that is PIC (Position Independent Code). However if the entry sits in a dynamic library, the code generated is not PIC.

```
>--+---------+----TAIL.FAR----EP,--------------------------------------->
   |         |
   +--Label--+

>--------------------+-----------------+---+-----------------------+---><
                     |                 |   |                       |
                     +--,-ARG=ArgList--+   +--,-Stack=StackSymbol--+
```

`Label`	Optional
`Name`	`TAIL.FAR`
`Pos. Parameters`	
`EP`	Entry point symbol.
`Key-word Parameters`	
`ARG`	List of arguments (see page 37).
`Stack`	Symbol indicating the start of the stack being used (see page 37).

In the following example, the `noReturn` function is tail called. Contents of register `T2` and `T5` are passed as arguments, and the stack used starts at symbol `myStack`:

```
EXTERNAL    noReturn
TAIL.FAR    noReturn    ARG=[ T2, T5 ],  Stack= myStack
```

4.6.9 TAIL.NEAR Macro

The `TAIL.NEAR` is the same as `TAIL.FAR` except that the entry point relative offset must be not more than +/- 1 megabyte. It should only be used when the entry point is part of the executable, and the executable itself is not larger than 1 megabyte. In this case it generates code that is PIC (Position Independent Code). If the executable is larger than 1 megabyte, special instructions should be given to the binder, so that the `TAIL.NEAR` code and the target entry point are as close as possible in the final executable. `TAIL.NEAR` should never be used when the entry point belongs to a dynamic library, since there is a very high probability that the entry point will be out of range at load time.

```
>--+---------+----TAIL.NEAR---EP,---------------------------------------->
   |         |
   +--Label--+

>--------------------+-----------------+---+-----------------------+---><
                     |                 |   |                       |
                     +--,-ARG=ArgList--+   +--,-Stack=StackSymbol--+
```

`Label`	Optional
`Name`	`TAIL.NEAR`
`Pos. Parameters`	
`EP`	Entry point symbol.
`Key-word Parameters`	
`ARG`	List of arguments (see page 37).
`Stack`	Symbol indicating the start of the stack being used (see page 37).

4.6.10 TAIL.LCL Macro

The `TAIL.LCL` macro is the same as the `TAIL.NEAR`, except that the target entry point is a local symbol in the current source code.

```
>--+---------+----TAIL.LCL----EP,--------------------------------------->
   |         |
   +--Label--+

>--------------------+----------------+---+-----------------------+--->< 
                     |                |   |                       |
                     +--,-ARG=ArgList--+   +--,-Stack=StackSymbol--+
```

`Label`	Optional
`Name`	`TAIL.LCL`
`Pos. Parameters`	
`EP`	Entry point symbol.
`Key-word Parameters`	
`ARG`	List of arguments (see page 37).
`Stack`	Symbol indicating the start of the stack being used (see page 37).

4.6.11 TAIL.GOT Macro

The `TAIL.GOT` macro is used to branch to an entry without any limit to the relative offset from the branch code to the entry point. It always generates PIC code, but cannot be used when coding boot loaders or kernels, since no loader is available to create the GOT at boot time.

```
>--+---------+----TAIL.GOT----EP,--------------------------------------->
   |         |
   +--Label--+

>--------------------+-----------------+---+-----------------------+--->< 
                     |                 |   |                       |
                     +--,-ARG=ArgList--+   +--,-Stack=StackSymbol--+
```

`Label`	Optional
`Name`	`TAIL.GOT`
`Pos. Parameters`	
`EP`	Entry point symbol.
`Key-word Parameters`	
`ARG`	List of arguments (see page 37).
`Stack`	Symbol indicating the start of the stack being used (see page 37).

This is the same `TAIL.FAR` example using `TAIL.GOT`:

```
EXTERNAL    noReturn
TAIL.GOT    noReturn    ARG=[ T2, T5 ],  Stack= myStack
```

4.6.12 ENTRY Macro

The `ENTRY` macro generate all the necessary code needed at an entry point.

```
>--Label-------ENTRY---+------------------------------+------------------>
                       |                              |
                       +--,-Stack=StackSymbol---------+

>----------------------+------------------------------+------------------>
                       |                              |
                       +--,-Stamp=StampMacro----------+

>----------------------+------------------------------+------------------>
                       |                              |
                       +--,-BaseReg=BaseRegList-------+

>----------------------+------------------------------+-----------------><
                       |                              |
                       +--,-Export=Boolean------------+
```

`Label`	Required
`Name`	`ENTRY`
`Pos. Parameters`	
`None`	
`Key-word Parameters`	
`Stack`	Symbol indicating the start of the stack.
`Stamp`	Macro name used to create an eye catcher preceding the entry.
`BaseReg`	List containing the symbol name, within the entry code, and the register to used as its base.
`Stamp`	Boolean stating if the entry symbol must be exported.

The entry point is aligned to 8 bytes.

When `Stack=` is set to a non-null string, it refers to the symbol at the start of the current stack, mapped by the `STACK` and `EDNSTACK` macros. In this case the macro generates code to reset the stack register, and save the return address. It also generates code for all `S` and `FS` type registers to be saved when being modified before return to the caller. `Stack=`

is defaulted to the null string.

When `Stamp=` is set to the name of a macro provided by the coder, it generates code to invoke the specified macro with `Label` as the only argument. The coder can use the macro to prefix the entry point with a string containing such information as entry point name (eye catcher), date, time, version, etc., and anything else that can help the coder in debugging when reading a memory dump. `Stamp=` is defaulted to the null string.

When `BaseReg` is not empty, it must be a string array containing the name of a symbol and a register. The symbol indicates a location in the code where addressability is needed by using the specified register as base register. The code generated loads the symbol address plus 2k bytes, into the register specified, and it than issues a `BASESET` directive, so that the base register can be used to address 4k bytes past the symbol. Typically this is used to address literals and other constants stored at the end of the code. `BaseReg[2]=` is defaulted to an empty array.

When `Export` is set to `TRUE`, the macro generates code to export the entry point using the `EXPORT` directives. `Export` is defaulted to `TRUE`.

This is an `ENTRY` macro example:

```
//
myFunc      ENTRY       Stack= myStack, Stamp= BuildStamp,  />
                            BaseReg= [constants, S2]
//
//          Some code here
//
            LD          T3, junkData    // Load first data in junkData
                                        //  using default base register
//
//          More code here and constants at the end of the code
//
constants   LITERALS                    // Start constants with literals
//
junkData    DWRD        23, 45, 12345, 2435, 36, 5687
```

In this example, the name of the stack is `myStack`. Register `S2` is used as base register for the constants at the bottom of the code. Macro `BuildStamp` is used to prefix the entry point with a string, to help when reading memory dump. For example macro `BuildStamp` could be written as follows:

```
//MACRO     BuildStamp    EntryName

    var date= new Date();           // Get date and time
    var s= date.getMonth() + "/" +
        date.getDay() + "/" +
        (date.getYear()-100) + " " +
        date.getHours() + ":" +
        date.getMinutes() + ":" +
        date.getSeconds();          // Format date and time

    DVASM.formatLine("", "UTF_8", "\"" + EntryName +
        " V1.R2 " + s + "\'", "");  // Define prefix/eye-catcher string
```

In this case macro `BuildStamp` generate a prefix string containing, entry point name, version and release numbers, date, and time.

4.6.13 EXIT Macro

The `EXIT` macro is used to return control to the caller of an entry point.

Macro `EXIT` is coded as follows:

```
>--+---------+----EXIT----+----------------------------+---------------><
   |         |            |                            |
   +--Label--+            +--,-Stack=StackSymbol-------+
```

`Label`	Optional
`Name`	`EXIT`
`Pos. Parameters`	
`None`	
`Key-word Parameters`	
`Stack`	Symbol indicating the start of the stack.

When `Stack` is set to a non-null string, it refers to the symbol at the start of the current stack, mapped by the `STACK` and `EDNSTACK` macros. This must be the same symbol specified for the stack in the corresponding `ENTRY` macro. In this case the macro generates code to restore the stack register and the return address. It also generates code for all modified `S` and `FS` type registers to be restored.

4.6.14 STACK and ENDSTACK macros

Macros `STACK` and `ENDSTACK` are used together to map the section of the stack used by an entry point. Each stack section that is allocated when entering an entry point, is allocated downward, so that the caller stack section low address is next to the callee stack section high address. The layout of a stack section that is compatible RISC-V `C` compilers is as follows (from low address up):

- Area reserved for overflow parameters passed when calling another entry point. If there is no call to other entry points, or there is no parameter overflow, this area length is zero. This area is reserved by the `STACK` macro;
- Area containing local variables. This area is specified directly by the coder in between the `SATCK` and `ENDSTACK` macros;
- Area reserved to save non S type registers (integers) and non SF type registers (floating point) to be saved when calling another entry point (caller save). If no entry point is called this area length is zero;
- Area reserved to save S type registers (integer) and SF type registers (floating point) when modified. This area is reserved by the `ENDSTACK` macro and only for those registers that are modified (callee save). If the entry point uses only A and T type registers this area length is zero;
- Area reserved to save the stack register `SP`;
- Area reserved to save the return register `RA`.

Macro `STACK` maps the starts of a stack.

```
>--Label-------STACK---+-----------------------------+------------------><
                       |                             |
                       +--,--ParmArea=NumOfWords----+
```

`Label`	Required
`Name`	`STACK`
`Pos. Parameters`	
`None`	
`Key-word Parameters`	
`ParmArea`	Overflow parameter area size within the stack.

`Label` is required and it is used to name the new `MMAP`. Key-word parameter `ParmArea` specifies in words (32 bits architecture), or double words (64 bits architecture) the size of the parameter overflow area. The default is zero.

Macro `ENDSTACK`maps the end of a stack.

```
>--+---------+---ENDSTACK---+----------------------------+-------------->
   |         |              |                            |
   +--Label--+              +--,-SaveReg=SaveRegList-----+

>---------------------------+------------------------------+----------><
                            |                              |
                            +--,-SaveFReg=SaveFRegList-----+
```

`Label`	Optional
`Name`	`ENDSTACK`
`Pos. Parameters`	
`None`	
`Key-word Parameters`	
`SaveReg`	List of `S` type registers that will be saved when calling other functions.
`SaveReg`	List of `S` type floating point registers that will be saved when calling other functions.

Key-word Parameters `SaveReg` and `SaveFReg` specify the list of non S type and non SF type registers that need space reserved for saving and restoring when when calling other entry points. The default for both is an empty list. There is no need to specify the list of modified `S` type and `SF` type registers that need to save on entry and restored on exit. The space for saving these registers is automatically reserved by the `ENDSTACK` macro.

This is an example on how to use `STACK` and `ENDSTACK`:

```
//
//  Start stack mapping - 8 words or double words for
//  parameter overflow
//
myStack         STACK       ParmArea= 8
//
//  Start local variables
//
var1            DWRD        0
var2            BYTE        0 [16]
var3            HWRD        0
//
//  End stack - create space for save registers t0, t1, t2, t3, t4
//  when calling other entry points
//
                ENDSTACK    SaveReg=[t0-4]
//
```

4.7 External Symbols Access Macros

There are three type of external symbol access macros. The first type loads the address of an external symbol, the second load data at an external symbol location, and the third store data at an external symbol location.

Important: for repeated load and store to the same symbol, it is more efficient to load and use the address of the symbol, instead of using direct load and store macros to external symbols.

4.7.1 LA.FAR Macro

Macro `LA.FAR` generate code to load the address of an external symbol.

```
>--+---------+----LA.FAR-----Rd,------Symbol----------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`LA.FAR`
`Pos. Parameters`	
`Rd`	Register used to load the address of an external symbol.
`Symbol`	External symbol.
`Key-word Parameters`	
`None`	

4.7.2 LA.GOT Macro

Macro `LA.GOT` generate code to load the address of an external symbol from the GOT table.

```
>--+---------+----LA.GOT-----Rd,------Symbol---------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`LA.GOT`
`Pos. Parameters`	
`Rd`	Register used to load the address of an external symbol.
`Symbol`	External symbol.
`Key-word Parameters`	
`None`	

There are no limits to the size of the relative offset of the symbol location from the code generated. The code generated is PIC. The macro cannot be used in boot loader or OS kernels where a GOT is not available.

4.7.3 LB.FAR Macro

Macro `LB.FAR` generate code to load a byte located at an external symbol memory address.

```
>--+---------+----LB.FAR-----Rd,------Symbol---------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`LB.FAR`
`Pos. Parameters`	
`Rd`	Register used to load the byte at the external symbol address.
`Symbol`	External symbol.
`Key-word Parameters`	
`None`	

The symbol must be at a relative offset from the code generated of less than +/- 2 gigabytes. The code generated is PIC, unless the symbol sits in a dynamic library.

4.7.4 LH.FAR Macros

Macro `LH.FAR` generate code to load a half word located at an external symbol memory address.

```
>--+---------+----LH.FAR-----Rd,------Symbol---------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`LH.FAR`
`Pos. Parameters`	
`Rd`	Register used to load the half word at the external symbol address.
`Symbol`	External symbol.
`Key-word Parameters`	
`None`	

The symbol must be located at a relative offset from the code generated of less than +/- 2 gigabytes. The code generated is PIC, unless the symbol sits in a dynamic library.

4.7.5 LW.FAR Macro

Macro `LW.FAR` generate code to load a word located at an external symbol memory address.

```
>--+---------+----LW.FAR-----Rd,------Symbol---------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`LW.FAR`
`Pos. Parameters`	
`Rd`	Register used to load the word at the external symbol address.
`Symbol`	External symbol.
`Key-word Parameters`	
`None`	

The symbol must be located at a relative offset from the code generated of less than +/- 2 gigabytes. The code generated is PIC, unless the symbol sits in a dynamic library.

4.7.6 LD.FAR Macro

Macro `LD.FAR` generate code to load a double word located at an external symbol memory address.

```
>--+---------+----LD.FAR-----Rd,------Symbol---------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`LD.FAR`
`Pos. Parameters`	
`Rd`	Register used to load the double word at the external symbol address.
`Symbol`	External symbol.
`Key-word Parameters`	
`None`	

The symbol must be located at a relative offset from the code generated of less than +/- 2 gigabytes. The code generated is PIC, unless the symbol sits in a dynamic library.

4.7.7 SB.FAR Macro

Macro `SB.FAR` generate code to store a byte at the memory address of an external symbol.

```
>--+---------+----SB.FAR-----Rd,--Rt,--Symbol--------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`SB.FAR`
`Pos. Parameters`	
`Rd`	Register containing the byte to be stored at the external symbol address.
`Rt`	Work register.
`Symbol`	External symbol.
`Key-word Parameters`	
`None`	

The symbol must be located at a relative offset from the code generated of less than +/- 2 gigabytes. The code generated is PIC, unless the symbol sits in a dynamic library.

4.7.8 SH.FAR Macro

Macro `SH.FAR` generate code to store a half word at the memory address of an external symbol.

```
>--+---------+----SH.FAR-----Rd,--Rt,--Symbol--------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`SH.FAR`
`Pos. Parameters`	
`Rd`	Register containing the half word to be stored at the external symbol address.
`Rt`	Work register.
`Symbol`	External symbol.
`Key-word Parameters`	
`None`	

The symbol must be located at a relative offset from the code generated of less than +/- 2 gigabytes. The code generated is PIC, unless the symbol sits in a dynamic library.

4.7.9 SW.FAR Macro

Macro `SW.FAR` generate code to store a word at the memory address of an external symbol.

```
>--+---------+----SW.FAR-----Rd,--Rt,--Symbol--------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`SW.FAR`
`Pos. Parameters`	
`Rd`	Register containing the word to be stored at the external symbol address.
`Rt`	Work register.
`Symbol`	External symbol.
`Key-word Parameters`	
`None`	

The symbol must be located at a relative offset from the code generated of less than +/- 2 gigabytes. The code generated is PIC, unless the symbol sits in a dynamic library.

4.7.10 SD.FAR Macro

Macro `SD.FAR` generate code to store a double word at the memory address of an external symbol.

```
>--+---------+----SD.FAR-----Rd,--Rt,--Symbol--------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`SD.FAR`
`Pos. Parameters`	
`Rd`	Register containing the double word to be stored at the external symbol address.
`Rt`	Work register.
`Symbol`	External symbol.
`Key-word Parameters`	
`None`	

The symbol must be located at a relative offset from the code generated of less than +/- 2 gigabytes. The code generated is PIC, unless the symbol sits in a dynamic library.

4.8 Buffer Handling Macros

Macros named `BUFFER.*` are provided to initialize, copy and compare buffers. The macro generate high performance code that whenever possible works using the full width of the registers (32 or 64 bits).

Important: TO take full advantage of the `BUFFER.*` macros all buffers should be 4 byte aligned for 32 bit architectures and 8 byte aligned for 64 bit architectures.

4.8.1 BUFFER.LDPADREG Macro

The `BUFFER.LDPADREG` macro initialize a register so that each byte in the register equals the provided pad byte.

```
>--+---------+----BUFFER.LDPADREG-----PadReg,---Char,---WReg----------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`BUFFER.LDPADREG`
`Pos. Parameters`	
`PadReg`	Register to be padded.
`Char`	Character to be used for padding.
`WReg`	Work register, only needed for 64 bits architectures.
`Key-word Parameters`	
`None`	

For some of the `BUFFER.*` macros a pad byte is used. In order to take advantage of the full register width, a register, used as pad register, must be first initialized, so that each byte in the register is set to the pad byte.

When the pad byte is zero (all bits 0) there is no need to use `BUFFER.LDPADREG`, and register `X0` should be used as pad register.

This is an example on how to use `BUFFER.LDPADREG`:

```
      InitPad     BUFFER.LDPADREG     T3, '-', T0     // Set pad register
                                                      //  to '-' character
```

4.8.2 BUFFER.SET Macro

The `BUFFER.SET` macro initialize all bytes in a buffer to the same values.

```
>--+---------+----BUFFER.SET-----Buffer,---Length,---PadReg------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`BUFFER.SET`
`Pos. Parameters`	
`Buffer`	Register containing the buffer address.
`Length`	Register containing the buffer length.
`PadReg`	Pad register.
`Key-word Parameters`	
`None`	

Upon completion, register `Buffer` will contain the address of the first byte following the buffer, register `Length` value is unpredictable, and register `PadReg` value is unchanged.

This is an example on how to use `BUFFER.SET`:

```
InitPad     BUFFER.LDPADREG     T0, '-', T1 // Set pad register
                                            //  to '-' character
Set         BUFFER.SET      S2, S3, T0      // Set buffer to '-'
```

In the following example a 4K page is cleared:

```
ClearPage   LI            T0, 4096          // Load page length
ClearPage   BUFFER.SET    S2, T0, X0        // CLear page at addr S2
```

4.8.3 BUFFER.COPY Macro

The `BUFFER.COPY` macro copy the content of a buffer to another buffer with the same length.

```
>--+---------+----BUFFER.COPY----ToBuffer,---FromBuffer,---Length,------>
   |         |
   +--Label--+

>-------------------------------Wreg---------------------------------><
```

`Label`	Optional
`Name`	`BUFFER.COPY`
`Pos. Parameters`	
ToBuffer	Register containing the destination buffer address.
FromBuffer	Register containing the source buffer address.
Length	Register containing both buffers length.
WReg	Work register.
`Key-word Parameters`	
None	

Upon completion, register `ToBuffer` contains the address of the first byte following the destination buffer. Register `FromBuffer` contains the address of the first byte following the source buffer. Register `Length` value is unpredictable.

This is an example on how to use `BUFFER.COPY`:

```
Copy        BUFFER.COPY     S2, S3, S4, T0  // Copy data at addr S2
                                            //  to addr S3
                                            // Length is in S4
```

4.8.4 BUFFER.COPYPAD Macro

The `BUFFER.COPYPAD` macro copy the content of a buffer to another buffer of different length.

```
>--+---------+----BUFFER.COPYPAD----ToBuffer,---ToLength,---------------->
   |         |
   +--Label--+

>-------------------------FromBuffer,---FromLength,-------------------->

>-------------------------PadReg,---WReg------------------------------><
```

`Label`	Optional
`Name`	`BUFFER.COPYPAD`
`Pos. Parameters`	
`ToBuffer`	Register containing the destination buffer address.
`ToLength`	Register containing the destination buffer length.
`FromBuffer`	Register containing the source buffer address.
`FromLength`	Register containing the source buffer length.
`WReg`	Work register.
`PadReg`	Pad register.
`Key-word Parameters`	
`None`	

If the source buffer length is longer, the data is truncated. If the source buffer length is shorter, the data is padded using a previously initialized pad register.

Upon completion, register `ToBuffer` contains the address of the first byte following the destination buffer. Register `FromBuffer` contains the address of the first byte following the source buffer. Registers `ToLength` and `FromLength` values are unpredictable. Register `PadReg` is unchanged.

This is an example on how to use `BUFFER.COPYPAD`:

```
InitPad     BUFFER.LDPADREG     T0, ' ', T1
                            // Set pad register to blank spaces

Copy        BUFFER.COPYPAD  S2, S3, S4, S5, T0, T1
                            // Copy data at addr S2 to addr S4
                            // IF S3 < S5 data is truncated
                            // IF S3 > S5 data is padded
                            //  with blank spaces
```

4.8.5 BUFFER.COMP Macro

The `BUFFER.COMP` macro compare the content of a buffer to another buffer. The length of both buffers is the same.

```
>--+---------+----BUFFER.COMP--------Buffer1,---Buffer2,---Length,------>
   |         |
   +--Label--+

>------------------------------Branch,---WReg--------------------------><
```

`Label`	Optional
`Name`	`BUFFER.COMP`
`Pos. Parameters`	
`Buffer1`	Register containing the first buffer address.
`Buffer2`	Register containing the second buffer address.
`Length`	Register containing both buffers length.
`Branch`	List of two labels. The first label is branched to if the first buffer is less than the first. The second label is branched to if the first buffer is greater than the second.
`WReg`	List containing two work registers.
`Key-word Parameters`	
`None`	

If the two buffers are equal no branch takes place.

Upon completion, the contents of registers `Buffer1`, `Buffer2`, `Length`, and the work registers are unpredictable.

This is an example on how to use `BUFFER.COMP`:

```
Comp        BUFFER.COMP     S2, S3, S4,     /> Compare data at addr S2
                                            />  with data at addr S3
                                            />  with length in S4
                            [LessThen, GreaterThen] /> Branch targets
                            [T0-1]          // Work registers

//
//  Equal code starts here
//


//
//  Less than code starts here
//
LessThen    BYTE            0[0]

//
//  Greater than code starts here
//
GreaterThen BYTE            0[0]
```

4.8.6 BUFFER.COMPPAD Macro

The `BUFFER.COMPPAD` macro compare the content of a buffer to another buffer with different lengths.

```
>--+---------+----BUFFER.COMPPAD-----Buffer1,---Length1,---Buffer2,----->
   |         |
   +--Label--+

>-------------------Length2,---PadReg,---Branch,---WReg----------------><
```

`Label`	Optional
`Name`	`BUFFER.COMPPAD`
`Pos. Parameters`	
`Buffer1`	Register containing the first buffer address.
`Length1`	Register containing the first buffer length.
`Buffer2`	Register containing the second buffer address.
`Length2`	Register containing the second buffer length.
`PadReg`	Pad register.
`Branch`	List of two labels. The first label is branched to if the first buffer is less than the first. The second label is branched to if the first buffer is greater than the second buffer.
`WReg`	List containing two work registers.
`Key-word Parameters`	
`None`	

If the two buffers have unequal length, the shorter buffer is logically padded with with bytes in the pad register. If the two buffers are equal no branch takes place.

Upon completion, the contents of registers `Buffer1`, `Length1`, `Buffer2`, `Length2`, and the work registers are unpredictable. Register `PadReg` is unchanged.

This is an example on how to use `BUFFER.COMPPAD`:

```
InitPad     BUFFER.LDPADREG     T0, ' ', T1 // Set pad register
                                            // to blank spaces

Comp        BUFFER.COMPPAD  S2, S3, S4, S5, /> Compare data at addr S2
                                            /> and length in s3
                                            />  with data at addr S4
                                            />  with length in S5
                            [LessThen, GreaterThen], /> Branch targets
                            T0,             /> Pad register
                            [T1-2]          // Work registers

//
//  Equal code starts here
//


//
//  Less than code starts here
//
LessThen    BYTE            0[0]

//
//  Greater than code starts here
//
GreaterThen BYTE            0[0]
```

4.9 String Handling Macros

String handling macros are equivalent to the `C` language library string functions. Each string must be a `C` language compatible string and it **must** be terminated by a null byte.

4.9.1 STRING.CLEAR Macro

Macro `STRING.CLEAR` sets the length of a string to zero.

```
>--+---------+----STRING.CLEAR-------String-----------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`STRING.CLEAR`
`Pos. Parameters`	
`String`	Register containing the string address.
`WReg`	List containing two work registers.
`Key-word Parameters`	
`None`	

Parameter `String` is a register that contains the address of the string and it remains unchanged.

4.9.2 STRING.COMP Macro

Macro `STRING.COMP` compares two strings and is functionally equivalent to the `C` language function `strcmp`.

```
>--+---------+----STRING.COMP--------String1,---String2,---Branch,------>
   |         |
   +--Label--+

>-----------------------------WReg----------------------------------><
```

`Label`	Optional
`Name`	`STRING.COMP`
`Pos. Parameters`	
`String`	Register containing the first string address.
`String`	Register containing the second string address.
`Branch`	List of two labels. The first label is branched to if the first string is less than the second. The second label is branched to if the first string is greater than the second.
`WReg`	List containing two work registers.
`Key-word Parameters`	
`None`	

If the two strings have unequal length, the shorter string is logically padded with null bytes. If the two strings are equal no branch takes place.

Upon completion, the contents of registers `String1` and `String2` are unpredictable.

This is an example on how to use `STRING.COMP`:

```
Comp        STRING.COMP         S2, S3,     /> Compare string at addr S2
                                            />  with string at addr s3
                            [LessThen, GreaterThen], /> Branch targets
                            [T1-2]          // Work registers

//
//  Equal code starts here
//


//
//  Less than code starts here
//
LessThen    BYTE            0[0]

//
//  Greater than code starts here
//
GreateThen                  0[0]
```

4.9.3 STRING.CONCAT Macro

Macro `STRING.CONCAT` concatenates two strings and is functionally equivalent to the `C` language function `strcat`.

```
>--+---------+----STRING.CONCAT------ToString,---FromString,---WReg---->< 
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`STRING.CONCAT`
`Pos. Parameters`	
`ToString`	Register containing the destination string address.
`FromString`	Register containing the source string address.
`WReg`	Work registers.
`Key-word Parameters`	
`None`	

Upon completion, register `ToString1` will contain the address of the byte following the terminating null byte of the destination string, and register `FromString` will contain the address if the byte following the null terminating byte of the source string. Content of register `WReg` is unpredictable.

This is an example on how to use `STRING.CONCAT`:

```
      Comp        STRING.CONCAT   S2, S3, T0  // Concatenate string at addr S3
                                              //  to string at addr s2
```

4.9.4 STRING.COPY Macro

Macro `STRING.COPY` copies a string to another string and is functionally equivalent to the `C` language function `strcpy`.

```
>--+---------+----STRING.COPY--------ToString,---FromString,---WReg---><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`STRING.COPY`
`Pos. Parameters`	
`ToString`	Register containing the destination string address.
`FromString`	Register containing the source string address.
`WReg`	Work registers.
`Key-word Parameters`	
`None`	

Upon completion, register `ToString1` will contain the address of the byte following the terminating null byte of the destination string, and register `FromString` will contain the address if the byte following the null terminating byte of the source string. Content of register `WReg` is unpredictable.

This is an example on how to use `STRING.COPY`:

```
Comp        STRING.COPY     S2, S3, T0  // Copy string at addr S3
                                        //  into string at addr S2
```

4.9.5 STRING.LENGTH Macro

Macro `STRING.LENGTH` computes the length of a string and is functionally equivalent to the `C` language function `strlen`.

```
>--+---------+----STRING.LENGTH------Length,---String,---WReg---------->< 
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`STRING.LENGTH`
`Pos. Parameters`	
`Length`	Register containing the string length upon completion.
`String`	Register containing the string address.
`WReg`	Work registers.
`Key-word Parameters`	
`None`	

Upon completion, register `String` will contain the address of the byte following the terminating null byte of the. Content of register `WReg` is unpredictable.

This is an example on how to use `STRING.LENGTH`:

```
      Comp        STRING.COPY     A0, S2, T0  // Calculate length of string
                                              //  at address S2 and
                                              //  store result in reg A0
```

4.10 HEAPSORT Macro

The `HEAPSORT` macro is used to sort a vector of words or double words.

```
>--+---------+----HEAPSORT-------BaseAddr,---Size,---RecSize,----------->
   |         |
   +--Label--+

>--------------------------CompMacro,---WReg---------------------------><
```

`Label`	Optional
`Name`	`HEAPSORT`
`Pos. Parameters`	
`BaseAddr`	Register containing the vector address.
`Size`	Register containing the number of records in the vector.
`RecSize`	Constant integer specifying the size in bits of the records in the vector to be sorted. It can only be 32 (word) or 64 (double word).
`CompMacro`	is the name of a comparison macro, provided by the coder, that is used by `HEAPSORT` to check the order of sorting of two records.
`WReg`	List of a minimum of 9 work registers (see below).
`Key-word Parameters`	
`None`	

The records to be sorted can contain the values to be sorted or addresses of structures to be sorted.

Code generated by macro `HEAPSORT` invokes the comparison macro in multiple places. The header of the comparison macro must follow this template:

```
//MACRO   MacroName   E1Addr, E2Addr, {Boolean}EqFlag, BrLbl, WReg[]= []
```

where the parameters are:

`E1Addr`	Register containing value of first element in vector to be compared;
`E2Addr`	Register containing value of second element in vector to be compared;
`EqFlag`	A Boolean flag indicating if comparison should include equality or not. This is necessary to guarantee consistent order sorting;
`BrLbl`	A branch label. The code generated by the macro must branch to this label if the first element is less then or less-equal than the second element. Less-equal must be used only when the equality flag is set to `true`;
`WReg`	A list of work registers starting from the 10th registers in the `WReg` list passed to macro `HEAPSORT` onward. For example if the comparison macro needs 3 work registers, than macro `HEAPSORT` must be invoked passing a `WReg` list with 12 registers so that the 10th, 11th and 12th registers are passed in a list of work registers to the comparison macro.

In the following example a function that can be called from a `C` program sort a double words vector in ascending order;

```
//------------------------------------------------------------------
//  Heap sort function using 64 bit integers - ascending order
//
//  Parameters: A0  Address of vector
//              A1  Size of vector
//------------------------------------------------------------------
//
            SETENV      "RISCV",                            />
                        "RV64I:a,c,d,m,n,zicsr,zifencei",   />
                        "LP64D", "linux"
//
#MACRO
//MACRO     CompLong    R1, R2, EqualFlag, BranchLabel, WReg[]= []
    if (EqualFlag)
\#Label     BLE         #R1, #R2, #BranchLabel
    else
\#Label     BLT         #R1, #R2, #BranchLabel
#END
//
            BASEDEF
//
Code        TextSect
//
sortlong    ENTRY       Stack=!""   // Leaf entry point
                                    //  only temp reg used
//
            HEAPSORT    a0, a1, 64, CompLong, [t0-6, a2-3]
//
            EXIT
            END
```

4.11 Linked List Macros

DVASM™ RISC-V module provides macros to support three types of linked lists. These are:

- single linked lists with one head pointer as anchor. These lists are normally used as stacks. They are named `SLL.*`;
- single linked lists with both head and tail pointer as anchor. These lists are normally used as queues. They are named `STLL.*`;
- doubly linked circular lists with a single head/tail pointer in the anchor. They are named `CDLL.*`.

4.11.1 Anchor Mapping Macros

Macros `*.ANCHOR` map the fields used to anchor each type of linked list.

```
>--+---------+-----+--CDLL.ANCHOR--+------------------------------------><
   |         |     |               |
   +--Label--+     +--SLL.ANCHOR---+
                   |               |
                   +--STLL.ANCHOR--+
```

`Label`	Optional

`Name`	`CDLL.ANCHOR`
	`SLL.ANCHOR`
	`STLL.ANCHOR`

`Pos. Parameters`

`None`

`Key-word Parameters`

`None`

In the following example:

```
myAnachor           STLL.ANCHOR         // Anchor for STLL list
```

the following code is generated for the 64 bits architecture:

```
myAnachor           DWRD        0[0]    // Start of anchor
myAnchor.head       DWRD        0       // Head pointer of list
myAnchor.tail       DWRD        0       // Tail pointer of list
```

When the same example is used without a label the code generated is:

```
                    DWRD        0[2]   // Space for STLL anchor
```

4.11.2 Anchor Initialization Macros

Macros `*.INIT` are used to initialized an anchor structure for each type of linked list.

```
>--+---------+---+--CDLL.INIT--+----Anchor-----------------------------><
   |         |   |             |
   +--Label--+   +--SLL.INIT---+
                 |             |
                 +--STLL.INIT--+
```

`Label`	Optional

`Name`	`CDLL.INIT`
	`SLL.INIT`
	`STLL.INIT`

`Pos. Parameters`	
`Anchor`	It is the addressable anchor field of the linked list. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Key-word Parameters`	
`None`	

Initialization is needed when the anchor structure sits in dynamic storage such as the stack.

In the following example macro `CDLL.INIT` is used to initialize a CDLL anchor allocated in the stack:

```
//
myEntry      ENTRY       Stack=myStack
//
             CDLL.INIT   MyAnchor    // Initialize anchor
//
//  Code using the CDLL list here
//
             EXIT
//
myStack      STACK
//
myAnchor     CDLL.ANCHOR             // Space in stack for CDLL anchor
//
             STACKEND
//
             END
```

4.11.3 Node Mapping Macros

Macros `*.NODE` map the pointers within each list node used to link the list.

```
  >--+---------+-----+--CDLL.NODE--+-------------------------------------><
     |         |     |             |
     +--Label--+     +--SLL.NODE---+
                     |             |
                     +--STLL.NODE--+
```

`Label`	Optional

`Name`	`CDLL.NODE`
	`SLL.NODE`
	`STLL.NODE`

`Pos. Parameters`

None

`Key-word Parameters`

None

In the following example:

```
      myNode              CDLL.NODE           // Node for CDLL list
```

the following code is generated for the 64 bits architecture:

```
      myNode              DWRD        0[0]    // Start of node
      myNode.next         DWRD        0       // Pointer to next node
      myNode.prev         DWRD        0       // Pointer to prev node
```

When the same example is used without label the code generated is:

```
                          DWRD        0[2]   // Space for CDLL node
```

4.11.4 Push Macros

Macros `*.PUSH` push a node into a corresponding linked list.

```
>--+---------+---+--CDLL.PUSH--+----Anchor,---Node,---WReg-------------->
   |         |   |             |
   +--Label--+   +--SLL.PUSH---+
                 |             |
                 +--STLL.PUSH--+

>--------------------------+-------------------------------------+-----><
                           |                                     |
                           +---,--FieldDispl=fieldDIsplacement---+
```

`Label`	Optional

`Name`	`CDLL.PUSH`
	`SLL.PUSH`
	`STLL.PUSH`

`Pos. Parameters`	
`Anchor`	It is the addressable anchor field of the linked list. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Node`	Register containing the address of the node to be pushed.
`WReg`	List containing two work registers for macro `CDLL.PUSH`. A single work register for macros `SLL.PUSH` and `STLL.PUSH`.
`Key-word Parameters`	
`FieldDispl`	Displacement of the node list pointer field within each node. It is defaulted to zero.

In the following example a node is pushed in a doubly linked list:

```
      //
      //      Anchor address is in register S2
      //      Node address is in register T1
      //      Displacement of node list pointers is zero
      //
              CDLL.PUSH   0[S20], T1, WReg=[T4, T5]
      //
```

4.11.5 Pop Macros

Pop macros `*.POP` pop a node from a corresponding linked list.

```
>--+---------+---+--CDLL.POP--+----Anchor,---Node,---NullLbl,---WReg---->
   |         |   |            |
   +--Label--+   +--SLL.POP---+
                 |            |
                 +--STLL.POP--+

>--------------------------+-------------------------------------+----->< 
                           |                                     |
                           +---,--FieldDispl=fieldDIsplacement---+
```

`Label`	Optional

`Name`	`CDLL.POP`
	`SLL.POP`
	`STLL.POP`

`Pos. Parameters`	
`Anchor`	It is the addressable anchor field of the linked list. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Node`	Register containing, upon completion, the address of the node to be popped.
`NullLbl`	Label to be branched to, if the list is empty.
`WReg`	List containing two work registers for macro `CDLL.POP`. A single work register for macros `SLL.POP` and `STLL.POP`.
`Key-word Parameters`	
`FieldDispl`	Displacement of the node list pointer field within each node. It is defaulted to zero.

In the following example a node is popped from a linked list:

```
//
//      Anchor address is in register S2
//      Node address is in register S3, if list is not empty
//      Displacement of node list pointers is zero
//
        SLL.POP     0[S20], T1, EmptyList, WReg=T4
//
//  Insert code to handle the popped node
//
//
EmptyList   WRD     0[0]    // Empty list code
                            //  starts here
//
```

4.11.6 Tailpush Macros

Macros `*.TAILPUSH` push a node into a corresponding linked list at the tail end of the list.

```
>--+---------+---+--CDLL.TAILPUSH--+----Anchor,---Node,---WReg---------->
   |         |   |                 |
   +--Label--+   +--STLL.TAILPUSH--+

>--------------------------+--------------------------------------+----><
                           |                                      |
                           +---,--FieldDispl=fieldDIsplacement---+
```

`Label`	Optional
`Name`	`CDLL.TAILPUSH` `STLL.TAILPUSH`
`Pos. Parameters`	
`Anchor`	It is the addressable anchor field of the linked list. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Node`	Register containing the address of the node to be pushed.
`WReg`	List containing two work registers for macro `CDLL.TAILPUSH`. A single work register for macro `STLL.TAILPUSH`.
`Key-word Parameters`	
`FieldDispl`	Displacement of the node list pointer field within each node. It is defaulted to zero.

In the following example a node is pushed at the tail end of a linked list, with both head and tail pointers:

```
//
//      Anchor address is in register S2
//      Node address is in register T1
//      Displacement of node list pointers is zero
//
        STLL.TAILPUSH   0[S20], T1, WReg=[T4, T5]
//
```

4.11.7 CDLL.TAILPOP Macro

Macro `CDLL.TAILPOP` pops a node from the tail end of a doubly linked list.

```
>--+---------+---+--CDLL.TAILPOP--+----Anchor,---Node,---NullLbl,------->
   |         |
   +--Label--+

>---------------------WReg---+-------------------------------------+---><
                             |                                     |
                             +---,--FieldDispl=fieldDIsplacement---+
```

`Label`	Optional
`Name`	`CDLL.TAILPOP`.
`Pos. Parameters`	
`Anchor`	It is the addressable anchor field of the linked list. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Node`	Register containing, upon completion, the address of the node to be popped.
`NullLbl`	Label to be branched to, if the list is empty.
`WReg`	List containing two work registers.
`Key-word Parameters`	
`FieldDispl`	Displacement of the node list pointer field within each node. It is defaulted to zero.

In the following example, a node is popped from the tail of a circular doubly linked list:

```
//
//      Anchor address is in register S2
//      Node address is in register S3, if list is not empty
//      Displacement of node list pointers is zero
//
        CDLL.TAILPOP    0[S20], T1, EmptyList, WReg=T4
//
//  Insert node to handle the popped node
//
//
EmptyList   WRD     0[0]    // Empty list code
                            //  starts here
//
```

4.11.8 Find Macros

Macros `*.FIND` find the first node in a corresponding list.

```
>--+---------+---+--CDLL.FIND--+----Anchor,---CompToken,---Node,-------->
   |         |   |             |
   +--Label--+   +--SLL.FIND---+
                 |             |
                 +--STLL.FIND--+

>--------------------------CompMacro,---NullLbl,---WReg----------------->

>--------------------------+-------------------------------------+------>
                           |                                     |
                           +---,--FieldDispl=fieldDIsplacement---+

>--------------------------+---------------------------+--------------><
                           |                           |
                           +---,--Remove=Boolean-------+
```

`Label`	Optional

`Name`	`CDLL.FIND`
	`SLL.FIND`
	`STLL.FIND`

`Pos. Parameters`	
`Anchor`	It is the addressable anchor field of the linked list. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`CompToken`	Register whose value is used, by the comparison macro, to locate the node being searched for.
`Node`	Register containing, upon completion, the address of the node found.
`CompMacro`	Name of a comparison macro, provided by the coder, used to check if the current node matches the search criteria (see below for more details).
`NullLbl`	Label to be branched to, if no node is found.

WReg	List containing the macro work registers, plus the registers needed by the comparison macro. Macro `CDLL.FIND` require 3 work registers when the `Remove` option is used, 1 register otherwise. Macros `SLL.FIND` and `STLL.FIND` requires 2 work register when the `Remove` option is used, none otherwise.

Key-word Parameters

FieldDispl	Displacement of the node list pointer field within each node. It is defaulted to zero.
Remove	Boolean value that indicates if the node found should be remove (true) or not (false). It is defaulted to false.

The comparison macro header must follow this template:

```
//MACRO     MacroName   CompToken, NodeAddr, MatchLbl, WReg[]
```

where the parameters are:

CompToken	This is the same `CompToken` register that is passed to the `FIND.*` macros, and it is used to decide if a node is a match;
NodeAddr	Register containing the address of the node being checked;
MatchLbl	Branch label to which the code generated by the comparison macro will branch if the node is a match;
WReg	A list of work registers. When the remove flag is set to true, this list starts from the 3rd register in the `WReg` list passed to macro `STLL.FIND` onward, and the 4th register in the `WReg` list passed to `CDLL.FIND` onward. When the remove flag is set to false, this list starts from the 1st register in the `WReg` list passed to macro `STLL.FIND` onward, and the 2nd register in the `WReg` list passed to `CDLL.FIND` onward. For example, when the remove flag is true, if the comparison macro needs 1 work register, than macro `CDLL.FIND` must be invoked passing a `WReg` list with 4 registers, so that the 4th register is passed in a list of work registers to the comparison macro.

In the following example a node is searched in a linked list with both head and tail pointers. If a matching node is found it will be removed.

```
//
//  Macro to find a node with matching Data01 field
//
#MACRO
//MACRO     CompData    CompToken, NodeAddr, MatchLbl, WReg[]
    var WReg0= WReg[0];
\#Label     LD          #WReg0, Data01-Node[#NodeAddr]
                                        // Load data from element
\           BEQ         #WReg0, #CompToken, #MachLbl
                                        // Branch to process match
#END
//
//      Anchor address is in register S2
//      Comparison token is in register S3
//      Node address is in register S4, if node is found
//
        STLL.TAILPOP    0[S20], S3, S4, CompData, NotFound, />
                            Remove=yes, WReg=[T2-5],        />
                            FieldDisp= NodeLnk-Node
                                        // List pointers are not
                                        // at displacement zero in node
//
//  Insert code to handle the node found and removed from list
//
//
NotFound    WRD     0[0]    // No node found code
                            //  starts here
//
//  Node mapping
//
Node    MMAP                // Start node mapping
Data01  DWRD        0       // Field used to find node
Nodelnk CDLL.NODE           // Space for list pointers
//
```

4.11.9 CDLL.REMOVE Macro

Macros `CDLL.REMOVE` removes node from a circular doubly linked list.

```
>--+---------+---+--CDLL.REMOVE--+----Anchor,---Node,---WReg------------>
   |         |
   +--Label--+

>----------------------------+-------------------------------------+--->< 
                             |                                     |
                             +---,--FieldDispl=fieldDIsplacement---+
```

`Label`	Optional
`Name`	`CDLL.REMOVE`.
`Pos. Parameters`	
`Anchor`	It is the addressable anchor field of the linked list. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Node`	Register containing the address of the node to be removed.
`WReg`	List containing two work registers.
`Key-word Parameters`	
`FieldDispl`	Displacement of the node list pointer field within each node. It is defaulted to zero.

In the following example a node is remove from a doubly linked list.

```
//
//      Anchor address is in register S2
//      Node address is in register S4
//
        CDLL.REMOVE     0[S20], S4, WReg=[T2, T3]
//
```

4.12 AVL Tree Macros

DVASM™ RISC-V module provides macros to support all standard operations with AVL trees.

4.12.1 AVL.ANCHOR Macro

Macro `AVL.ANCHOR` maps the fields used to anchor an AVL tree.

```
>--+---------+-----AVL.ANCHOR------------------------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`AVL.ANCHOR`

`Pos. Parameters`

`None`

`Key-word Parameters`

`None`

In the following example:

```
myAnachor           AVL.ANCHOR          // Anchor for AVL tree
```

the following code is generated for the 64 bits architecture:

```
myAnachor           DWRD        0[0]    // Start of anchor
myAnchor.head       DWRD        0       // Head pointer of list
```

When the same example is used without a label the code generated is:

```
                    DWRD        0      // Space for AVL tree anchor
```

4.12.2 AVL.INIT Macro

Macro `AVL.INIT` is used to initialize an AVL tree anchor structure.

```
>--+---------+-----AVL.INIT--------Anchor-----------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`AVL.INIT`
`Pos. Parameters`	
`Anchor`	It is the addressable anchor field of the linked list. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Key-word Parameters`	
`None`	

Initialization is needed when the anchor structure sits in dynamic storage, such as the stack. In the following example macro `AVL.INIT` is used to initialize an AVL tree anchor allocated in the stack:

```
//
myEntry     ENTRY       Stack=myStack
//
            AVL.INIT   MyAnchor     // Initialize anchor
//
//  Code using the AVL tree here
//
            EXIT
//
myStack     STACK
//
myAnchor    AVL.ANCHOR              // Space in stack for AVL tree anchor
//
            STACKEND
//
            END
```

4.12.3 AVL.NODE Macro

Macros `AVL.NODE` map the link pointers within each tree node used to link the tree.

```
>--+---------+-----AVL.NODE-------------------------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`AVL.NODE`

`Pos. Parameters`

`None`

`Key-word Parameters`

`None`

It is important to note that the low three bits of the parent node pointer are used as attributes of the node, and for this reason **each node must be aligned on 8 bytes boundaries even when using 32 bits processors**.

In the following example

```
myNode      AVL.NODE                // Node for AVL tree
```

`AVL.NODE` generates the following code for 64 bits architecture:

```
#Label      #wrd    0[0]            // Start of node pointer fields
#Label..left   #wrd    0            // Node pointer to left child
#Label..parent #wrd    0            // Node pointer to parent
//
//  Low 3 bits of parent link are used as link and balance flags as follows:
//    Bit 1-0 -> 0bx00 (0) on when node right subtree is one deeper than left subtree
//    Bit 1-0 -> 0bx01 (1) on when node right subtree and left subtree have equal depth
//    Bit 1-0 -> 0bx10 (2) on when node left subtree is one deeper than right subtree
//    Bit 1-0 -> 0bx11 INVALID - this is a state error
//    Bit 2   -> 0b0xx when parent link belongs to parent right child
//    Bit 2   -> 0b1xx when parent link belongs to parent left child
//
//    !!! Node address MUST BE 8 BYTE ALIGNED in order to use parent link 3 low bits !!!
#Label..right #wrd    0             // Node pointer to right child
```

When the same example is used without label the code generated is:

```
DWRD    0[3]        // Space for AVL tree node pointers
```

4.12.4 AVL.INSERT Macro

Macros `AVL.INSERT` inserts a node into an AVL tree.

```
>--+---------+---+--AVL.INSERT----Anchor,---Node,---CompMacro,-------->
   |         |
   +--Label--+

>-------------------------------DuplLbl,---WReg----------------------->

>------------------------+-------------------------------------+-----><
                         |                                     |
                         +---,--FieldDispl=fieldDIsplacement---+
```

`Label`	Optional
`Name`	`AVL.INSERT`
`Pos. Parameters`	
`Anchor`	It is the addressable anchor field of the AVL tree. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Node`	Register containing the address of the node to be pushed.
`CompMacro`	Name of a comparison macro, provided by the coder, used by `AVL.INSERT` to check the sorted order of two nodes (see below).
`DuplLbl`	Label to branched to, if the node to be inserted is found to be a duplicate.
`WReg`	List of either 8 work registers or 2 plus the number of work registers used by the comparison macro, whichever list is larger.
`Key-word Parameters`	
`FieldDispl`	Displacement of the node pointers field within each node. It is defaulted to zero.

The header of the comparison macro must follow this template:

```
//MACRO     MacroName  Node1, Node2, GtLbl, EqLbl, WReg[]
```

where

`Node1`	Register containing the address of the first node to be compared.
`Node1`	Register containing the address of the second node to be compared.
`GtLbl`	Label to which the code generated by the macro will branch, if the key of the first node is larger than the key of the second node.
`EqLbl`	Label to which the code generated by the macro will branch, if the key of the first node is equal to the key of the second node. This is the same label passed to the `AVL.INSERT` macro as `DuplLbl`.
`WReg`	List containing the work registers for the comparison macro.

In the following example a node is inserted in an AVL tree using 64 bit architecture:

```
//
//      Anchor address is in register S2
//      Node address is in register S3
//      Tree link pointers are at the beginning of the node
//
//      Comparison macro
//
#MACRO
//MACRO     CompLong  Node1, Node2, GtLbl, EqLbl, WReg[]
    var WReg0= WReg[0];
    var WReg1= WReg[1];

\#Label     LD      #WReg0, nodeKey-node[#Node1]    // Load data from node 1
\           LD      #WReg1, nodeKey-node[#Node2]    // Load data from node 2
\           BGT     #WReg0, #WReg1, #GtLbl          // Branch GT label
\           BEQ     #WReg0, #WReg1, #EqLbl          // Branch EQ label
#END
//
            AVL.INSERT  0[S2], S3, CompLong, duplKey, [t0-6,a2]
//
//  Code following insert
//
//
//  Duplicate key handler code starts here
//
duplKey     WRD     0[0]
//
//
//  Node mapping
//
node        MMAP
            AVL.NODE        // Insert space for tree links
nodeKey     DWRD        0   // Space for 64 bits key
//
```

4.12.5 AVL.FIND Macro

Macros `AVL.FIND` finds a node in an AVL tree.

```
>--+---------+---+--AVL.FIND----Anchor,---Node,---Key,---CompMacro,--->
   |         |
   +--Label--+

>-------------------------------NullLbl,---WReg----------------------->

>-----------------------+--------------------------------------+----->< 
                        |                                      |
                        +---,--FieldDispl=fieldDIsplacement---+
```

`Label`	Optional
`Name`	`AVL.FIND`
`Pos. Parameters`	
`Anchor`	It is the addressable anchor field of the AVL tree. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Node`	Register containing, upon completion, the address of the node found.
`Key`	Register containing the key, or the key address, used to search the node.
`CompMacro`	Name of a comparison macro, provided by the coder, used by `AVL.FIND` to search for the node (see below).
`NullLbl`	Label to be branched to, if no matching node is found.
`WReg`	List of either 1 work registers or the work registers used by the comparison macro, whichever list is larger.
`Key-word Parameters`	
`FieldDispl`	Displacement of the node list pointer field within each node. It is defaulted to zero.

The key field by the `AVL.FIND` and the sort order <u>must be the same</u> as those used in the `ACL.INSERT` macro.

The comparison macro must be coded to generate code that compare the key passed to the `AVL.FIND` macro with a node key. The header of the comparison macro must follow this template:

```
//MACRO     MacroName  Key, Node, GtLbl, EqLbl, WReg[]
```

`Key`	This is the same `Key` parameter register passed to `AVL.FIND` macro whose content is unchanged.
`Node`	Register containing the address of a node.
`GtLbl`	Label to which the code generated by the macro will branch if the key identified by the key is greater than the key in `Node`.
`EqLbl`	Label to which the code generated by the macro will branch if the key is equal to the key in the node (i.e. a match has been found).
`WReg`	is a list containing the work registers needed by the comparison macro.

In the following example an attempt is made to find a node using 64 bit architecture:

```
//
//      Anchor address is in register S2
//      Node address is returned in register S3
//      Key is in register T0
//      Node address is in register T1
//      Tree link pointers are at the beginning of the node
//
//      Comparison key-node macro
//
#MACRO
//MACRO     CompKeyNode     Key, Node, GtLbl, EqLbl, WReg[]
    var WReg0= WReg[0];

\#Label     LD          #WReg0, nodeKey[#Node]  // Load data from element 1
\           BGT         #Key, #WReg0, #GtLbl    // Branch GT label
\           BEQ         #Key, #WReg0, #EqLbl    // Branch EQ label
#END

//
            AVL.FIND  0[S2], S3, T0, T1, CompkeyNode, notFound, [t1]
//
//  Code handling found node - node is in register S2
//
//
//  Node not found handler code starts here
//
notFound    WRD     0[0]
//
//
//  Node mapping
//
node        MMAP
            AVL.NODE        // Insert space for tree links
nodeKey     DWRD        0   // Space for 64 bits key
//
```

4.12.6 AVL.REMOVE Macro

Macros `AVL.REMOVE` removes a node from an AVL tree.

```
>--+---------+------AVL.REMOVE---Anchor,---Node,---WReg--------------->
   |         |
   +--Label--+

>------------------------+-------------------------------------+-----><
                         |                                     |
                         +---,--FieldDispl=fieldDIsplacement---+
```

`Label`	Optional
`Name`	`AVL.REMOVE`
`Pos. Parameters`	
`Anchor`	It is the addressable anchor field of the AVL tree. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Node`	Register containing the address of the node to be removed.
`WReg`	List containing 8 work registers.
`Key-word Parameters`	
`FieldDispl`	Displacement of the node list pointer field within each node. It is defaulted to zero.

In the following example a node is removed from an AVL tree:

```
//
//      Anchor address is in register S2
//      Node address is in register T1
//      Tree link pointers are at the beginning of the node
//
//
            AVL.FIND  0[S2], T1, [A0-3, T1-4]
//
```

4.12.7 Tree Traversal Macros

Several macros are provided to traverse an AVL binary tree in in-order, inverse in-order, pre-order and post-order. Inverse in-order traversal is used to traverse the tree in opposite sorting order, as opposed to in-order which traverse the tree in sorted order.

The macros are divided into two groups. One set of macros are used to locate the first node in the tree to start the traversal. Another set is provided to locate the next node in the traversal.

The macros used to locate the first node in a traversals are (AVL.LOCFIRST), for in-order traversal, `AVL.LOCLAST` for reverse in-order traversal, and `AVL.LOCROOT` for pre-order and post-order traversals.

```
>--+---------+---+--AVL.LOCROOT---+--Anchor,---Node,---NullLbl,------->
   |         |   |                |
   +--Label--+   +--AVL.LOCLAST---+
                 |                |
                 +--AVL.LOCLAST---+

>-----------------WReg---+-------------------------------------+-----><
                         |                                     |
                         +---,--FieldDispl=fieldDIsplacement---+
```

`Label`	Optional

`Name`	`AVL.LOCROOT`
	`AVL.LOCFIRST`
	`AVL.LOCLAST`

`Pos. Parameters`	
`Anchor`	It is the addressable anchor field of the AVL tree. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Node`	Register containing the address of the node to be located.
`Node`	Label to be branched to, if the tree is empty.
`WReg`	Work register.
`Key-word Parameters`	
`FieldDispl`	Displacement of the node list pointer field within each node. It is defaulted to zero.

The macros used to locate the next node in a traversals are `AVL.LOCNEXT`, for in-order traversal, `AVL.LOCPREV` for reverse in-order traversal, `AVL.LOCPRENEXT` for pre-order traversal and `AVL.LOCPOSTNEXT` for post-order traversal.

```
>--+---------+---+--AVL.LOCNEXT------+---Node,---NullLbl,------------->
   |         |   |                   |
   +--Label--+   +--AVL.LOCPREV------+
                 |                   |
                 +--AVL.LOCPRENEXT---+
                 |                   |
                 +--AVL.LOCPOSTNEXT--+

>-----------------WReg---+-------------------------------------+-----><
                         |                                     |
                         +---,--FieldDispl=fieldDIsplacement---+
```

`Label`	Optional

`Name`	`AVL.LOCNEXT`
	`AVL.PREV`
	`AVL.LOCPRENEXT`
	`AVL.LOCPOSTNEXT`

`Pos. Parameters`	
`Node`	Register containing the address of the node to be located.
`Node`	Label to be branched to, if there are no more nodes to traverse.
`WReg`	Work register.
`Key-word Parameters`	
`FieldDispl`	Displacement of the node list pointer field within each node. It is defaulted to zero.

In the following example an AVL tree is traversed in reverse in-order:

```
//
//      Anchor address is in register S2
//      Node address is in register T1
//      Tree link pointers are at the beginning of the node
//
//
            AVL.LOCLAST   [S2], T1, Done, T2
//
            WHILE           // Unconditional loop
                AVL.LOCPREV     T1, Done, T2
                ENDWHILE
//
Done        WRD     0[0]    // Code after traversal starts here
//
```

4.12.8 AVL.FREEALL Macro

Macro `AVL.FREEALL` is used to free all nodes in an AVL tree.

```
>--+---------+----AVL.FREEALL----Anchor,---FreeMacro,---WReg---------->
   |         |
   +--Label--+

>-----------------------+-------------------------------------+----->< 
                        |                                     |
                        +---,--FieldDispl=fieldDIsplacement---+
```

`Label`	Optional
`Name`	`AVL.FREEALL`
`Pos. Parameters`	
`Anchor`	It is the addressable anchor field of the AVL tree. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`FreeMacro`	Name of a free macro, provided by the coder, used by `AVL.FIND` to dispose of the node being freed (see below).
`WReg`	List of 2 work registers plus the registers used by the free macro.
`Key-word Parameters`	
`FieldDispl`	Displacement of the node list pointer field within each node. It is defaulted to zero.

The free macro is used to dispose of each node right after it has been removed from the tree. The header of this macro must follow this template:

```
//MACRO     MacroName  Node, WReg[]
```

where

`Node`	Register containing the address of the node just removed from the tree.
`WReg`	List containing the work registers for the `FreeMacro` macro.

In the following example all nodes in an AVL tree are removed and added to a stack for later reuse:

```
//
//      Anchor address is in register S2
//      Tree link pointers are at the beginning of the node
//
#MACRO
//MACRO     FreeNode     Node, WReg[]
    var wreg= WReg[0];
\#Label     SLL.PUSH    stackAnch,  #Node,  #wreg // Push node in stack
#END
//
            AVL.FREEALL   0[S2], FreeNode, [T2-4]
//
```

4.13 Hash Macros

DVASM™ RISC-V module provides hashing macros. The hashing function used by this macros is the Multiplicative Hashing function as described in https://en.wikipedia.org/wiki/Hash_function#Multiplicative_hashing.

The algorithm used is very efficient, since it uses only one multiplication, but requires that both the hash vector size and the size of the cells forming the vector to be power of twos.

4.13.1 HASH.ANCHOR Macro

The `HAHS.INIT` map the hash vector anchor.

```
>--+---------+-----HASH.INIT-------------------------------------------><
   |         |
   +--Label--+
```

`Label` Optional

`Name` `HASH.ANCHOR`

`Pos. Parameters`

  `None`

`Key-word Parameters`

  `None`

Every hashing macro that generates code to hash a word, double word, buffer or string need access to a control area called anchor. The `HASH.ANCHOR` macro maps this control area, which contains the hashing vector address, the prime number used for hashing, and the left and right shifts values used during hashing. Macro `HASH.ANCHOR` does not have any parameters, and the label is optional. When a label is not specified a memory buffer, sized to contain the anchor, is defined. Otherwise each anchor field is defined, by prefixing the label value to the field name, separated by a dot ('.').

In the following example macro `ANCHOR.ANCHOR` is used with a label in 64 bits architecture: for a hash anchor:

```
myAnachor           HASH.ANCHOR     // Anchor for STLL list
```

The code generated is:

```
myAnachor           DWRD    0[0]    // Start of hash anchor
myAnachor.vaddr     DWRD    0       // Address of hash vector
myAnachor.prime     DWRD    0       // Prime number used for hashing
myAnachor.keyshit   WRD     0       // Key shift
myAnachor.cellshift WRD     0       // Cell shift
```

The same macro used without label generates the following code:

```
myAnachor           DWRD    0[3]    // Hash anchor
```

HASH.INIT Macro

Macro `HASH.INIT` initialize the hash anchor area.

```
>--+---------+-----HASH.INIT------Anchor,---Vector,---VectSizeExp,---->
   |         |
   +--Label--+

>-----------------------------CellSizeExp,---WReg------------------><
```

`Label`	Optional
`Name`	`HASH.INIT`
`Pos. Parameters`	
`Anchor`	It is the addressable hash anchor field. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Vector`	Register containing the address of the hash vector.
`VectSizeExp`	Register containing the exponent, base 2, of the number of cells within the hash vector.
`CellSizeExp`	Register containing the exponent, base 2, of the size, in bytes, of each cell in the hashing vector.
`WReg`	List containing two work registers.
`Key-word Parameters`	
`None`	

Before use, the hash anchor area needs to be initialized using the `HASH.INIT` macro.

The macro `HASH.INIT` randomly selects a prime number out of a list of 128 prime numbers. To select the prime number, the code generated uses the lowest 7 bits returned by the machine instruction `RDCYCLE`, which returns the number of CPU cycles since power up. This index is used to select the prime from a list of pre-defined primes, that the macro stores in the literal pool. For this reason the `LITERAL` directive must be added at the end of the code, and addressability must be established to it using the `BASEDEF` directive. For example, the base register parameter of the `ENTRY` macro could be used (see page 55).

The random selection of a prime at execution time, is intended as a partial protection from denial of service attacks when the data being hashed is received through the internet. However, even though the selection is random, the list of primes used for selection is pre-defined and a determined attacker could target each prime in 128 separate attacks to create a very high number of collisions when the the prime selected for hashing is used by the attacker. For this reason, for absolute protection, each cell in the hash vector should be the root of an AVL tree, which would defeat any attack with minimal overhead.

In the following example macro `HASH.INIT` is used to initialize a hash table anchor:

```
//
//  Anchor addr is in register S1
//  Hash vector addr is in register S2
//  Hash vector size exponent is in registers S3
//  Hash vector cell size exponent is in registers S4
//
//  Addressability has been established to the literal pool
//
            HASH.INIT  0[S1], S2, S3, S4, [T0, T1]
//
//  Add the literal pool at the end of the code
//
    LITERALS
//
```

4.13.2 HASH Macro

Macro `HASH` is used to hash a word in 32 bit architecture and a double word in 64 bit architecture.

```
>--+---------+-----HASH----Anchor,---Key,---WReg----------------------->
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`HASH`
`Pos. Parameters`	
`Anchor`	It is the addressable hash anchor field. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Key`	Register containing, on input, the value to be hashed, and, on output, the corresponding hash table cell address.
`WReg`	List containing two work registers.
`Key-word Parameters`	
`None`	

In the following example a key is hashed into a has table cell address in 64 bit architecture:

```
//
//      Anchor address is in register S2
//      Key address is in register S3
//
            HASH    0[S2], S31, [t0-1]
//
```

4.13.3 HASH.BUFFER Macro

Macro `HASH.BUFFER` is used to hash a key in a buffer with no length limitations.

```
>--+---------+-----HASH.BUFFER----Anchor,---Buffer,---Length,-------->< 
   |         |
   +--Label--+

>----------------------------HashAddr,---WReg----------------------->
```

`Label`	Optional
`Name`	`HASH.BUFFER`
`Pos. Parameters`	
`Anchor`	It is the addressable hash anchor field. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`Buffer`	Register containing the address of the buffer to be hashed.
`Length`	Register containing the length, in bytes, of the buffer to be hashed.
`HashAddr`	Output register that, upon completion, contains the computed hash table cell address.
`WReg`	List containing two work registers.
`Key-word Parameters`	
`None`	

In the following example a key in a buffer is hashed into a hash table cell address:

```
//
//      Anchor address is in register S2
//      Buffer address is in register S3
//      Buffer length is in register S4
//      Hash address is returned in register T3
//
            HASH    0[S2], S3, S4, T3, [T0-2]
//
```

4.13.4 HASH.STRING Macro

Macro `HASH.STRING` hashes a `C` string (i.e null terminated) with no length limitations.

```
>--+---------+-----HASH.STRING----Anchor,---String,------------------><
   |         |
   +--Label--+

>-----------------------------HashAddr,---WReg------------------------>
```

`Label`	Optional
`Name`	`HASH.STRING`
`Pos. Parameters`	
`Anchor`	It is the addressable hash anchor field. If the address of the anchor is in a register, say `S0`, parameter `Anchor` must be coded as `0[S0]`.
`String`	Register containing the address of the string to be hashed.
`HashAddr`	Output register that, upon completion, contains the computed hash table cell address.
`WReg`	List containing three work registers.
`Key-word Parameters`	
`None`	

In the following example a key in a buffer is hashed into a hash table cell address:

```
//
//      Anchor address is in register S2
//      String address is in register S3
//      Hash address is returned in register T3
//
            HASH    0[S2], S3, T3, [T0-2]
//
```

4.14 Miscellaneous Macros

4.14.1 PRINTF Macro

The `PRINTF` macro provide an interface to the `C printf` function available in Linux and in Unix. It uses the macro `CALL.GOT` to call function `printf`.

```
>--+---------+-----PRINTF----Format,---Arg,---SaveReg,--------------->
   |         |
   +--Label--+

>-----------------------------SaveFReg,---Stack---------------------><
```

`Label`	Optional
`Name`	`PRINTF`
`Pos. Parameters`	
`Format`	Printf format string enclosed in double quotes (see below).
`Arg`	List containing the parameters to the `printf` function that follows the format string. It must be coded as documented by the `CALL.*` macros (see page 37).
`SaveReg`	List of registers that must be save/restored before/after the call to printf. Must be coded as documented by the `CALL.*` macros (see page 37).
`SaveFReg`	List of floating point registers that must be save/restored before/after the call to printf. Must be coded as documented by the `CALL.*` macros (see page 37).
`Stack`	Label referencing the current stack.
`Key-word Parameters`	
`None`	

Macro `printf` uses the macro `CALL.GOT` to call function `printf`. Parameter `Format` is the `printf` format string enclosed in double quotes and null terminated. The string is stored in the literal pool which must be addressable from the code generated by the macro. For this reason the `LITERAL` directive must be added at the end of the code, and addressability must be established to it using the `BASEDEF` directive. For example, the base register parameter of the `ENTRY` macro could be used (see page 55).

In the following example two registers S2 and S3 are printed to `stdout` in hexadecimal format:

```
//
//
//
            PRINTF                                                    />
                "Register  S2 -> %016X\nRegister  S3 -> %016X\u0000", />
                Arg=[ 0[S2], 0[S3] ], SaveReg=[T0-3], Stack= myStack
//
//  Following code here
//
            LITERALS                // Literal pool at end of code
//
```

4.14.2 Spin Lock Macros

Two macros are provided to acquire and release a spin lock, using `AMOSWAP.*.AQ` and `AMOSWAP.*.RL`. These macros use words and double words lockwords depending on the architecture being 32 bit or 64 bits.

Spin locks should be used with the CPU in disabled state, to avoid performance degradation. If the lock is shared from both mainline code and disable interrupt code, than the lock **must** be acquired and held only in disable stated to avoid deadlocks.

Macro `LOCK.GET` gets a spin lock.

```
>--+---------+-----LOCK.GET-----LockAddr,---WReg---------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`LOCK.GET`
`Pos. Parameters`	
`LockAddr`	Register containing the lock word or double word address, depending if the architecture is 32 or 64 bits.
`WReg`	Work register.
`Key-word Parameters`	
`None`	

Macro `LOCK.FREE` release a spin lock.

```
>--+---------+-----LOCK.FREE----LockAddr-----------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`LOCK.FREE`
`Pos. Parameters`	
`LockAddr`	Register containing the lock word or double word address, depending if the architecture is 32 or 64 bits.
`Key-word Parameters`	
`None`	

In this example a lock is acquired and released:

```
//
//  Register S2 contains the address of the lock word
//  CPU is disabled for interrupts
//
        LOCK.GET    S2, T1      // Get spin lock
//
//  Code executing in locked mode here
//
        LOC.FREE    S2          // Release spin lock
//
```

4.14.3 Load Immediate 32 Bit Constant Macro

Macro `LI32` load a 32 bit constant with two machine instruction.

```
>--+---------+-----LI32----Reg,---Value------------------------------><
   |         |
   +--Label--+
```

`Label`	Optional
`Name`	`LI32`
`Pos. Parameters`	
`Reg`	Destination register.
`Value`	Expression which resolve to an integer of not more than 32 significant bits.
`Key-word Parameters`	
`None`	

This is an example on how to use macro `LI32`:

```
              LI32    S2, 0x123ABC99  // Load a 32 bit immediate
```