## The Virtual Machine

# Overview

The Limp is a virtual machine ideallized by Erick S. Oliveira, the only goal is just to create a programmable enviorment for operational system ideas, or just in hardware softwares. The intent is to create a customizable Machine, customizable Components, and a programable runner.

The system is predominant 32-bit, its address access is of 32-bit and any arithmetical operation is of 32-bit. The system deals with words and long words data in Little-Endian.

The following concepts are adopted when threading with this system specifically:

* Byte is a 8-bit value;
* Word is 16-bit wide, or 2 bytes;
* Dword is 32-bit wide, or 4 bytes;
* Qword is 64-bit wide, or 8 bytes;
* 1 Hz is one instruction cycle execution (1 for second);
* 1 MHz are equal to 1,048,576 or 220 instructions cycles (for second).

# Cycling and States

The processor speed is measured in Hz (Hertz per second), or usually, MHz (Mega hertz per second). Each memory access consumes 1 Hz to execute (wich means, for complex instructions operations, may require high ammount of cycles to execute, like, read or write to memory, or even worse, **mov** through memory instructions), also, instructions operations may require more cycles to execute, this implies on his cycles base.

The system has two cycling states: the BUSY state, and the PHASE state. In the BUSY state, the system cannot handdle any external interruption, and stay stucked until a defined ammount of cycles is consumed. The instructions execution is a atomic process (which means, once started, it may not be interrupted by others devices), even passing this limit, once started the instruction execution, will be finished, but, the next BUSY state will be discounted.

The PHASE state is composed of n BUSY states, between each BUSY state, the cpu is able to handle one external request at time, the system have a well defined ammount of 64 PHASE states for second, which means, the cycling of machine is of 64x(ammount of BUSY states per PHASE)x(ammount of cycles per BUSY state).

Also, the system has two more states: the WAIT and HALT. When the system is put in WAIT, it waits for any interruption, but if is put in specific interruption code, it will go out from WAIT state to handle the requesteds interruptions, but, if do not match with the expected, when returning from it, will become again to WAIT mode. HALT mode merely stops the system execution, and cannot exit, except in case of any other external device request to it.

# Registers and Fast storages

The system contains 2 registers set, the standard (**RegS**) and the extra set (**RegE**), all 32-bit, the standard set registers are:

* **AX** => The accumulator of system, makes pair with **DX**;
* **DX** => The data holder and comparator, makes pair with **AX**;
* **CX** => The counter and indexer, makes pair with **BX**;
* **BX** => The base register and operand, makes pair with **CX**;
* **FP** => The pointer for program frame, makes pair with **SP**;
* **SP** => The stack pointer of system, makes pair with **FP**;
* **SS** => The stream source index, makes pair with **SD**;
* **SD** => The stream destination, makes pair with **SS**;

The extra set registers are:

* **E0**
* **E1**
* **E2**
* **E3**
* **E4**
* **E5**
* **E6**
* **E7**

These are two separated registers set, they cannot be freely used simultaneous in the same instruction, but, has others rules appliable: As operand, neither can be used at same time, except for instructions with AMD argument, which can only use registers from the standard set, but, the developer can use the extra set for others arguments. And finally, has two instructions wich allow to interchange value between differents registers sets, the **mvres** and **mvrse**.

Also, the system has others internal registers, but cannot be accessed directly in instructions operations, they are:

* **EPC** => Program counter register, indicates where to fetch the next dword.
* **ST** => System status register, stores the control flags and results statuses.
* **LPC** => Last program counter, used for relative jumps, is always updated before each instruction fetch and execution.
* **EFD** => Fetched Data, stores the last fetched data (aka, instruction storager for decoder).
* **IT** => Interruption table, stores the absolute address in bus of interruption vector for interruptions calling.
* **SSIV** => Stack Set on Interruption from Virtual, stores the absolute pointer to set to stack on dealing with a interruption call from vitual mode.

Besides the registers, system can also access to itself cache, which is other fast storage for data saving, using the instructions **mvfc** and **mvtc**. This operation is protected mode saved, which means, no any protected program procedure can access to cache.

# System Status

The system status are stored in the register **EST**, they keep the system flags and operations results. The status is composed of two parts, the software word (from bit zero to 15) and system word (from bit 16 to 31), the software word can be freely accessed in protected mode, likely be changed, but on other side, the system word is protected from access in protected mode, for security reasons. The available flags are:

Software word:

* **CF**:0 => Carry Flag, store carry status on sum instructions operations;
* **BF**:1 => Borrow Flag, store borrow status on subtraction instructions operations;
* **VF**:2 => Overflow Flag, store overflow status on sum and subtraction instructions operations;
* **ZF**:3 => Zero Flag, set if some operations results zero;
* **NF**:4 => Negative Flag, set if the last bit of some operations is setted;
* **OF**:5 => Odd Flag, set if the first bit of some operations is setted, also, used to verify some statuses.

System word:

* **EI**:16 => Enable Interruption, if set, will able the system to hear to external interruptions requests;
* **PM**:17 => Protected Mode, if set, the system operations will be restricted, any over control instruction will be enabled, and any try to access theses features will trigger a violation interruption;
* **VM**:18 => Virtual Mode, if the system has a MMU and this flag is set, will enable translation of any memory access address (this is not enabled for interruptions, cause the system must access the right vectors);
* **CS**:24..31 => Current Interruption, this special flag holds whats the selector of actual interruption.

# Interruption

As any other processor, the system can handle and deal with interruptions request, as from the system self, or requested by external periphericals. The goal of interruptions is deal with a requested feature from system (in case of programs or hardwares), or response to a request (from hardwares), or also to alert the system a execution and process have completed (also from hardwares and softwares), but is mainly for system deal with wrong operations, likely, access a not allowed address or operation in protected mode, or deal with divisions by zero.

The interruption process is: once the system detected a interruption request, first detect where from, if is from himself proceeds, if from external periphericals, detect if system has **EI** activated, if not, sends a not executed signal to device and keep its task, else, proceeds with the interruption dealing. The system query for interruption selector, the current **SP**, **ST** and **EPC** are pushed in this order, if system is on **Virtual Mode**, sets the value from internal register **SSIV** to **SP**, the system disables the flags **PM**, **VM** and **EI**, jumps to address stored in the vector in selector and goes out the wait state.

To return from interruption, the system is put in wait state if waiting was set to a specific selector and the current interruption do not match, else, do not stay in wait state. And the register **EPC**, **ST** and **SP** are restored from stack in this order.

The following is the interruption vector:

|  |  |
| --- | --- |
| **Index** | **Cause** |
| 0 | -- Invalid Interruption Selector -- |
| 1 | Violation of privilege access |
| 2 | Invalid Opcode |
| 3 |  |
| 4 | Denied Memory data Access |
| 5 | Denied Code Access |
| 6 |  |
| 7 | Division by Zero |
| 8 |  |
| 9 | Debugger Interruption |
| 10 | Coprocessor 0 Interruption |
| 11 | Coprocessor 1 Interruption |
| 12 | Coprocessor 2 Interruption |
| 13 | Coprocessor 3 Interruption |
| 14 | BIOS Interruption System Reserved 1 |
| 15 | BIOS Interruption System Reserved 2 |
| 16 -> 127 | Software Reserved |
| 128 -> 255 | Hardware Reserved |

# Coprocessor and MMU

The coprocessors are on side processors for the main processor. The Limp architeture can handle up to 4 coprocessors in max. For this, the system has special registers to deal with.

You can deal with them, by modifying their registers, sending commands, enabling or disabling them and verifying their state.

The MMU is also a coprocessor, just to allow to programmer to access their registers and control the memory accesses mode, this is implementation dependent, not standartized. The MMU can only interffer on cpu memory access only if the flag **VM** is enabled.

# Periphericals and the SLI

The cpu can drectly comunicates to others periphericals only through the BUS PCI, aka: Peripherical Communication Interface, and vice versa. Otherwise, the devices will may depend of memory mapping to do communications between them.

Any peripherical is a running state independent of any others, so, to communicate to them, are provided the instructions **IN** and **OUT** to send a dword message, and wait for input changing with the instruction **INUP** which updates the flag **OF** in case of changing, or wait for any interruption with instruction **WAIT**, or **WAITI** to wait a interruption from a specific port.

The **SLI**, aka: Standard Limp Interface, is a way to ensure a communication to any peripherical. Absolutely, all periphericals must follow this standard (is not called standard for nothing). His principles are:

* All periphericals must be initialized, and start requesting for interruption;
* They must have a specie of lobby, the cpu send a message data wich tells him what to do, they receive the data and do its operation;
* The reserved message is **0x10** for type requesting, the reserved types codes are:
  + **0** => means a not connected peripherical;
  + **1** => For Standard output;
  + **2** => For Hard disk;
  + **3** => For Flash disk;
  + **4** => For Keyboard input;
  + **5** => For Mouse input;
  + **6** => For Input controller;
  + **7** => For Monitor (video graphic display);
  + **8** => For Sound card;
  + **9** => For Network card;
* The reserved message **0x11** is for brand identification;
* The reserved message **0x12** is for branch device type;
* The reserved message **0x13** is for version number;

## The Assembler

# Instructions Formats

Has 6 formats of instructions, each one determine the type of operands to deal with, the way to deal with, sub-operations and also, extra data fetching. The following are the formats and furthers details:

**IR** => Immediate to Register

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | | 20 | 19 | 18 | | 17 | 16 | 15 | | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Base Opcode | | | | | | S | M | IM | | | RegD | | | | RegB | | | | 16-Immediate | | | | | | | | | | | | | | | |

**AMI** => Addressing Mode Instruction

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Base Opcode | | | | | | S | M | AdrM | | | | DSize | | F | RegD | | | RegI | | | RegB | | | 8-Immediate | | | | | | | |

**SI** => Sub-Functional Instruction

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | | 16 | | 15 | 14 | 13 | | 12 | 11 | 10 | | 9 | 8 | 7 | | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Base Opcode | | | | | | S | M | Func | | | | | | | F | | RegD | | | | RegP | | | | RegB | | | | 8-Immediate | | | | | | | | |

**ADI** => Address Program Dealer Instruction

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | | 17 | 16 | 15 | | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Base Opcode | | | | | | S | M | Condition | | | | | | RegO | | | | 16-Immediate | | | | | | | | | | | | | | | | |

**CDI** => Conditionally Dealer Instruction

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Base Opcode | | | | | | S | M | Condition | | | | | O | F | RegD | | | RegO | | | RegB | | | 8-Immediate | | | | | | | |

**JL** => Jump Long

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Base Opcode | | | | | | 26-Immediate | | | | | | | | | | | | | | | | | | | | | | | | | |

Each one of theses arguments has a specific behavior in processing of instructions:

* **Base Opcode** => A base code for instruction decoding. This is also used for format distinguish in instruction decoding, wich means, if the base opcode is 0x00, every sub-instruction of this base is of the same format.
* **S (Selector)** => This bit interchanges between the register set to be dealed in current instruction. Obs: In format **AMI**, do not affect the register **regi** or **regb** (will use the standard set instead), if **AdrM** specify instruction to retrieve data directly from **regb**, it will be affected by this flag.
* **M (Mod)** => The modifier of instruction, also can branch in sub-operations.
* **IM** => Specify the mode of immediate in IR Format:
  + 0 – 16-bit data with no shifting
  + 1 – 16-bit data with shifting to upper word
  + 2 – 16-bit data with sign extension
  + 3 – 32-bit extra data fetched
* **Func** => Branchs in sub-operations for the same base.
* **AdrM** => The mode of adressing. The data fetched for instruction is obtained by the mode specified in this argument.
* **DSize** => The size of memory data access (and the immediate shift factor).
* **Condition** => The condition for instruction execution.
* **O (Operand Set)** => Does the same as the Selector bit, but applies to operand register in format **CDI**. Obs: in format **ADI** the Selector bit will take this behavior.
* **F (Fetch)** => Fetch flag, determines if instruction fetches the next data as immediate.
* **RegD** => Usually, the destination register in instruction.
* **RegB** => Usually, the base register for some operations in instruction.
* **RegI** => A Intermediate register for base address indexing.
* **RegP** => A parity register, for operations with pairs registers.
* **RegO** => Operand register, used in conditions testing.
* **Immediate** => A in instruction directly data.

The instruction may have the correctly format for some instructions base and sub-functions. For example, actually, a instruction with base 0x3F is invalid, cause does not exists. Likely, a instruction with base 0x03 and sub-function 0x30, it does not exists, when trying to execute, will trigger a invalid instruction interruption.

# Instructions Encoding

The instructions follows the encoding:

Mnemonic + descriptors + arguments

Mnemonic is a specifc name for instruction base, modify and sub-function, for Example: **ADD**, has the base 0x10 and mod 0x0.

The assembly code for Limp is case-sensitive, wich means, all mnemonics must be low case, the descriptors and registers names, otherwise, will be refering to other symbol name.

A instruction can have from zero up to 2 descriptors, but they may vary by the the instruction format itself. Look below:

* **Descriptor: .im** => This descriptor modifies the argument **IM**. The compatible format is **IR**, and has the following formats:
  + **.w/.uw**:0 => The immediate is of type unsigned word (inside instruction) (Default);
  + **.hw**:1 => The immediate is word shifted to upper (inside instruction);
  + **.sw**:2 => The immediate is a word with sign extension to 32-bit (inside instruction);
  + **.d**:3 => The immediate is a dword (extra fetch immediate data)
* **Descriptor: .f** => This descriptor modifies the argument **F**. The compatible formats are **AMI**, **SI** and **CDI**, and has the following formats:
  + **.b**:0 => Specify a byte immediate type (inside instruction) (Default);
  + **.d**:1 => Specify a dword immediate type (extra fetch immediate data).
* **Descriptor: .cond** => This descriptor modifies the argument **Condition**. The compatibles formats are **ADI** and **CDI**, and has the followinf formats:
  + **.aw**:0 => Will always be executed, no test is did;
  + **.eq**:1 => If equal, test flags: *ZF*;
  + **.ne**:2 => If not equal, test flags: *!ZF*;
  + **.lt**:3 => If less than, test flags: *(VF!=NF)&&!ZF*;
  + **.gt**:4 => If great than, test flags: *(VF==NF)&&!ZF*;
  + **.le**:5 => If less or equal, test flags: *(VF!=NF)||!ZF*;
  + **.ge**:6 => If great or equal, test flags: *(VF==NF)||!ZF*;
  + **.blw**:7 => If is below, test flags: *BF&&!ZF*;
  + **.ab**:8 => If is above, test flags: *!BF&&!ZF*;
  + **.be**:9 => If is below or equal, test flags: *BF||ZF*;
  + **.ae**:10 => If is above or equal, test flags: *!BF||ZF*;
  + **.ez**:11 => If is zero, test flags: *ZF&&!CF&&!BF*;
  + **.nz**:12 => If is not zero, test flags: *!ZF&&!CF&&!BF*;
  + **.gz**:13 => If is great than zero, test flags: *!ZF&&!NF*;
  + **.lz**:14 => If is less than zero, test flags: *!ZF&&NF*;
  + **.oez****<RegO>**:15 => If operator is equal zero, test: *RegO==0*;
  + **.onz<RegO>**:16 => If operator is not equal zero, test: *RegO!=0*;
  + **.ogz<RegO>**:17 => If operator is great than zero, test: *RegO>0*;
  + **.olz<RegO>**:18 => If operator is less than zero, test: *RegO<0*;
  + **.oed<RegO>**:19 => If operator is equal to reg **EDX**, test: *RegO==****EDX***;
  + **.ond<RegO>**:20 => If operator is not equal to reg **EDX**, test: *RegO!=****EDX***;
  + **.old<RegO>**:21 => If operator is less than reg **EDX**, test: *RegO<****EDX****;*
  + **.ogd<RegO>**:22 => If operator is great than reg **EDX**, test: *RegO>****EDX****;*
  + **.oea<RegO>**:23 => If operator is equal to reg **EAX**, test: *RegO==****EAX***;
  + **.ona<RegO>**:24 => If operator is not equal to reg **EAX**, test: *RegO!=****EAX***;
  + **.ov**:25 => If flag overflow is set, test flag: *V*;
  + **.sc**:26 => If flag carry is set, test flag: *C*;
  + **.cc**:27 => If flag carry is clear, test flag: *!C*;
  + **.sb**:28 => If flag borrow is set, test flag: *B*;
  + **.cb**:29 => If flag borrow is clear, test flag: *!B*;
  + **.so**:30 => If flag odd is set, test flag: *O*;
  + **.co**:31 => If flag odd is clear, test flag: *!O*;

A instruction can have from zero up to 4 arguments, accordling to its defined format. A argument can be a **Reg**, **Imm** or **Amd** type.

A **Reg** argument can be only one of eight Limp registers: **EAX**, **EDX**, **ECX**, **EBX**, **EFP**, **ESP**, **ESS** and **ESD**.

A **Imm** argument can be a lonely literal value, or a inline constant expression.

A **Amd** argument can be a lonely **Reg** value (AdrM=1), a **Imm** expression (AdrM=0) or any of memory access mode combination, each is, as its AdrM code:

* **Indirect**:2 => [ Imm ]#DS;
* **Pointer**:3 => [ RegB ]#DS;
* **Pointer Immediate Indexed**:4 => [ RegB + Imm ]#DS;
* **Pointer Indexed**:5 => [ RegB + RegI ]#DS;
* **Pointer Dynamic**:6 => [ RegB + RegI + Imm ]#DS;
* **Pointer Element**:7 => [ RegB + RegI\*Imm]#DS;
* **Pointer Pre-Increment**:8 => [ ++RegB ]#DS;
* **Pointer Pre-Decrement**:9 => [ --RegB ]#DS;
* **Pointer Pos-Increment**:10 => [ RegB++ ]#DS;
* **Pointer Pos-Decrement**:11 => [ RegB-- ]#DS;
* **Pointer Indexed Pos-Increment**:12 => [ RegB + (RegI++) ]#DS;
* **Pointer Indexed Pos-Decrement**:13 => [ RegB + (RegI--) ]#DS;
* **Pointer Dynamic Pos-Increment**:14 => [ RegB + (RegI++) + Imm ]#DS;
* **Pointer Dynamic Pos-Decrement**:15 => [ RegB + (RegI--) + Imm ]#DS;

The DS posfix in **Amd** is a name of memory size access: **Byte**, **Word**, **Dword** and a **Qword**, but can be omitted.

# Immediate Expressions

Any Immediate expression is a constant inline expression, it may involve some binary and unary operations, i.e:

* **Binary**: (Base arithmetical operations) +, -, \*, /, (Module) %, (Bitwise operations) &, |, ^, >>, <<, (Logical operations) &&, ||, ==, !=, >, <, >=, <=.
* **Unary**: (Signal) +, -, (Others too) ~, !

The allowed values operands are only the **literals**, and **defined symbols**. Arithmetic can involve parenthesis for expression isolation. Values can be cast to others size types too, using the symbol ‘**#’** for unsigned cast, and ‘**:**’ for signed cast, followed by name of memory size (i.e: **Byte**, **Word**, **Dword** or **Qword**).

Has a reserved symbol ‘**@**’, using lonely, is equal to current program address, using followed by a symbol name or literal, corresponds to a relative offset to it, its useful for relative jumps to labels (Ex: ‘@label’).

# Labels

Labels are symbols names for addresses in program, a very help utility for almost all the situation in assembly programming.

# Preprocessors

Any others features beside the instruction encoding and labels definition are preprocessors. Preprocessors are defined by a dot followed by a valid preprocessor name (i.e: ‘.include’); all preprocessors available in assembler are:

* **adr** *addr:Imm* => Sets the current program address
* **align** => Aligns the current program address in instructions bound (4-bytes)
* **bin** *path:Str* => Import a external binary file at current position in output file
* **break** => Exits the current file
* **const** *name:Sym value:Imm*=> Defines a constant symbol in 2nd Phase
* **d8, d16 and d32** *[, value:Imm]*=> Defines a sequence of raw data at current position in output file
* **define** *name:Sym code:Raw*=> Defines a segment code with a symbol name
* **defonce** *once:Str* => Define a once control name
* **include** *path:Str* => Include a external assembly file
* **macro** *name:Sym [, argName:Sym[‘:’ type:Type]]* **/.endmacro**=> Defines a macro for parametrized code segment reuse
* **notonce** *once:Str* => Keeps processing file if once control is defined
* **once** *[once:Str]* => Keeps processing file if once contron is not defined
* **predef** *name:Sym value:Imm* => Defines a constant symbol in 1st Phase
* **scope** *tag:Sym* **/.endscope** => Enters a enclosed scope
* **text** *str:Str* => Defines a raw text to output file

# Scopes

A very helpful feature for enclosing code symbols. They are defined using the preprocessors **scope** and **endscope** to delimitate code. Defining a name for scope is optional, but, if you do, will be same as declarating a label in outer scope.

Once the program is inside a scope, every symbol defined inside it won’t be accessible outside, its his purpose, isolating symbols for enclosed routines. You can access outside symbols, and, once is not defined in same scope, you can define a symbol with same name as other, but only the closest symbol with duplicated name will be used.

You can defined stacked scopes, its means, can go deeper in sub-scoping and symbol restriction definition.

# Macro

Also, other very useful utility for code reuse, you can define a macro using the preprocessors **macro** and **endmacro** to delimitate code. A macro must have a name, but arguments are optional.

One usefulness of macros are the scope restriction, inside a macro, every name and symbol is restricted, which means you will be able to define labels for casuals routines, and symbols naming.

Macros can be specified with arguments, but, once a macro is called, the arguments must match. Arguments are comma separated (like in instruction), either in declaration and in call. Arguments can be typified with **Reg**, **Imm** or **Amd** arguments types, but, in this cases, types must match on calling. Arguments macros are by him self enclosed, wich means, only exists in scope of macros, if you call for other macro inside a macro, the arguments of previous macros won’t be available.

You can define sub-macros, inside macro declaration, you can define a scope restricted macro for most deeper sub-routines, they will be sequencially expanded once are called.

# Once Processing

A file can be defined to be processed only one time by including the preprocessor **once** at the header, you can specify a once tagging by using a string as argument, otherwise, the path of file will be used.

Once Control is a great feature for avoiding declarations duplicates, you can also ommit a file processing by other using the same tag for both.

You can define a once control by the preprocessor **defonce**, which require a string argument for tag definition, wich means, any other file using this tag at **once** preprocessor will be skipped.

Also, you can use other preprocessor called **notonce**, it’s the inverse of **once**, the file will be processed only if the control tag is defined (and is required to especify). Once a time you make a use of the preprocessor **once**, the tag will be defined.

## Instruction Set

[Checks the docs.html]