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 clock speed goal is to achieve 4 mips, but *is a concept in work*. 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 Mips are equal to 1,048,576 or 2^{20} instructions for second.

Registers and Fast storages

The system contains 8 main regsters for instructions operations, all 32-bit, the registers are:

- **EAX** => The accumulator of system, makes pair with **edx**;
- **EDX** => The data holder and comparator, makes pair with **eax**;
- **ECX** => The counter and indexer, makes pair with **ebx**;
- **EBX** => The base register and operand, makes pair with **ecx**;
- **EFP** => The pointer for program frame, makes pair with **esp**;
- **ESP** => The stack pointer of system, makes pair with **efp**;

- **ESS** => The stream source index, makes pair with **esd**;
- **ESD** => The stream destination, makes pair with **ess**;

Besides the main registers, the system can have a register pool of registers which can be accessed by the special instructions **mvfr** and **mvtr**. 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.
- **EST** => System status register, stores the control flags and results statuses.
- **LPC** => Last program counter, used for relative jumps.
- **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.

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 **EST** and **EPC** are pushed in this order, 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** and **EST** 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	
10	Debugger Interruption
11	
12	
13	
14	
15	
16 -> 127	Software Reserved
127 -> 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 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

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="6">Base Opcode</td><td colspan="2">Mod</td><td colspan="2">IM</td><td colspan="3">RegD</td><td colspan="3">RegB</td><td colspan="16">16-Immediate</td></tr>
</table>

AMI => Addressing Mode Instruction

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="6">Base Opcode</td><td colspan="2">Mod</td><td colspan="4">AdrM</td><td colspan="2">DSize</td><td>F</td><td colspan="3">RegD</td><td colspan="3">RegI</td><td colspan="3">RegB</td><td colspan="8">8-Immediate</td></tr>
</table>

SI => Sub-Functional Instruction

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="6">Base Opcode</td><td colspan="2">Mod</td><td colspan="6">Func</td><td>F</td><td colspan="3">RegD</td><td colspan="3">RegP</td><td colspan="3">RegB</td><td colspan="8">8-Immediate</td></tr>
</table>

ADI => Address Program Dealer Instruction

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="6">Base Opcode</td><td colspan="2">Mod</td><td colspan="5">Condition</td><td colspan="3">RegO</td><td colspan="16">16-Immediate</td></tr>
</table>

CDI => Conditionally Dealer Instruction

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="6">Base Opcode</td><td colspan="2">Mod</td><td colspan="5">Condition</td><td>1</td><td>F</td><td colspan="3">RegD</td><td colspan="3">RegO</td><td colspan="3">RegB</td><td colspan="8">8-Immediate</td></tr>
</table>

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.
- **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.
- **F** => 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 2^{nd} 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 1^{st} 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

aba

SI Format: Protected Mode

Opc: 0x14   Func: 9

aba[.f] regd, regb

Adjusts the value in regb from ascii to binary and stores in regd

---

adc

AMI Format: Protected Mode

Opc: 0x10   Mode: 1

adc[.f] regd, $amd

Adds a value data with carry to register

add

AMI Format: Protected Mode

Opc: 0x10   Mode: 0

add[.f] regd, $amd

Adds a value data to register

and

AMI Format: Protected Mode

Opc: 0x18   Mode: 0

and[.f] regd, $amd

Do and bitwise value data to register

---

ba

ADI Format: Protected Mode

Opc: 0x20   Mode: 1

ba[.cond] imm16<2

Branchs to a absolute address position

---

baa

SI Format: Protected Mode

Opc: 0x14   Func: 8

baa[.f] regd, regb

Adjusts the value in regb from binary to ascii and stores in regd

---

bit

AMI Format: Protected Mode

Opc: 0x1a   Mode: 1

bit[.f] regd, $amd

Do and bitwise between regd and data, doing a bit test, without store the result

bl

JL Format: Protected Mode

Opc: 0x25

bl imm<2

Do a long branch inside the current 256 MB Bank

blp

JL Format: #Super Mode

Opc: 0x27

blp imm<2

Do a long branch inside the current 256 MB Bank in protected mode

---

br

ADI Format: Protected Mode

Opc: 0x20   Mode: 3

br[.cond] imm16<2

Branchs by a relative offset

---

bra

CDI Format: Protected Mode

Opc: 0x22   Mode: 1

bra[.cond][.f] regp

Branchs to a absolute address by regp

---

brap

CDI Format: #Super Mode

Opc: 0x22   Mode: 3

brap[.cond][.f] regp

Branchs to a absolute address by regp in protected mode

brr

CDI Format: Protected Mode

Opc: 0x23   Mode: 1

brr[.cond][.f] regp

Branchs by a relative offset by regp

clb

SI Format: Protected Mode

Opc: 0x38   Func: 4

clb[.f]

Clears the flag BF

---

clc

SI Format: Protected Mode

Opc: 0x38   Func: 1

clc[.f]

Clears the flag CF

---

cln

SI Format: Protected Mode

Opc: 0x38   Func: 13

cln[.f]

Clears the flag NF

---

clo

SI Format: Protected Mode

Opc: 0x38   Func: 16

clo[.f]

Clears the flag OF

clrb

SI Format: Protected Mode

Opc: 0x19   Func: 2

clrb[.f] regd, imm8

Clears the bit of regd at offset of imm

clv

SI Format: Protected Mode

Opc: 0x38   Func: 7

clv[.f]

Clears the flag VF

---

clz

SI Format: Protected Mode

Opc: 0x38   Func: 10

clz[.f]

Clears the flag ZF

---

cmp

AMI Format: Protected Mode

Opc: 0x15   Mode: 0

cmp[.f] regd, $amd

Do a subtraction between a value data from register without saving the result, doing a comparation

---

cp0chkst

SI Format: #Super Mode

Opc: 0x8   Func: 6   Mode: 0

cp0chkst[.f] regd

Reads the status of coprocessor 0 and stores in regd. Note: 0 = Not present; 1 = Enabled; -1 = Disabled

---

cp0cmd

SI Format: #Super Mode

Opc: 0x8   Func: 7   Mode: 0

cp0cmd[.f] regb

Sends a command encoded in regb to coprocessor 0

---

cp0di

SI Format: #Super Mode

Opc: 0x8   Func: 4   Mode: 0

cp0di[.f]

Disables coprocessor 0

---

cp0en

SI Format: #Super Mode

Opc: 0x8 Func: 5 Mode: 0

cp0en[.f]

Enables coprocessor 0

---

cp0rr

SI Format: #Super Mode

Opc: 0x8   Func: 2   Mode: 0

cp0rr[.f] regd, regp

Reads register by regp from coprocessor 0 to register regd

---

cp0wr

SI Format: #Super Mode

Opc: 0x8   Func: 3   Mode: 0

cp0wr[.f] regp, regb

Writes regb value to register by regp to coprocessor 0

cp1chkst

SI Format: #Super Mode

Opc: 0x8 Func: 6 Mode: 1

cp1chkst[.f] regd

Reads the status of coprocessor 1 and stores in regd. Note: 0 = Not present; 1 = Enabled; -1 = Disabled

cp1cmd

SI Format: #Super Mode

Opc: 0x8 Func: 7 Mode: 1

cp1cmd[.f] regb

Sends a command encoded in regb to coprocessor 1

---

cp1di

SI Format: #Super Mode

Opc: 0x8   Func: 4   Mode: 1

cp1di[.f]

Disables coprocessor 1

---

cp1en

SI Format: #Super Mode

Opc: 0x8   Func: 5   Mode: 1

cp1en[.f]

Enables coprocessor 1

---

cp1rr

SI Format: #Super Mode

Opc: 0x8   Func: 2   Mode: 1

cp1rr[.f] regd, regp

Reads register by regp from coprocessor 1 to register regd

cp1wr

SI Format: #Super Mode

Opc: 0x8   Func: 3   Mode: 1

cp1wr[.f] regp, regb

Writes regb value to register by regp to coprocessor 1

cp2chkst

SI Format: #Super Mode

Opc: 0x8   Func: 6   Mode: 2

cp2chkst[.f] regd

Reads the status of coprocessor 2 and stores in regd. Note: 0 = Not present; 1 = Enabled; -1 = Disabled

---

cp2cmd

SI Format: #Super Mode

Opc: 0x8   Func: 7   Mode: 2

cp2cmd[.f] regb

Sends a command encoded in regb to coprocessor 2

---

cp2di

SI Format: #Super Mode

Opc: 0x8   Func: 4   Mode: 2

cp2di[.f]

Disables coprocessor 2

---

cp2en

SI Format: #Super Mode

Opc: 0x8   Func: 5   Mode: 2

cp2en[.f]

Enables coprocessor 2

cp2rr

SI Format: #Super Mode

Opc: 0x8   Func: 2   Mode: 2

cp2rr[.f] regd, regp

Reads register by regp from coprocessor 2 to register regd

cp2wr

SI Format: #Super Mode

Opc: 0x8   Func: 3   Mode: 2

cp2wr[.f] regp, regb

Writes regb value to register by regp to coprocessor 2

cp3chkst

SI Format: #Super Mode

Opc: 0x8   Func: 6   Mode: 3

cp3chkst[.f] regd

Reads the status of coprocessor 3 and stores in regd. Note: 0 = Not present; 1 = Enabled; -1 = Disabled

cp3cmd

SI Format: #Super Mode

Opc: 0x8   Func: 7   Mode: 3

cp3cmd[.f] regb

Sends a command encoded in regb to coprocessor 3

---

cp3di

SI Format: #Super Mode

Opc: 0x8   Func: 4   Mode: 3

cp3di[.f]

Disables coprocessor 3

---

cp3en

SI Format: #Super Mode

Opc: 0x8   Func: 5   Mode: 3

cp3en[.f]

Enables coprocessor 3

---

cp3rr

SI Format: #Super Mode

Opc: 0x8   Func: 2   Mode: 3

cp3rr[.f] regd, regp

Reads register by regp from coprocessor 3 to register regd

---

cp3wr

SI Format: #Super Mode

Opc: 0x8   Func: 3   Mode: 3

cp3wr[.f] regp, regb

Writes regb value to register by regp to coprocessor 3

---

cpb

AMI Format: Protected Mode

Opc: 0x15   Mode: 1

cpb[.f] regd, $amd

Do a subtraction with borrow between a value data from register without saving the result, doing a comparation

---

cvbd

IR Format: Protected Mode

Opc: 0x31   Mode: 1

cvbd[.im] regd, regb

Converts a byte data to sign extended dword to register

---

cvbw

IR Format: Protected Mode

Opc: 0x31   Mode: 0

cvbw[.im] regd, regb

Converts a byte data to sign extended word to register

---

cvwd

IR Format: Protected Mode

Opc: 0x31   Mode: 2

cvwd[.im] regd, regb

Converts a word data to sign extended dword to register

---

cvwdi

IR Format: Protected Mode

Opc: 0x31   Mode: 3

cvwdi[.im] regd, regb

Converts a word immediate data to sign extended dword to register

---

dec

SI Format: Protected Mode

Opc: 0x14   Func: 1

dec[.f] regd

Do a decrement in regd

---

div

AMI Format: Protected Mode

Opc: 0x11   Mode: 2

div[.f] regd, $amd

Divides a value data to register

dsbi

SI Format: #Super Mode

Opc: 0x38   Func: 19

dsbi[.f]

Disables interruption

dsbv

SI Format: #Super Mode

Opc: 0x38   Func: 21

dsbv[.f]

Disables virtual mode

---

enbi

SI Format: #Super Mode

Opc: 0x38   Func: 18

enbi[.f]

Enables interruption

---

enbv

SI Format: #Super Mode

Opc: 0x38   Func: 20

enbv[.f]

Enables virtual mode

---

enter

SI Format: Protected Mode

Opc: 0x21   Func: 2   Mode: 0

enter[.f] imm

Reserves a sized space stack for current procedure and stores offset in EFP

enterv

SI Format: Protected Mode

Opc: 0x21 Func: 2 Mode: 1

enterv[.f] regb

Reserves a sized variable space stack for current procedure and stores offset in EFP

fabs

SI Format: Protected Mode

Opc: 0x1c Func: 12

fabs[.f] regd

In Floating-Point format, sets regd to its absolute value

---

facos

SI Format: Protected Mode

Opc: 0x1c   Func: 26

facos[.f] regd, regb

In Floating-Point format, do arcosine of regb and stores to regd

---

fadc

SI Format: Protected Mode

Opc: 0x1c   Func: 1

fadc[.f] regd, regb

In Floating-Point format, adds regb to regd with carry

---

fadd

SI Format: Protected Mode

Opc: 0x1c   Func: 0

fadd[.f] regd, regb

In Floating-Point format, adds regb to regd

fasin

SI Format: Protected Mode

Opc: 0x1c   Func: 25

fasin[.f] regd, regb

In Floating-Point format, do arcsine of regb and stores to regd

fatan

SI Format: Protected Mode

Opc: 0x1c   Func: 27

fatan[.f] regd, regb

In Floating-Point format, do arctangent of regb and stores to regd

---

fatan2

SI Format: Protected Mode

Opc: 0x1c   Func: 28

fatan2[.f] regd, regb, regp

In Floating-Point format, do arctangent in y of regb and x of regp to regd

---

fcbrt

SI Format: Protected Mode

Opc: 0x1c   Func: 16

fcbrt[.f] regd, regb

In Floating-Point format, do cubic root of regb and stores to regd

---

fcil

SI Format: Protected Mode

Opc: 0x1c   Func: 31

fcil[.f] regd, regb

In Floating-Point format, rounds regb to ceil and stores to regd

fcint

SI Format: Protected Mode

Opc: 0x1c   Func: 18

fcint[.f] regd, regb

In Floating-Point format, do a ceil of regd to int format and stores to regd

fcos

SI Format: Protected Mode

Opc: 0x1c   Func: 23

fcos[.f] regd, regb

In Floating-Point format, do cosine of regb and stores to regd

---

fcvb

SI Format: Protected Mode

Opc: 0x1c   Func: 59

fcvb[.f] regd, regb

Convert from Floating-Point format regb to byte regd

---

fcvd

SI Format: Protected Mode

Opc: 0x1c   Func: 61

fcvd[.f] regd, regb

Convert from Floating-Point format regb to dword regd

---

fcvub

SI Format: Protected Mode

Opc: 0x1c   Func: 56

fcvub[.f] regd, regb

Convert from Floating-Point format regb to unsigned byte regd

fcvud

SI Format: Protected Mode

Opc: 0x1c   Func: 58

fcvud[.f] regd, regb

Convert from Floating-Point format regb to unsigned dword regd

fcvuw

SI Format: Protected Mode

Opc: 0x1c   Func: 57

fcvuw[.f] regd, regb

Convert from Floating-Point format regb to unsigned word regd

---

fcvw

SI Format: Protected Mode

Opc: 0x1c   Func: 60

fcvw[.f] regd, regb

Convert from Floating-Point format regb to word regd

---

fdcv

SI Format: Protected Mode

Opc: 0x1c   Func: 63

fdcv[.f] regd, regb

Convert from dword regb to Floating-Point format regd

---

fdiv

SI Format: Protected Mode

Opc: 0x1c   Func: 9

fdiv[.f] regd, regb

In Floating-Point format, divides regb by regd

---

fflr

SI Format: Protected Mode

Opc: 0x1c   Func: 29

fflr[.f] regd, regb

In Floating-Point format, rounds regb to floor and stores to regd

---

flog10

SI Format: Protected Mode

Opc: 0x1c   Func: 21

flog10[.f] regd, regb

In Floating-Point format, do a log in base 10 of regb and stores to regd

---

flog2

SI Format: Protected Mode

Opc: 0x1c Func: 20

flog2[.f] regd, regb

In Floating-Point format, do a log in base 2 of regb and stores to regd

---

fmadc

SI Format: Protected Mode

Opc: 0x1c   Func: 6

fmadc[.f] regd, regb, regp

In Floating-Point format, multiply regb to regd and adds regp with carry

---

fmadd

SI Format: Protected Mode

Opc: 0x1c   Func: 5

fmadd[.f] regd, regb, regp

In Floating-Point format, multiply regb to regd and adds regp

fmod

SI Format: Protected Mode

Opc: 0x1c   Func: 10

fmod[.f] regd, regb

In Floating-Point format, modulates regb by regd

fmsbb

SI Format: Protected Mode

Opc: 0x1c   Func: 8

fmsbb[.f] regd, regb, regp

In Floating-Point format, multiply regb to regd and subtracts regp with borrow

---

fmsub

SI Format: Protected Mode

Opc: 0x1c   Func: 7

fmsub[.f] regd, regb, regp

In Floating-Point format, multiply regb to regd and subtracts regp

---

fmul

SI Format: Protected Mode

Opc: 0x1c   Func: 4

fmul[.f] regd, regb

In Floating-Point format, multiply regb to regd

---

fneg

SI Format: Protected Mode

Opc: 0x1c   Func: 13

fneg[.f] regd

In Floating-Point format, do 0-regd and stores to regd

fpow

SI Format: Protected Mode

Opc: 0x1c   Func: 14

fpow[.f] regd, regb

In Floating-Point format, do pow of regd by regb

fqtrt

SI Format: Protected Mode

Opc: 0x1c   Func: 17

fqtrt[.f] regd, regb

In Floating-Point format, do 4th root of regb and stores to regd

---

frnd

SI Format: Protected Mode

Opc: 0x1c Func: 30

frnd[.f] regd, regb

In Floating-Point format, rounds regb and stores to regd

---

fsbb

SI Format: Protected Mode

Opc: 0x1c   Func: 3

fsbb[.f] regd, regb

In Floating-Point format, subtracts regb from regd with borrow

---

fscale

SI Format: Protected Mode

Opc: 0x1c   Func: 11

fscale[.f] regd, regb

In Floating-Point format, scales regb in power of two by regb in dword format

fsin

SI Format: Protected Mode

Opc: 0x1c   Func: 22

fsin[.f] regd, regb

In Floating-Point format, do sine of regb and stores to regd

fsqrt

SI Format: Protected Mode

Opc: 0x1c   Func: 15

fsqrt[.f] regd, regb

In Floating-Point format, do square root of regb and stores to regd

---

fsteq

SI Format: Protected Mode

Opc: 0x1d Func: 0

fsteq[.f] regd, imm8, regb, regp

Stores a immediate value to register if in Floating-point regb is equal to regp

---

fstez

SI Format: Protected Mode

Opc: 0x1d   Func: 12

fstez[.f] regd, regp, regb

Stores regp value to register if in Floating-point regb is equal to zero

---

fstezi

SI Format: Protected Mode

Opc: 0x1d   Func: 6

fstezi[.f] regd, imm8, regb

Stores a immediate value to register if in Floating-point regb is equal to zero

fstge

SI Format: Protected Mode

Opc: 0x1d   Func: 5

fstge[.f] regd, imm8, regb, regp

Stores a immediate value to register if in Floating-point regb is greater or equal than regp

fstgez

SI Format: Protected Mode

Opc: 0x1d   Func: 17

fstgez[.f] regd, regp, regb

Stores regp value to register if in Floating-point regb is greater or equal than zero

---

fstgezi

SI Format: Protected Mode

Opc: 0x1d Func: 11

fstgezi[.f] regd, imm8, regb

Stores a immediate value to register if in Floating-point regb is greater or equal than zero

---

fstgt

SI Format: Protected Mode

Opc: 0x1d   Func: 3

fstgt[.f] regd, imm8, regb, regp

Stores a immediate value to register if in Floating-point regb is greater than regp

---

fstgtz

SI Format: Protected Mode

Opc: 0x1d   Func: 15

fstgtz[.f] regd, regp, regb

Stores regp value to register if in Floating-point regb is greater than zero

fstgtzi

SI Format: Protected Mode

Opc: 0x1d   Func: 9

fstgtzi[.f] regd, imm8, regb

Stores a immediate value to register if in
Floating-point regb is greater than zero

fstle

SI Format: Protected Mode

Opc: 0x1d   Func: 4

fstle[.f] regd, imm8, regb, regp

Stores a immediate value to register if in Floating-point regb is less or equal than regp

---

fstlez

SI Format: Protected Mode

Opc: 0x1d   Func: 16

fstlez[.f] regd, regp, regb

Stores regp value to register if in Floating-point regb is less or equal than zero

---

fstlezi

SI Format: Protected Mode

Opc: 0x1d   Func: 10

fstlezi[.f] regd, imm8, regb

Stores a immediate value to register if in Floating-point regb is less or equal than zero

---

fstlt

SI Format: Protected Mode

Opc: 0x1d   Func: 2

fstlt[.f] regd, imm8, regb, regp

Stores a immediate value to register if in Floating-point regb is less than regp

fstltz

SI Format: Protected Mode

Opc: 0x1d   Func: 14

fstltz[.f] regd, regp, regb

Stores regp value to register if in Floating-point regb is less than zero

fstltzi

SI Format: Protected Mode

Opc: 0x1d   Func: 8

fstltzi[.f] regd, imm8, regb

Stores a immediate value to register if in Floating-point regb is less than zero

---

fstne

SI Format: Protected Mode

Opc: 0x1d   Func: 1

fstne[.f] regd, imm8, regb, regp

Stores a immediate value to register if in Floating-point regb is not equal to regp

---

fstnz

SI Format: Protected Mode

Opc: 0x1d   Func: 13

fstnz[.f] regd, regp, regb

Stores regp value to register if in Floating-point regb is not equal to zero

---

fstnzi

SI Format: Protected Mode

Opc: 0x1d   Func: 7

fstnzi[.f] regd, imm8, regb

Stores a immediate value to register if in Floating-point regb is not equal to zero

fsub

SI Format: Protected Mode

Opc: 0x1c   Func: 2

fsub[.f] regd, regb

In Floating-Point format, subtracts regb from regd

ftan

SI Format: Protected Mode

Opc: 0x1c   Func: 24

ftan[.f] regd, regb

In Floating-Point format, do tangent of regb and stores to regd

fudcv

SI Format: Protected Mode

Opc: 0x1c Func: 62

fudcv[.f] regd, regb

Convert from unsigned dword regb to Floating-Point format regd

fxam

SI Format: Protected Mode

Opc: 0x1c   Func: 19

fxam[.f] regd, regb

In Floating-Point format, examinates value of regb and store status in regd. Note: 0 = Zero; 1 = Normal; 2 = SubNormal; 3 = NaN; 4 = Infinity

---

halt

SI Format: Protected Mode

Opc: 0x1   Func: 0

halt[.f]

Halts the system execution

hmul

AMI Format: Protected Mode

Opc: 0x11   Mode: 1

hmul[.f] regd, $amd

Multiplies a value data with register, and store the higher dword part to register

in

SI Format: #Super Mode

Opc: 0x8   Func: 0

in[.f] regd, regp

Get input from external device in port by regp to regd

---

inc

SI Format: Protected Mode

Opc: 0x14   Func: 0

inc[.f] regd

Do a increment in regd

---

int

SI Format: Protected Mode

Opc: 0x21   Func: 4

int[.f] imm8

Calls for system interruption

---

inup

SI Format: #Super Mode

Opc: 0x8   Func: 8   Mode: 0

inup[.f] regd

Verify if input from port in regd is updated, if then, sets the OF flag, otherwise clears it

---

inus

SI Format: #Super Mode

Opc: 0x8 Func: 8 Mode: 1

inus[.f] regd

Verify if input from port in regd is updated, if then, sets the OF flag, otherwise clears it, and resets the check

---

iret

SI Format: #Super Mode

Opc: 0x21 Func: 1

iret[.f]

Returns from a interruption

---

ja

ADI Format: Protected Mode

Opc: 0x20   Mode: 0

ja[.cond] imm16<2

Jumps to a absolute address position

---

jl

JL Format: Protected Mode

Opc: 0x24

jl imm<2

Do a long jump inside the current 256 MB Bank

---

jlp

JL Format: #Super Mode

Opc: 0x26

jlp imm<2

Do a long jump inside the current 256 MB Bank in protected mode

jr

ADI Format: Protected Mode

Opc: 0x20 Mode: 2

jr[.cond] imm16<2

Jumps by a relative offset

jra

CDI Format: Protected Mode

Opc: 0x22 Mode: 0

jra[.cond][.f] regb

Jumps to a absolute address by regp

---

jrap

CDI Format: #Super Mode

Opc: 0x22   Mode: 2

jrap[.cond][.f] regp

Jumps to a absolute address by regp in protected mode

---

jrr

CDI Format: Protected Mode

Opc: 0x23   Mode: 0

jrr[.cond][.f] regb

Jumps by a relative offset by regp

---

ldiv

SI Format: Protected Mode

Opc: 0x14   Func: 7

ldiv[.f] regd, regb

Divides regd by regb, but stores at regd the under integer (decimal part) value

ldmb

AMI Format: Protected Mode

Opc: 0x7 Mode: 0

ldmb[.f] regd, $amd

Loads a byte data from memory address to register

ldmd

AMI Format: Protected Mode

Opc: 0x7 Mode: 2

ldmd[.f] regd, $amd

Loads a dword data from memory address to register

---

ldmq

AMI Format: Protected Mode

Opc: 0x7   Mode: 3

ldmq[.f] regd, $amd

Loads a dword data from memory address to double registers

---

ldmw

AMI Format: Protected Mode

Opc: 0x7   Mode: 1

ldmw[.f] regd, $amd

Loads a word data from memory address to register

---

leave

SI Format: Protected Mode

Opc: 0x21   Func: 3

leave[.f] imm

Free the reserved space stack and return the old EFP and ESP values (May be called after using ENTER or ENTERV)

lrot

AMI Format: Protected Mode

Opc: 0x1b   Mode: 2

`lrot[.f] regd, $amd`

Do a left rotate in regd by data

lshf

AMI Format: Protected Mode

Opc: 0x1b   Mode: 0

`lshf[.f] regd, $amd`

Do a left shift in regd by data

---

madc

SI Format: Protected Mode

Opc: 0x14   Func: 5

madc[.f] regd, regb, regp

Multiply regd by regb, then adds regp with carry and store at regd

---

madd

SI Format: Protected Mode

Opc: 0x14   Func: 3

madd[.f] regd, regb, regp

Multiply regd by regb, then adds regp and store at regd

---

mmsd

CDI Format: Protected Mode

Opc: 0xe   Mode: 3

mmsd[.cond][.f] regd, regb, rego

Stores 32-bit value from memory at regb to memory at regd, increments regb and regd, and decrements rego

mmsi

CDI Format: Protected Mode

Opc: 0xe   Mode: 2

mmsi[.cond][.f] regd, regb, rego

Stores 32-bit value from memory at regb to memory at regd, increments regb and regd, and increments rego

mod

AMI Format: Protected Mode

Opc: 0x11 Mode: 3

mod[.f] regd, $amd

Modules a value data to register

---

mov

IR Format: Protected Mode

Opc: 0x30 Mode: 0

mov[.im] regd, regb

Moves a value from one register to another

---

movi

IR Format: Protected Mode

Opc: 0x30   Mode: 1

movi[.im] regd, imm16

Moves a immediate value to a register

---

msbb

SI Format: Protected Mode

Opc: 0x14   Func: 6

msbb[.f] regd, regb, regp

Multiply regd by regb, then subtracts regp with borrow and store at regd

---

msub

SI Format: Protected Mode

Opc: 0x14   Func: 4

msub[.f] regd, regb, regp

Multiply regd by regb, then subtracts regp and store at regd

---

mul

AMI Format: Protected Mode

Opc: 0x11   Mode: 0

mul[.f] regd, $amd

Multiplies a value data to register

---

mv

CDI Format: Protected Mode

Opc: 0x32   Mode: 0

mv[.cond][.f] regd, regb

Moves value from regb to regd conditionally

---

mvfit

SI Format: #Super Mode

Opc: 0x8   Func: 9   Mode: 1

mvfit[.f] regd

Moves a value from register it to regd

---

mvfr

IR Format: #Super Mode

Opc: 0x35   Mode: 0

mvfr[.im] regd, imm

Moves value from extra register in index of imm in processor to regd

---

mvfst

SI Format: #Super Mode

Opc: 0x8   Func: 9   Mode: 3

mvfst[.f] regd

Moves a value from register est to regd

---

mvtit

SI Format: #Super Mode

Opc: 0x8   Func: 9   Mode: 0

mvtit[.f] regb

Moves a value from regb to register it

---

mvtr

IR Format: #Super Mode

Opc: 0x35   Mode: 1

mvtr[.im] imm, regb

Moves value from regb to extra register in index of imm in processor

---

mvtst

SI Format: #Super Mode

Opc: 0x8   Func: 9   Mode: 2

mvtst[.f] regb

Moves a value from regb to register est

---

nand

AMI Format: Protected Mode

Opc: 0x18   Mode: 3

nand[.f] regd, $amd

Do nand bitwise value data to register

---

neg

SI Format: Protected Mode

Opc: 0x14   Func: 2

neg[.f] regd

Do a subtraction of zero by regd, and stores

---

nop

JL Format: Protected Mode

Opc: 0x0

nop

Do not execute any operation

---

not

SI Format: Protected Mode

Opc: 0x19   Func: 0

not[.f] regd

Inverts the bits values of regd

or

AMI Format: Protected Mode

Opc: 0x18   Mode: 1

or[.f] regd, $amd

Do or bitwise value data with register

out

SI Format: #Super Mode

Opc: 0x8   Func: 1   Mode: 0

out[.f] regp, regb

Outputs a value from regb to device in port regp

---

outi

SI Format: #Super Mode

Opc: 0x8   Func: 1   Mode: 1

outi[.f] regp, imm

Outputs a immediate value to device in port regp

---

popas

IR Format: Protected Mode

Opc: 0x3   Mode: 0

popas[.im]

Pops to EST register Application part

---

popp

IR Format: Protected Mode

Opc: 0x4   Mode: 1

popp[.im] regb, regd

Pops values from stack to a sequence of register from regd to regb

popr

IR Format: Protected Mode

Opc: 0x4   Mode: 0

popr[.im] regd

Pops a value from stack to regd

pops

IR Format: #Super Mode

Opc: 0x3   Mode: 2

pops[.im]

Pops to EST register

popss

IR Format: #Super Mode

Opc: 0x3   Mode: 1

popss[.im]

Pops to EST register System part

pshas

IR Format: Protected Mode

Opc: 0xb   Mode: 0

pshas[.im]

Pushes the EST Register Application part to stack

---

pshi

IR Format: Protected Mode

Opc: 0xc   Mode: 0

pshi[.im] imm

Pushes a immediate value to stack

---

pshp

IR Format: Protected Mode

Opc: 0xc   Mode: 2

pshp[.im] regb, regd

Pushes values from sequence regb to regd to stack

---

pshr

IR Format: Protected Mode

Opc: 0xc   Mode: 1

pshr[.im] regb

Pushes regb value to stack

---

pshs

IR Format: #Super Mode

Opc: 0xb   Mode: 2

pshs[.im]

Pushes the EST Register to stack

---

pshss

IR Format: #Super Mode

Opc: 0xb   Mode: 1

pshss[.im]

Pushes the EST Register System part to stack

---

ret

SI Format: Protected Mode

Opc: 0x21   Func: 0

ret[.f]

Returns from a branch

rrot

AMI Format: Protected Mode

Opc: 0x1b   Mode: 3

rrot[.f] regd, $amd

Do a right rotate in regd by data

rshf

AMI Format: Protected Mode

Opc: 0x1b   Mode: 1

rshf[.f] regd, $amd

Do a right shift in regd by data

---

sbb

AMI Format: Protected Mode

Opc: 0x10   Mode: 3

sbb[.f] regd, $amd

Subtracts a value with borrow to register

---

setb

SI Format: Protected Mode

Opc: 0x19   Func: 1

setb[.f] regd, imm8

Sets the bit of regd at offset of imm

---

stab

SI Format: Protected Mode

Opc: 0x34   Func: 7

stab[.f] regd, imm8, regb, regp

Stores a immediate value to register if regb is above than regp

---

stabz

SI Format: Protected Mode

Opc: 0x34   Func: 27

stabz[.f] regd, regp, regb

Stores regp value to register if regb is above than zero

---

stabzi

SI Format: Protected Mode

Opc: 0x34   Func: 17

stabzi[.f] regd, imm8, regb

Stores a immediate value to register if regb is above than zero

---

stae

SI Format: Protected Mode

Opc: 0x34   Func: 9

stae[.f] regd, imm8, regb, regp

Stores a immediate value to register if regb is above or equal than regp

---

staez

SI Format: Protected Mode

Opc: 0x34   Func: 29

staez[.f] regd, regp, regb

Stores regp value to register if regb is above or equal than zero

---

staezi

SI Format: Protected Mode

Opc: 0x34   Func: 19

staezi[.f] regd, imm8, regb

Stores a immediate value to register if regb is above or equal than zero

stb

SI Format: Protected Mode

Opc: 0x38 Func: 3

stb[.f]

Sets the flag BF

stbe

SI Format: Protected Mode

Opc: 0x34 Func: 8

stbe[.f] regd, imm8, regb, regp

Stores a immediate value to register if regb is below or equal than regp

---

stbez

SI Format: Protected Mode

Opc: 0x34   Func: 28

stbez[.f] regd, regp, regb

Stores regp value to register if regb is below or equal than zero

---

stbezi

SI Format: Protected Mode

Opc: 0x34   Func: 18

stbezi[.f] regd, imm8, regb

Stores a immediate value to register if regb is below or equal than zero

---

stbl

SI Format: Protected Mode

Opc: 0x34   Func: 6

stbl[.f] regd, imm8, regb, regp

Stores a immediate value to register if regb is below than regp

stblz

SI Format: Protected Mode

Opc: 0x34   Func: 26

stblz[.f] regd, regp, regb

Stores regp value to register if regb is below than zero

stblzi

SI Format: Protected Mode

Opc: 0x34   Func: 16

stblzi[.f] regd, imm8, regb

Stores a immediate value to register if regb is below than zero

---

stc

SI Format: Protected Mode

Opc: 0x38   Func: 0

stc[.f]

Sets the flag CF

---

steq

SI Format: Protected Mode

Opc: 0x34   Func: 0

steq[.f] regd, imm8, regb, regp

Stores a immediate value to register if regb is equal to regp

---

stez

SI Format: Protected Mode

Opc: 0x34   Func: 20

stez[.f] regd, regp, regb

Stores regp value to register if regb is equal to zero

stezi

SI Format: Protected Mode

Opc: 0x34   Func: 10

stezi[.f] regd, imm8, regb

Stores a immediate value to register if regb is equal to zero

stge

SI Format: Protected Mode

Opc: 0x34   Func: 5

stge[.f] regd, imm8, regb, regp

Stores a immediate value to register if regb is greater or equal than regp

stgez

SI Format: Protected Mode

Opc: 0x34   Func: 25

stgez[.f] regd, regp, regb

Stores regp value to register if regb is greater or equal than zero

stgezi

SI Format: Protected Mode

Opc: 0x34   Func: 15

stgezi[.f] regd, imm8, regb

Stores a immediate value to register if regb is greater or equal than zero

---

stgt

SI Format: Protected Mode

Opc: 0x34   Func: 3

stgt[.f] regd, imm8, regb, regp

Stores a immediate value to register if regb is greater than regp

---

stgtz

SI Format: Protected Mode

Opc: 0x34   Func: 23

stgtz[.f] regd, regp, regb

Stores regp value to register if regb is greater than zero

---

stgtzi

SI Format: Protected Mode

Opc: 0x34   Func: 13

stgtzi[.f] regd, imm8, regb

Stores a immediate value to register if regb is greater than zero

---

stle

SI Format: Protected Mode

Opc: 0x34   Func: 4

stle[.f] regd, imm8, regb, regp

Stores a immediate value to register if regb is less or equal than regp

---

stlez

SI Format: Protected Mode

Opc: 0x34   Func: 24

stlez[.f] regd, regp, regb

Stores regp value to register if regb is less or equal than zero

---

stlezi

SI Format: Protected Mode

Opc: 0x34   Func: 14

stlezi[.f] regd, imm8, regb

Stores a immediate value to register if regb is less or equal than zero

stlt

SI Format: Protected Mode

Opc: 0x34   Func: 2

stlt[.f] regd, imm8, regb, regp

Stores a immediate value to register if regb is less than regp

stltz

SI Format: Protected Mode

Opc: 0x34   Func: 22

stltz[.f] regd, regp, regb

Stores regp value to register if regb is less than zero

---

stltzi

SI Format: Protected Mode

Opc: 0x34   Func: 12

stltzi[.f] regd, imm8, regb

Stores a immediate value to register if regb is less than zero

---

stn

SI Format: Protected Mode

Opc: 0x38   Func: 12

stn[.f]

Sets the flag NF

---

stne

SI Format: Protected Mode

Opc: 0x34   Func: 1

stne[.f] regd, imm8, regb, regp

Stores a immediate value to register if regb is not equal to regp

---

stnz

SI Format: Protected Mode

Opc: 0x34   Func: 21

stnz[.f] regd, regp, regb

Stores regp value to register if regb is not equal to zero

---

stnzi

SI Format: Protected Mode

Opc: 0x34   Func: 11

stnzi[.f] regd, imm8, regb

Stores a immediate value to register if regb is not equal to zero

---

sto

SI Format: Protected Mode

Opc: 0x38   Func: 15

sto[.f]

Sets the flag OF

---

strb

AMI Format: Protected Mode

Opc: 0xf   Mode: 0

strb[.f] $amd, regd

Stores a byte register data to memory address

---

strd

AMI Format: Protected Mode

Opc: 0xf   Mode: 2

strd[.f] $amd, regd

Stores a dword register data to memory address

---

strq

AMI Format: Protected Mode

Opc: 0xf   Mode: 3

strq[.f] $amd, regd

Stores a double dword registers data to memory address

---

strw

AMI Format: Protected Mode

Opc: 0xf   Mode: 1

strw[.f] $amd, regd

Stores a word register data to memory address

---

stsd

CDI Format: Protected Mode

Opc: 0xe   Mode: 1

stsd[.cond][.f] regd, regb, rego

Stores 32-bit value from regb to memory at regd, increments regd, and decrements rego

---

stsi

CDI Format: Protected Mode

Opc: 0xe   Mode: 0

stsi[.cond][.f] regd, regb, rego

Stores 32-bit value from regb to memory at regd, increments regd, and increments rego

---

stv

SI Format: Protected Mode

Opc: 0x38   Func: 6

stv[.f]

Sets the flag VF

stz

SI Format: Protected Mode

Opc: 0x38   Func: 9

stz[.f]

Sets the flag ZF

sub

AMI Format: Protected Mode

Opc: 0x10   Mode: 2

sub[.f] regd, $amd

Subtracts a value data from register

---

swap

SI Format: Protected Mode

Opc: 0x19 Func: 3

swap[.f] regd

Swaps the bytes of regd, changing the endianess

---

swapb

SI Format: Protected Mode

Opc: 0x19   Func: 4

swapb[.f] regd

Swaps the bits of regd, changing its format

---

test

AMI Format: Protected Mode

Opc: 0x1a   Mode: 0

test[.f] $amd

Test the data value and sets the bits acoordling

tgb

SI Format: Protected Mode

Opc: 0x38   Func: 5

tgb[.f]

Toggles the flag BF

tgc

SI Format: Protected Mode

Opc: 0x38   Func: 2

tgc[.f]

Toggles the flag CF

---

tgn

SI Format: Protected Mode

Opc: 0x38   Func: 14

tgn[.f]

Toggles the flag NF

---

tgo

SI Format: Protected Mode

Opc: 0x38   Func: 17

tgo[.f]

Toggles the flag OF

---

tgv

SI Format: Protected Mode

Opc: 0x38   Func: 8

tgv[.f]

Toggles the flag VF

tgz

SI Format: Protected Mode

Opc: 0x38   Func: 11

tgz[.f]

Toggles the flag ZF

wait

SI Format: Protected Mode

Opc: 0x1   Func: 1   Mode: 0

wait[.f]

Waits for any interruption

---

waiti

SI Format: Protected Mode

Opc: 0x1   Func: 1   Mode: 2

waiti[.f] regb

Waits for a specific interruption of regb

---

waiti

SI Format: Protected Mode

Opc: 0x1   Func: 1   Mode: 1

waiti[.f] imm8

Waits for a specific interruption of imm

---

xbr

CDI Format: Protected Mode

Opc: 0x23   Mode: 3

xbr[.cond][.f] regb

Exchange value between regp and epc, doing a branch

xchg

SI Format: Protected Mode

Opc: 0x33   Func: 0

xchg[.f] regd, regb

Exchanges the value of regd by regb

xjp

CDI Format: Protected Mode

Opc: 0x23   Mode: 2

xjp[.cond][.f] regb

Exchange value between regp and epc, doing a jump

---

xor

AMI Format: Protected Mode

Opc: 0x18   Mode: 2

xor[.f] regd, $amd

Do xor bitwise value data to register
