LEG

ARCHITECTURE

(codename)

I. Abstract

This technical document is a computer science specification describing a computer architecture named *LEG Architecture*.

The *LEG Architecture* is a RISC architecture that specifies both 32-bit and 64-bit addressing system implementations.

It supports advanced memory management such as *Paging*, an IEEE 754 compliant *FPU* and an efficient and clean design to support multi-tasking operating systems.

The simplicity of the *LEG Architecture* makes it possible to be implemented on a Field-Programmable Gate Array (FPGA), which turns the implemented CPU suitable for low power consumption applications.

II. Authors

This document, describing the *LEG Architecture Specification*, was written by:

Author's Name:	Pedro A. Hortas
Author's Email:	pedro.hortas@ieee.org
Current Revision:	Draft E5
Date:	9th August 2012

The author was also the creator of *LEG Architecture*.

III. Conventions

Although the concepts found in this document are mostly self-contained and are described as clearly as possible, it is recommended that the reader has a some basic understating on Computer Architecture domain.

This document also uses some notations for numbering, such as binary and hexadecimal, that the reader must be comfortable with in order to make this document easier to read.

Sometimes, abbreviations and/or acronyms are used. The following list covers most part of these abbreviations and acronyms with their respective meaning:

- **ALU** – Arithmetic Logic Unit
- **ASCII** – American Standard Code for Information Interchange
- **CPU** – Central Process Unit
- **FPU** – Floating-Point Unit
- **ID** – Identifier
- **IEEE** – Institute of Electrical and Electronics Engineers
- **I/O** – Input / Output
- **LEG** – LEG Architecture (either 32-bit and 64-bit)
- **LEG32** – LEG Architecture with 32-bit addressing system
- **LEG64** – LEG Architecture with 64-bit addressing system
- **LSB** – Least Significant Bit
- **MSB** – Most Significant Bit
- **Opcode** – Operation Code
- **OS** – Operating System
- **RISC** – Reduced Instruction Set Computing

IV. Table of Contents

V. Specifications

1. Architecture Overview

The *LEG Architecture* is a RISC architecture. It specifies twenty-eight (28) registers, being twelve (12) defined as *Control Registers* (See ***Section 2.1. Control Registers***), eight (8) *General Purpose Registers* (See ***Section 2.2. General Purpose Registers***), four (4) *Arithmetic Logic Registers* (See ***Section 2.3. Arithmetic Logic Registers***) and four (4) *Floating-Point Registers* (See ***Section 2.4. Floating-Point Registers***). It also implements simple *Paging* (See ***Section 7.4. Paging***) and *Multi-Tasking* (See ***Section 8. Multi-Tasking***) mechanisms.

LEG Architecture specifies both 32-bit and 64-bit implementations: The *LEG32 Architecture* (32-bit) and *LEG64 Architecture* (64-bit).

1.1. Notations

This section describe the notations used along this document.

Numbers in this document may be represented either in decimal, hexadecimal or binary format. The notations used for these numbers are:

- Decimal numbers are in the format 00
- Hexadecimal numbers are in the format 0x00
- Binary numbers are in the format 0b00

1.2. Endianess

Endianess, also known as *Byte Order*, is the ordering by which the CPU represents, in the system memory, an addressable value that is greater than 1 byte. There are two types of *Endianess*: *Little Endian* and *Big Endian*.

Little Endian systems represent a greater than 1 byte value in system memory from the least significant byte to the most significant.

Big Endian systems represent a greater than 1 byte value in system memory from the most significant byte to the least significant.

LEG Architecture is *Big Endian*.

1.3. Floating-Point Numbers

LEG Architecture Floating-Point numbers are represented as specified by the IEEE 754 standard. Please refer to this standard for more information regarding this topic.

1.4. Exceptions

Exceptions in *LEG Architecture* are considered a state that cannot be handled alone by the *CPU*, and therefore are considered a fault state. This means that there's no distinguishable state that separate an *Exception* from a *Fault*, being any *Exception* in *LEG Architecture* treated as a *Fault*.

Faults can occur at any moment, when the CPU is unable to handle the current machine state. These states may be caused by software requesting operations that the CPU cannot perform, or by hardware such as the detection of a failed hardware component either by the CPU or by an external controller, causing a *Machine Check Fault*.

For more information regarding *Fault States*, please refer to ***Section 6. Faults*** and ***Section 6.1. Fault States***.

1.5. External Hardware Controllers

LEG Architecture does not specify how the CPU should interact with

External Hardware Controllers. This should be implementation specific of the CPU.

The CPU implementation is responsible to correctly interpret what is specified in this document and implement their own communication mechanisms between the CPU and the *External Hardware Controllers*.

Refer to ***Section 1.6. Expansibility and Extensions*** for more information regarding this topic.

1.6. Expansibility and Extensions

There are no limitations on this specification that inhibits the expansibility of the *LEG Architecture* as long as the CPU implementation is in accordance with the specifications described in this document.

It is allowed to implement new *Interrupts* (See ***Section 5. Interrupts***) for expansibility purposes as long as their IDs fall in the range of the last ten (10) *Architecture-Specific Interrupts* (from ID 21 to ID 31) (See ***Section 5.2. Architecture-Specific Interrupts***). These IDs are specially reserved for that purpose and will never be used in future *LEG Architecture* specifications.

2. Registers

There are twenty-eight (28) registers available in *LEG Architecture* divided as twelve (12) *Control Registers* (*RIP*, *RST*, *RFF*, *RFA*, *RBT*, *RCT*, *RPA*, *RRA*, *RSA*, *RCMP*, *RLGIC* and *RARTH*), eight (8) *General Purpose Registers* (*RGP1*, *RGP2*, *RGP3*, *RGP4*, *RGP5*, *RGP6*, *RGP7* and *RGP8*), four (4) *Arithmetic Logic Registers* (*RAL1*, *RAL2*, *RAL3* and *RAL4*) and four (4) *Floating Point Registers* (*RFP1*, *RFP2*, *RFP3* and *RFP4*). All register sizes are 32-bit long for *LEG32 Architecture* and 64-bit long for *LEG64 Architecture*.

Control Registers are responsible for CPU, instruction and code execution flow control. They provide ways to configure parameters for memory and task management, instruction behavior modification, etc.

General Purpose Registers, as the name states, are registers for general purpose operations that may be used for memory handling, arithmetic and logic operations.

Arithmetic Logic Registers are optimized registers for arithmetic and logic operations. Despite the fact that those operations may be performed with *General Purpose Registers*, it's strongly recommended the use of *Arithmetic Logic Registers* for this purpose as they are optimized for such operations.

Floating-Point Registers are reserved for operations that involve floating-point numbers. It is important to note that floating-point operations shall never be performed through any other register but the *Floating-Point Registers*, as undefined behavior may occur.

2.1. Control Registers

The *Control Registers* control the CPU behavior. There are twelve (12) *Control Registers* that can be configured by operating system processes running at *Privilege Level 0* (see ***Section 3. Privilege Levels***).

Other processes running at a different *Privilege Level* may only read the *Control Registers* and are unable to modify them directly, except for *RSA*, *RCMP*, *RARTH* and *RLGIC* that can still be modified by code being executed at any *Privilege Level*. The specifications for each *Control Register* are defined in the following sub-sections (from ***Section 2.1.1. Instruction Pointer Register*** through ***Section 2.1.11. Arithmetic Register***).

2.1.1. Instruction Pointer Register (RIP)

The *Instruction Pointer Register* (*RIP*) controls the code execution flow. It points directly to a memory reference containing valid opcodes, performing their respective operations.

When *Paging* is enabled (see ***Section 7.2. Paging***) the *Logical Address* referenced by *RIP* is first translated to the respective *Physical Address*. Pages containing code must have the *Executable Permission* flag enabled or a *Page Permission Fault* (see ***Section 6.1.10. Page Permission Fault***) will occur.

RIP cannot be directly modified by processes running at *Privilege Level 1* nor *Privilege Level 0* (see ***Section 3. Privilege Levels***). Any operation requesting modifications to *RIP* will cause an Invalid Instruction Fault to occur (see ***Section 6.1.7. Illegal Instruction Fault***). Code execution flow may be modified through *CALL*, *RET* and *JMP Instructions* (see ***Section 4.6. JMP***, ***Section 4.7. CALL*** and ***Section 4.8. RET***).

After CPU initialization, *RIP* will point to the first address in system memory that is not specified as a *Reserved Memory Region* (See ***Section 7.1. Reserved Memory Regions***).

2.1.2. Status Register (RST)

The *Status Register* (*RST*) provides a set of configurable bit flags for CPU control and configuration. There are N possible configuration flags for

RST, being N equal to 32 on *LEG32 Architecture* or equal to 64 on *LEG64 Architecture*, but only 5 are currently implemented, being the others reserved for future specifications and shall not be used as a *Bad Register Value Fault* may occur (See ***Section 6.1.4. Bad Register Value Fault***).

Table 2.1 describes the currently implemented flags for *RST* register.

RST bit	Description	Status	Section
0	Enable/Disable Interrupts	Set to enable	**5.** Interrupts
1	Enable/Disable Fault Handling	Set to enable	**6.** Faults
2	Enable/Disable Task Registers	Set to enable	**8.** Multi-Tasking
3	Privilege Level Configuration	Set to Privilege Level 1	**3.** Privilege Levels
4	Enable/Disable Paging	Set to enable	**7.4.** Paging

(Table 2.1. Status Register Description)

Detailed information on flags 0 to 4 can be found on the corresponding indicated sections.

2.1.3. Fault Flags Register (RFF)

The *Fault Flags Register* (*RFF*) indicates, through bit flags, which *Faults* occurred. Whenever a *Fault* occur (See ***Section 6. Faults***), the corresponding bit for that fault is set in this register. If *Fault Handling* (See ***Section 6.2. Fault Handling***) is enabled, the routine pointed by *RFA* (See ***Section 2.1.4. Fault-Handler Address Register***) is executed and if *Task Registers* are enabled (See ***Section 2.1.6. Current Task Register*** and ***Section 9. Multi-Tasking***) the *Current Task Context* is saved to the address pointed by *RCT* and the *Base Task Context* is loaded from *RBT* address.

Table 2.2 describes the *Fault* bit flags currently defined for *RFF* register.

RFF bit	Fault	Section
0	Machine Check	**6.1.1.** Machine Check Fault
1	Bad Memory Reference	**6.1.2.** Bad Memory Referenced Fault
2	Bad Register Reference	**6.1.3.** Bad Register Reference Fault
3	Bad Register Value	**6.1.4.** Bad Register Value Fault
4	Bad Operation Value	**6.1.5.** Bad Operation Value Fault
5	Illegal Interrupt	**6.1.6.** Illegal Interrupt Fault
6	Illegal Instruction	**6.1.7.** Illegal Instruction
7	Floating Point Unit	**6.1.8.** Floating Point Unit Fault
8	Arithmetic Logic Unit	**6.1.9.** Arithmetic Logic Unit Fault
9	Page Permission	**6.1.10.** Page Permission Fault
10	Page	**6.1.11.** Page Fault
11	Privilege	**6.1.12.** Privilege Fault
12	Input/Output Operation	**6.1.13.** Input/Output Operation Fault
13	Interrupt	**6.1.14.** Interrupt Fault
24-31	Interrupt ID	**6.1.14.** Interrupt Fault

(Table 2.2. Fault Flags Register Description)

2.1.4. Fault-Handler Address Register (RFA)

Fault-Handler Address Register (*RFA*) must be configured to hold the address for an operating system Fault Handler routine when *RST* bit 1 is set (See ***Section 2.1.2. Status Register (RST)***). In order to enable Fault Handling (See ***Section 6.2. Fault Handling***), the bit 1 of *RST* must be set to 0 and a valid *Physical Address* or, if *Paging* is enabled, a *Logical Address* (See ***Section 7.3. Physical Address Space*** and ***Section 7.4. Logical Address Space***) must be set as the value of *RFA*.

Whenever a *Fault* occurs and *RST* bit 2 is set, the *Current Task Context* is automatically saved by the CPU to the address pointed by *RCT*

(See ***Section 2.1.5.2. Current Task Register (RCT)***) and the *Base Task Context* is loaded from *RBT* (See ***Section 2.1.5.1. Base Task Register (RBT)***). See ***Section 8. Multi-Tasking*** for more information regarding this topic.

The operating system routine responsible for *Fault Handling* may identify which *Faults* occurred through *RFF* status.

2.1.5. Task Registers (RBT and RCT)

Task Registers aid the control of *Task Contexts* (See ***Section 8.3. Context Switching***). A *Task Context* may be defined as the state of the Registers of a given task or process. The *Task Context* type is defined by a *Task Structure* (See ***Section 8.1. Task Structure***). For detailed information regarding this topic, refer to ***Section 8. Multi-Tasking***.

Two Task Registers are available for all LEG Architectures: *Current Task Register* (*RCT*) and *Base Task Register* (*RBT*) (See ***Section 2.1.5.1. Base Task Register (RBT)*** and ***Section 2.1.5.2. Current Task Register (RCT)***).

The *RBT* shall only be used by tasks running at *Privilege Level 0* (see ***Section 3.1 Privilege Level 0***), such as the Operating System Kernel and it is specially designed for that purpose. The *Task Context* saved at the memory address pointed by *RBT* is loaded every time an *Interrupt* or a *Fault* occur and before calling any *User* or *Architecture-specific Interrupt Handling Routine* or *User-defined Fault Handling Routine*. The *Task Context* loaded under this circumstances will only update the *Control Registers*, except *RIP, RFF* and *RCT*. *General Purpose Registers*, *Arithmetic Logic Registers* and *Floating Point Registers* saved on *RBT* location are ignored but may or may not be modified by *Interrupt Handling Routines* and *Fault Handling Routines* (See ***Section 5. Interrupts***, ***Section 6. Faults***, ***Section 2.1.5.1 Base Task Register (RBT)*** and ***Section 2.1.5.2. Current Task Register (RCT)***). The *Task Context* is saved to *RBT* location every time a *LTSK* instruction is executed (See ***Section 4.14. LTSK***).

The *RCT* shall only point to *Task Structures* running at *Privilege Level 1* (see ***Section 3.2 Privilege Level 1***), such as Operating System User-Space tasks or processes. The *Task Context* is saved to *RCT* memory address location every time an *Interrupt* or *Fault* occur and before calling any *User* or *Architecture-specific Interrupt Handling Routine* or *User-defined Fault Handling Routine*. All *Registers* states are saved. After this operation is performed, *RBT Task Context* is loaded. The *Task Context* is loaded from *RCT* every time a *LTSK* instruction is executed (See ***Section 4.14. LTSK***). Before loading *RCT Task Context*, *RBT Task Context* is saved.

2.1.5.1 Base Task Register (RBT)

The *Base Task Register* (*RBT*) is only active when the bit 2 of *RST* is set to 1. If active, it must point to a valid Physical or Logical Address (See ***Section 7.2. Physical Address Space*** and ***Section 7.3. Logical Address Space***) capable of storing a continuous memory region to hold data with length of 116 bytes for *LEG32 Architecture* and 228 bytes for *LEG64 Architecture* (See ***Section 8.1. Task Structure***). The CPU uses this register to load the *Base Task Context* of the *Operating System* kernel task when an *Interrupt* or *Fault* occur (See ***Section 5. Interrupts*** *and* ***Section 6. Faults***).

Only the *Control Registers*, except *RIP, RFF and RCT*, are loaded from the *Task Structure* pointed by *RBT* address. *General Purpose Registers*, *Arithmetic Logic Registers* and *Floating Point Registers* may or may not be modified during the *Interrupt* or *Fault* occurrence. *User-defined* Interrupt Handlers do not change non-*Control Registers* unless the handler routine does so. For detailed information about this topic please refer to ***Section 5. Interrupts*** and ***Section 6. Faults***.

Whenever the *Operating System* needs to modify *RBT* address for *Task Structure* relocation, it is recommended to disable *Interrupts* and *Fault Handling* during the routine responsible for this modification.

The *Base Task Context* is saved at *RBT* address when a *LTSK* instruction (See ***Section 4.14. LTSK***) is executed to load *CTR* (See ***Section 2.1.6 Current Task Register (RCT)***).

RBT shall always point to a *Task Structure* representing a *Privilege Level 0* task or a *Machine Check Fault* will occur when a Context Switch is performed. Please refer to ***Section 3. Privilege Levels***, ***Section 6.1.1. Machine Check Fault*** and ***Section 8.3. Context Switching*** for more information regarding this topic.

2.1.5.2 Current Task Register (RCT)

The *Current Task Register* (*RCT*) is only active when the bit 2 of *RST* is set to 1. If active, it must point to a valid *Physical* or *Logical Address* (See ***Section 7.3. Physical Address Space*** and ***Section 7.4. Logical Address Space***) capable of storing a continuous memory region to hold data with length of 116 bytes for *LEG32 Architecture* and 228 bytes for *LEG64 Architecture (See **Section 8.1. Task Structure**)*. The CPU uses this register to save the *Current Task Context* of the current running task when an *Interrupt* or *Fault* occur (See ***Section 5. Interrupts*** and ***Section 6. Faults***).

There is no way to force the CPU to save the current *Task Context* through an instruction or any method other than an *Interrupt* or *Fault* occurrence.

Saved *Task Context* may only be loaded by operating system tasks running at *Privilege Level 0* (See ***Section 3. Privilege Level***) through the instruction *LTSK* (See ***Section 4.14. LTSK***). This instruction also causes the *Base Task Context* to be saved to *RBT* address.

2.1.6. Paging Address Register (RPA)

The *Paging Address Register* (*RPA*) is only active when the bit 4 of *RST* is set to 1. If active, it must point to a valid *Physical Address* (See

Section 7.3. Physical Address Space) capable of storing a continuous memory region to hold data with length of 24 bytes for *LEG32 Architecture* and 48 bytes for *LEG64 Architecture*. The CPU uses this register to load a *Page Structure* in order to perform *Address Translations* (See ***Section 7.4.2. Address Translation***).

It is imperative that this register points to a *Physical Address* since there's no way for the CPU to translate a *Logical Address* (See ***Section 7.3. Logical Address Space***) before loading a *Page Structure*.

For more information regarding *Paging*, please refer to ***Section 7.4. Paging***.

2.1.7. Return Address Register (RRA)

The *Return Address Register* (*RRA*) shall point to a memory reference capable to grow as it was a Stack Address Register (See ***Section 2.1.8. Stack Address Register (RSA)***). It stores the *Return Addresses* when a *CALL* (See ***Section 4.7. CALL***) instruction is performed. The *RET* (See ***Section 4.8. RET***) instruction reads the last 4 bytes on *LEG32 Architectures or the last 8 bytes on LEG64 Architectures* that were written by *CALL* in order to properly set the *RIP*.

Whenever a *CALL* instruction occurs, the next instruction address after *CALL* is stored at the memory reference pointed by *RRA*. *RRA* content is then incremented by the value of 4 on *LEG32 Architecture* or by the value of 8 on *LEG64 Architecture*.

Whenever a *RET* instruction occurs, the *RRA* value is decremented by 4 on *LEG32 Architecture* or by 8 on *LEG64 Architecture* and the value stored at that memory reference is loaded onto *RIP*.

If *Paging* is enabled (See ***Section 7.4. Paging***), it's imperative that the *Page Permission* (See ***Section 7.4.3. Page Permissions***) of the *Page* mapping the *RRA* memory region be set to *Read-Only*. Failing to do so will cause a *Page Permission Fault* (See ***Section 6.1.10. Page Permission***

Fault) to occur.

2.1.8. Stack Address Register (RSA)

The *Stack Address Register* (*RSA*) is reserved for *Stack Memory Management.* If *Paging* (See ***Section 7.4. Paging***) is enabled, it's recommended that the *Page Permission* (See ***Section 7.4.3. Page Permissions***) of the *Page* mapping the stack memory region shall be set to *Read-Only.*

There are no implemented instructions that abstract the *Stack Memory Management* in *LEG Architecture*. The *RSA* value must be handled through instructions such as *CPVL*, *CPVR*, *ARTH*, *LGIC*, etc. (See ***Section 4. Instruction Set***).

The growth direction of stack in *LEG Architecture* is arbitrary.

2.1.9. Comparator Register (RCMP)

Comparator Register (*RCMP*) is a bit flag register responsible for behavior modification of the *CMP* instruction (See ***Section 4.5. CMP***). This means that *CMP* instruction behaves differently, depending on the configuration set at *RCMP*. Note that *RCMP* must be configured before *CMP* instruction is performed.

RCMP instructs *CMP* on how to compare the values. The result of the comparison is stored at bit 0 of *RCMP* after *CMP* is performed.

Table 2.3 describes permitted flags for *RCMP.*

More than one flag may be set to *RCMP* in order to perform comparisons such as *Greater Than* or *Equal*, or *Lesser Than* or *Equal*, by respectively setting the bits 4 and 2 and bits 4 and 3.

RCMP bit	Description	Obs.
0	Result of CMP instruction	0 is False, 1 is True
1	Not Equal	-
2	Greater Than	-
3	Lesser Than	-
4	Equal	-

(Table 2.3. Comparator Register Description)

2.1.10. Logic Register (RLGIC)

Logic Register (*RLGIC*) is a bit flag register responsible for behavior modification of the *LGIC* instruction (See ***Section 4.10. LGIC***). This means that *LGIC* instruction behaves differently, depending on the configuration set at *RLGIC*. Note that *RLGIC* must be configured before *LGIC* instruction is performed.

RLGIC instructs *LGIC* on which operation will be performed on the values. The result of the operation is stored at the target register LGIC instruction.

Table 2.4 describes the permitted flags for *RLGIC.*

More than one flag may be set to *RCMP* in order to perform operations such as *XNOR*, *NAND* or *NOR*, by respectively setting the bits 1 and 0, 2 and 0 and bits 3 and 0.

RLGIC bits ranging from 16 to 18 specify the number of LSBs affected on the *LGIC* (See ***Section 4.10. LGIC***) instruction operands. Only RAL register operands are sensitive to these flags. Note that bit 18 is only implemented on *LEG64 Architecture*. If set on *LEG32 Architecture* it will cause a *Bad Register Value Fault* to occur (See ***Section 6.1.4. Bad Register Value Fault***). If all bits, ranging from 16 to 18, are cleared, the operation is performed including all register bits.

RLGIC bit	Description	Obs.
0	NOT operation	Requires two (2) operands.
1	XOR operation	-
2	AND operation	-
3	OR operation	-
4	Shift Left operation	-
5	Shift Right operation	-
6	Rotate Left operation	-
7	Rotate Right operation	-
16	8-bit Operation (LSB)	Only evaluated on RAL registers
17	16-bit Operation (LSB)	Only evaluated on RAL registers
18	32-bit Operation (LSB)*	Only evaluated on RAL registers *LEG64 Only

(Table 2.4. Logic Register Description)

2.1.11. Arithmetic Register (RARTH)

Arithmetic Register (*RARTH*) is a bit flag register responsible for behavior modification of the *ARTH* instruction (See ***Section 4.9. ARTH***). This means that *ARTH* instruction behaves differently, depending on the configuration set at *RARTH*. Note that *RARTH* must be configured before *ARTH* instruction is performed.

RARTH instructs *ARTH* on which operation will be performed on the values. The result of the operation is stored at the target register of the *ARTH* instruction.

Table 2.5 describes the permitted flags for *RARTH.*

Operations involving signed values must have the bit 5 flag set at *RARTH*.

RCMP bit	Description	Obs.
0	Multiplication	-
1	Division	-
2	Subtraction	-
3	Addition	-
4	Modulus	-
5	Signed Operation	Set to perform signed operations
6	Overflow Indicator	Set when Overflow occur
7	Underflow Indicator	Set when Underflow occur
8	Extended ALU Operand (RAL1)	-
9	Extended ALU Operand (RAL2)	-
10	Extended ALU Operand (RAL3)	-
11	Extended ALU Operand (RAL4)	-
12	Extended FPU Operand (RFP1)	-
13	Extended FPU Operand (RFP2)	-
14	Extended FPU Operand (RFP3)	-
15	Extended FPU Operand (RFP4)	-
16	8-bit Operation (LSB)	Only evaluated on RAL registers
17	16-bit Operation (LSB)	Only evaluated on RAL registers
18	32-bit Operation (LSB)*	Only evaluated on RAL registers *LEG64 only

(Table 2.5. Arithmetic Register Description)

RARTH bits 6 and 7 are set when an arithmetic operation causes the target register to respectively overflow or underflow its value.

The *Overflow* behavior is defined to reset all the bits of the register, where the overflow occurred, when an additional MSB, beyond the register size, was required during the arithmetic operation to represent the result. This means, assuming that *RARTH* bits from 16 to 18 are

cleared, that the operation 0xFFFFFFFF + 0x03 will result in the value 0x02 (*LEG32*) with the *RARTH* bit 6 set indicating that an *Overflow* occurred.

The *Underflow* behavior is defined to set all the bits of the register, where the underflow occurred, to one when all its bits were set to 0 during the arithmetic operation. This means, assuming that *RARTH* bits from 16 to 18 are cleared, that the operation 0x00000000 – 0x03 will result in the value 0xFFFFFFFD (*LEG32*) with the *RARTH* bit 7 set indicating that an *Underflow* occurred.

Extended ALU Operand (*RARTH* bits 8 to 11) and *Extended FPU Operand* (*RARTH* bits 12 to 15) flags activate the corresponding register to extend the target operand, being the target operand the least significant part and the extended operand the most significant part. This enables the possibility to perform 32-bit operations with 64-bit results on LEG32 Architecture and 64-bit operations with 128-bit results on LEG64 Architecture. When one extended operand is active, neither the *Overflow* nor *Underflow* flag will be set during the arithmetic operation.

RARTH bits ranging from 16 to 18 specify the number of LSBs affected on the *ARTH* (See ***Section 4.9. ARTH***) instruction operands. Only RAL register operands are sensitive to these flags. Note that bit 18 is only implemented on *LEG64 Architecture*. If set on *LEG32 Architecture* it will cause a *Bad Register Value Fault* to occur (See ***Section 6.1.4. Bad Register Value Fault***). If all bits, ranging from 16 to 18, are cleared, the operation is performed including all register bits.

2.2. General Purpose Registers (RGP1 to RGP8)

There are eight *General Purpose Registers* available on *LEG Architecture*: *RGP1*, *RGP2*, *RGP3*, *RGP4*, *RGP5*, *RGP6*, *RGP7* and *RGP8*.

These registers permit the following types of operations: *Literal Assignment*, *Memory Handling*, *Logic Operations*, *Arithmetic Operations* (excluding *Floating Point Operations*) and *Comparisons*.

Any *General Purpose Register* may be used along with any other register in a given instruction that permits two registers as arguments.

Although *Logic* and *Arithmetic Operations* are permitted between *General Purpose Registers*, it is strongly recommended the use of *Arithmetic Logic Registers* for this purpose (See ***Section 2.1.13. Arithmetic Logic Registers (RAL1 to RAL4)***).

2.3. Arithmetic Logic Registers (RAL1 to RAL4)

There are four *Arithmetic Logic Registers* available on *LEG Architecture*: *RAL1*, *RAL2*, *RAL3* and *RAL4*.

Permitted operations for *Arithmetic Logic Registers* are: *Literal Assignment*, *Memory Handling*, *Logic Operations*, *Arithmetic Operations* (excluding *Floating Point Operations*) and *Comparisons*.

These are *ALU* (*Arithmetic Logic Unit*) registers, being highly optimized for *Arithmetic* and *Logic Operations*. It is strongly recommended that *Arithmetic* and *Logic Operations* involving the instructions *ARTH* and *LGIC* should be performed directly on these registers, since they are integrated in the *ALU* unit. *Arithmetic* and *Logic Operations* are permitted on other registers, but the CPU needs to internally copy the values from the non-*RAL* registers into *RAL* registers before performing the requested operation. If non-*RAL* registers are used for an *Arithmetic* or *Logic Operation*, the CPU will cache the values of two *RAL* registers before copying the values of the non-*RAL* registers into it. After the operation is completed, the previous values of the used *RAL* registers are restored.

Arithmetic and *Logic Operations* involving *Floating Point* values are required to be performed with *Floating Point Registers* (See ***Section 2.1.14. Floating-Point Registers (RFP1 to RFP4)***).

2.4. Floating-Point Registers (RFP1 to RFP4)

There are four *Floating-Point Registers* available on *LEG Architecture*: *RFP1*, *RFP2*, *RFP3* and *RFP4*.

Permitted operations for *Floating-Point Registers* are: *Literal Assignment*, *Arithmetic Operations* and *Comparisons*.

These registers are part of an *ALU* (*Arithmetic Logic Unit*) sub-component called *FPU* (*Floating-Point Unit*), being highly optimized for *Floating-Point Arithmetic* and *Logic Operations*.

Floating-Point Arithmetic Operations are required to be performed in the *Floating-Point Registers*. The behavior for *Floating-Point Operations* performed with non-*RFP* registers is unspecified.

3. Privilege Levels

LEG Architecture support two *Privilege Levels* of operation: *Privilege Level 0* and *Privilege Level 1*.

Privilege Level is selected by modifying *RST* bit 3.

Privilege Level 0 is the highest privilege level available. Code executed at this privilege level is allowed to modify directly any CPU register, except *RIP*. This level is granted for code being executed with *RST* bit 3 unset. See ***Section 3.1. Privilege Level 0*** for more information regarding this topic.

Privilege Level 1 is the lowest privilege level available. Code executed at this privilege level is only allowed to directly modify *General Purpose Registers*, *Arithmetic Logic Registers*, *Floating Point Registers* and the *Control Registers RSA*, *RCMP*, *RLGIC* and *RARTH*. This level is granted for code being executed with *RST* bit 3 set. See ***Section 3.2. Privilege Level 1*** for more information regarding this topic.

Privilege Levels are also evaluated by *Interrupt Handling Routines*. *Architecture-specific Interrupt Handlers* run at *Privilege Level 0*. Software *Interrupts* that cause an *Architecture-specific Interrupt Handlers* to be executed can only be performed by code running at *Privilege Level 0* (Eg. *INTR* 0x0A) (See ***Section 4.11. INTR***). *User-defined Interrupts* may set the minimum Privilege Level required to invoke the respective handler. For more information regarding this topic, refer to ***Section 5. Interrupts***.

3.1. Privilege Level 0

Privilege Level 0 is the highest privilege level available on *LEG Architecture*. This privilege level shall be reserved for tasks or processes requiring full control of *CPU Configuration*, *Direct Hardware Access*, *Memory Management*, *Task Management*, *Interrupt Handling and Fault Handling*. Refer to ***Section 2.1. Control Registers***, ***Section 5. Interrupts***,

Section 6. Faults, ***Section 7. Memory Management*** and ***Section 8. Multi-Tasking*** for more information regarding these topics.

It is allowed to modify directly all registers for code running at this privilege level except *RIP* that can only be indirectly modified by *CALL* and *RET* instructions (See ***Section 4.7. CALL*** and ***Section 4.8. RET***). There is also no restrictions for any instructions referred in ***Section 4. Instruction Set***.

It is strongly recommended, when implementing a multi-task, multi-user operating system based on *LEG Architecture*, that only the kernel code should run at this privilege level. User-land tasks or processes should always run at *Privilege Level 1* (See ***Section 3.2. Privilege Level 1***).

3.2. Privilege Level 1

Privilege Level 1 is the lowest privilege level available on *LEG Architecture*. This privilege level shall be reserved for tasks or processes that do not require direct access to hardware resources nor hardware management.

Code running at this privilege level cannot modify any of the *Control Registers* directly, except for *RSA*, *RCMP*, *RLGIC* and *RARTH*. The *RIP* register can still be modified indirectly by *CALL* and *RET* instructions (See ***Section 4.7. CALL*** and ***Section 4.8. RET***). The instruction *LTSK* (See ***Section 4.14. LTSK***) is disabled at this level, causing an *Illegal Instruction Fault* (See ***Section 6.1.7. Illegal Instruction Fault***) if its execution is attempted.

Software Interrupts are also disabled for *Architecture-specific Interrupts*. *Software Interrupts* targeting *User-defined Interrupts* may or may be enabled, depending on the privilege level set on the *Interrupt Vector* (See ***Section 5.1. Interrupt Vector***) for that routine. Attempts to execute software interrupts reserved or configured to *Privilege Level 0* will cause a *Privilege Fault* to occur (See ***Section 6.1.12. Privilege Fault***).

It is strongly recommended, when implementing multi-task, multi-user operating systems based on *LEG Architecture*, that the *User-land* tasks or processes run at this privilege level.

4. Instruction Set

LEG Architecture implements fourteen (14) instructions: *CPVR*, *CPVL*, *CPR*, *CPRR*, *CMP*, *JMP*, *CALL*, *RET*, *ARTH*, *LGIC*, *INTR*, *CEB*, *NOP* and *LTSK*.

These mnemonics are translated into *opcodes* by an assembler. The *opcode* structure for the combination of operation and its operands is described in ***Table 4.1*** and ***Table 4.2***.

The *Register IDs* for each *LEG Architecture* register are described in ***Table 4.3***.

Opcode	Instruction ID	Operands	Source Operand	Target Operand
0x000000ZZ	Z	0	-	-
0x0000YYZZ	Z	1	Special Literal Y	-
0x00XXYYZZ	Z	2	Register Y	Register X
0x0000YYZZ 0xNNNNNNNN	Z	2	Register Y	Memory N
0x000000ZZ 0xNNNNNNNN	Z	1	Memory N	-
0x000000ZZ 0xNNNNNNNN 0xKKKKKKKK	Z	2	Memory N Literal N	Memory K
0x00XX00ZZ 0xNNNNNNNN	Z	2	Memory N Literal N	Register X

(Table 4.1. Opcode Structure)

ID	Description	Type
XX	Register Reference ID	Operand
YY	Special Literal Register Reference ID	Operand
ZZ	Instruction ID	Operation
NN	Literal Memory Reference	Operand
KK	Memory Reference	Operand

(Table 4.2. ID descriptions of Table 4.1)

Register	ID
RIP	0x00
RST	0x04
RFF	0x08
RFA	0x0C
RBT	0x10
RCT	0x14
RPA	0x18
RRA	0x1C
RSA	0x20
RCMP	0x24
RLGIC	0x28
RARTH	0x2C
RGP1	0x30
RGP2	0x34
RGP3	0x38
RGP4	0x3C
RGP5	0x40

RGP6	0x44
RGP7	0x48
RGP8	0x4C
RAL1	0x50
RAL2	0x54
RAL3	0x58
RAL4	0x5C
RFP1	0x60
RFP2	0x64
RFP3	0x68
RFP4	0x6C

(Table 4.3. Register ID Reference)

The next sub-sections (From ***Section 4.1. CPVR*** to ***Section 4.14. LTSK***) specify the opcodes for each Instruction and describe in detail their operations.

4.1. CPVR

Mnemonic: CPVR
Instruction Opcode: 0x00000001
Maximum Operands: 2
Minimum Operands: 2
Source Operand Type: Register
Target Operand Type: Register, Memory Address
Maximum Clock Cycles: To be defined

Description: Copy the register contents referred in the source operand to the target operand.

Opcodes Example:

```
CPVR RST, RGP1              # 0x00043001
```

```
CPVR RGP7, RFP1            # 0x00486001
CPVR RGP2, 0xAABBCCDD      # 0x00003401 0xAABBCCDD
```

4.2. CPVL

Mnemonic:	CPVL
Instruction Opcode:	0x00000002
Maximum Operands:	2
Minimum Operands:	2
Source Operand Type:	Literal
Target Operand Type:	Register, Memory Address
Maximum Clock Cycles:	To be defined

Description: Copy a literal value from the source operand to the target operand.

Opcode Example:

```
CPVL 0x12, RAL1            # 0x00500002 0x00000012
CPVL 0xAC, RSA             # 0x00200002 0x000000AC
CPVL 0xABCD, 0x11223344    # 0x00000002 0x0000ABCD 0x11223344
```

4.3. CPR

Mnemonic:	CPR
Instruction Opcode:	0x00000003
Maximum Operands:	2
Minimum Operands.	2
Source Operand Type:	Register, Memory Address
Target Operand Type:	Register, Memory Address
Maximum Clock Cycles:	To be defined

Description: Copy the value referenced by the memory address in the source operand to the target operand. If the source operand is a register, its value is considered the memory address to copy the value

from.

Opcode Example:

```
CPR RSA, RGP1              # 0x00203003
CPR 0x3344, RSA            # 0x00200003 0x00003344
CPR 0xAABB, 0xCCDD         # 0x00000003 0x0000AABB 0x0000CCDD
```

4.4. CPRR

Mnemonic:	CPRR
Instruction Opcode:	0x00000004
Maximum Operands:	2
Minimum Operands:	2
Source Operand Type:	Register
Target Operand Type:	Register
Maximum Clock Cycles:	To be defined

Description: Copy the register contents referenced by the source operand to the memory address location referenced by the value of the target operand.

Opcode Example:

```
CPRR RGP3, RSA             # 0x00382004
```

4.5. CMP

Mnemonic:	CMP
Instruction Opcode:	0x00000005
Maximum Operands:	2
Minimum Operands:	2
Source Operand Type:	Register
Target Operand Type:	Register
Maximum Clock Cycles:	To be defined

Description: Performs the comparison currently selected on *RCMP* (See ***Section 2.1.9. Comparator Register (RCMP)***) between the register contents referenced by the source operand and the register contents referenced by the target operand. The result of the comparison is stored at *RCMP* bit 0. Also, *CMP* instruction clears *RCMP* bit 0 when it is executed.

Opcode Example:

```
CMP RAL2, RGP5              # 0x00544005
```

4.6. JMP

Mnemonic:	JMP
Instruction Opcode:	0x00000006
Maximum Operands:	1
Minimum Operands:	1
Source Operand Type:	Register, Memory Address
Maximum Clock Cycles:	To be defined

Description: Set the *RIP* (See ***Section 2.1.1. Instruction Pointer Register (RIP)***) register value to the memory address referenced by the source operand. If the source operand is a register, its value is considered the memory address to be set to *RIP*. This instruction has no effect is the *RCMP* bit 0 is set to 0 (See ***Section 2.1.9. Comparator Register (RCMP)***).

Opcode Example:

```
JMP 0x11EEFF                # 0x00000006 0x0011EEFF
JMP RGP8                    # 0x00004C06
```

4.7. CALL

Mnemonic:	CALL

Instruction Opcode:	0x00000007
Maximum Operands:	1
Minimum Operands:	1
Source Operand Type:	Register, Memory Address
Maximum Clock Cycles:	To be defined

Description: Set the *RIP* (See ***Section 2.1.1. Instruction Pointer Register (RIP)***) register value to the memory address referenced by the source operand. The memory address containing the instruction next to *CALL* is pushed into the address referenced by *RRA* (See ***Section 2.1.7. Return Address Register (RRA)***).

Opcode Example:

```
CALL 0xAACCBBDD             # 0x00000007 0xAACCBBDD
CALL RGP4                   # 0x00003C07
```

4.8. RET

Mnemonic:	RET
Instruction Opcode:	0x00000008
Maximum Operands:	0
Maximum Clock Cycles:	To be defined

Description: Set the *RIP* (See ***Section 2.1.1. Instruction Pointer Register (RIP)***) register value to the memory address referenced by the last pushed *RRA* address. This address is then popped from *RRA*. See ***Section 2.1.7. Return Address Register (RRA)***.

Opcode Example:

```
RET                         # 0x00000008
```

4.9. ARTH

Mnemonic:	ARTH
Instruction Opcode:	0x00000009
Maximum Operands:	2
Minimum Operands:	2
Source Operand Type:	Register
Target Operand Type:	Register
Maximum Clock Cycles:	To be defined

Description: Performs the arithmetic operation selected on *RARTH* (See ***Section 2.1.11. Arithmetic Register (RARTH)***) between the source and target operands. The result of the operation is stored at the target operand. *ARTH* instruction clears the *RARTH* bits 6 and 7 when it is executed.

Opcode Example:

```
ARTH RAL1, RAL3          # 0x00505809
ARTH RAL4, RAL2          # 0x005C5409
```

4.10. LGIC

Mnemonic:	LGIC
Instruction Opcode:	0x0000000A
Maximum Operands:	2
Minimum Operands:	2
Source Operand Type:	Register
Target Operand Type:	Register
Maximum Clock Cycles:	To be defined

Description: Performs the logic operation selected on *RLGIC* (See ***Section 2.1.10. Logic Register (RLGIC)***) between the source and target operands. The result of the operation is stored at the target operand.

Opcode Example:

```
LGIC RAL3, RAL2          # 0x0058540A
LGIC RAL4, RAL1          # 0x005C500A
```

4.1.11. INTR

Mnemonic:	INTR
Instruction Opcode:	0x0000000B
Maximum Operands:	1
Minimum Operands:	1
Source Operand Type:	Literal
Maximum Clock Cycles:	To be defined

Description: Causes a Software Interrupt. The source operand indicates the *Interrupt ID*. See ***Section 5. Interrupts*** for more information regarding this topic.

Opcode Example:

```
INTR 0x0A                # 0x00000A0B
INTR 0x01                # 0x0000010B
```

4.1.12. CEB

Mnemonic:	CEB
Instruction Opcode:	0x0000000C
Maximum Operands:	2
Minimum Operands:	2
Source Operand Type:	Register, Memory Address
Target Operand Type:	Register, Memory Address
Maximum Clock Cycles:	To be defined

Description: Performs a parallel processing of the instruction block starting at the memory address referenced by the source operand and ending at the memory address referenced by the target operand. If the

source and/or target operand is a register, its value is considered the memory address to start/end from/to.

Opcode Example:

```
CEB RGP1, RGP2              # 0x0030340C
CEB 0x4567, RGP3            # 0x0000380C 0x00004567
CEB 0xAA00, 0xAAFF          # 0x0000000C 0x0000AA00 0x0000AAFF
```

Notes: An *Illegal Instruction Fault* (See ***Section 6.1.6. Illegal Instruction Fault***) will occur if the target operand value is lesser than or equal to the value of the source operand, or if the specified range isn't aligned to 32-bits.

4.1.13. NOP

Mnemonic:	NOP
Instruction Opcode:	0x0000000D
Maximum Operands:	0
Maximum Clock Cycles:	To be defined

Description: Increments the *RIP* value by 4.

Opcode Example:

```
NOP                         # 0x0000000D
```

4.1.14. LTSK

Mnemonic:	LTSK
Instruction Opcode:	0x0000000E
Maximum Operands:	0
Maximum Clock Cycles:	To be defined

Description: Load the *Task Context* from the *Task Structure* (See

Section 8.1. Task Structure) from the memory address referenced by the register *RCT* and causes the *Current Task Context* to be saved at the memory address referenced by the register *RBT*. See ***Section 2.1.5. Task Registers (RBT and RCT)*** for more information regarding this topic.

Opcode Example:

```
LTSK                        # 0x0000000E
```

Notes: This is instruction may only be performed by code being executed at the *Privilege Level 0* (See ***Section 3.1. Privilege Level 0***) or a *Privilege Fault* (See ***Section 6.1.12. Privilege Fault***) will occur. This instruction has no effect if *RST* bit 2 is cleared (See ***Section 2.1.2. Status Register (RST)***).

5. Interrupts

Interrupts are asynchronous signals that indicate the CPU needs to handle data at some location in the system. *Interrupts* may be caused by hardware or by software, being the ones caused by hardware named *Hardware Interrupts* and the ones caused by software named *Software Interrupts*.

Software Interrupts are caused by the *INTR* instruction (See ***Section 4.1.11. INTR***) and are not affected by the *RST* bit 0 status (See ***Section 2.1.2. Status Register (RST)***). *INTR* instruction causes an immediate context switch to an *Interrupt Handler* (See ***Section 5.4. Interrupt Handling***). *Task Structures* (See ***Section 8.1. Task Structure***) pointed by *Task Registers* (See ***Section 2.5. Task Registers (RBT and RCT)***) will be updated and loaded depending on whether the *RST* bit 2 is set or cleared. If it is set, the *Task Structure* pointed by *RCT* will be updated with the current CPU context and the *Task Structure* pointed by *RBT* will be loaded as the current CPU context. The *RIP* (See ***Section 2.1.1. Instruction Pointer Register (RIP)***) address will still point to the *INTR* instruction address until all the requirements needed for the execution of such interrupt are verified and granted. This grants that any *Fault* (See ***Section 6. Faults***) occurring during the validation process will correctly identify the instruction that caused it.

Hardware Interrupts are caused by hardware devices, indicating that they need the attention of the CPU to handle data. As it happens with *Software Interrupts*, *Hardware Interrupts* also cause a context switch to an *Interrupt Handler* that may be or may not be *Trappable*. When an *Hardware Interrupt* occurs, it does not cause the cancellation of the current instruction being executed. After the current instruction is completed, the interrupt is triggered.

Trappable Interrupts are interrupts that does not ignore the *Interrupt Vector* (See ***Section 5.1. Interrupt Vector***) entry for their ID. This means that the *Interrupt Handler* may be customized by the Operating System. Note that the new configured *Interrupt Handler* for any given *Architecture-specific Interrupt* does not override any possible

internal CPU handling routines implemented for that interrupt. The new *Interrupt Handler* is always executed after this CPU handling routine, if any is internally implemented for that given interrupt.

An *Interrupt Handler* is a procedure that may be internally executed by the CPU (*Architecture-specific Interrupts*), or may be a user-defined procedure which the starting address of the code may be configured in the *Interrupt Vector* (*User-defined Interrupts*). See ***Section 5.1. Interrupt Vector***, ***Section 5.2. Architecture-specific Interrupts***, ***Section 5.3. User-defined Interrupts*** and ***Section 5.4. Interrupt Handling***.

5.1. Interrupt Vector

The *Interrupt Vector* is a memory structure that allows the operating system to configure Software Interrupt Handlers. The structure of the *Interrupt Vector* is composed by 127 elements of two 32-bit fields for *LEG32 Architecture* or one 64-bit field and one 32-bit field for *LEG64 Architecture*. The first field is a bit flag field and it is described in ***Table 5.1***. The second field is the memory address that points to the start of the handler code. The index of each element identifies the *Interrupt ID* for which the corresponding handler will be executed.

Bit	Description
0	Privilege Level (0 or 1)
1-31	Reserved

(Table 5.1. Interrupt Vector Entry Flags)

Interrupt Handlers configured in the *Interrupt Vector* with flag 0 cleared cannot be performed by *Privilege Level 1* (See ***Section 3.1. Privilege Level 1***) code. Attempts to do so will cause a *Privilege Fault* (See ***Section 6.1.12. Privilege Fault***) to occur.

The first 31 entries of the *Interrupt Vector* are reserved for *Architecture-specific Interrupts*. The entries ranging from 32 to 127 are reserved for *User-defined Interrupts*.

Customized *Interrupt Handlers* may be assigned to all *User-defined Interrupts* and for *Trappable Architecture-specific Interrupts*. The Operating System can perform such assignments through the *INTR 0x01* instruction. Non-*Trappable Architecture-specific Interrupts* ignore their corresponding entry at the *Interrupt Vector*. Reserved entries belonging to the range of *Architecture-specific Interrupts* are considered Non-*Trappable*, except for those in the entry ID range 21-31 which are reserved for implementation specific interrupts (See ***Section 1.6. Expansibility and Extensions***). All *User-defined Interrupts* are considered *Trappable*.

Interrupt Vector address is unmodifiable and its located at the *Memory Region* 0x000-0x3F8 (127 * 8 bytes) on *LEG32 Architecture* and 0x000-0x5F4 (127 * 12 bytes) on *LEG64 Architecture*. See ***Section 7.1. Reserved Memory Regions*** for more information regarding *Reserved Memory Regions*.

It is strongly recommended the reading of ***Section 4.1.11. INTR***, ***Section 5.2.1. Interrupt Vector Configuration Interrupt***, ***Section 5.3. User-defined Interrupts*** and ***Section 5.4. Interrupt Handling*** for more information regarding this topic.

5.2. Architecture-specific Interrupts

Architecture-specific Interrupts are interrupts internally handled by the CPU. Their behavior is not allowed to be modified. This means that there's no way to change the internal handler routine executed by the CPU. However, if the *Interrupt* is *Trappable* (See ***Section 5.1. Interrupt Vector***), a customized *Interrupt Handler* can be assigned to the interrupt that will be executed right after the internal handler routine termination.

Architecture-specific Interrupts handle software and hardware events, such as *Storage I/O* requests, *Display Output* requests, *Keyboard Input* requests, *Bad hardware* detections, *Interrupt Vector Configuration*, etc.

The list of *Architecture-specific Interrupts* are described in detail in the following subsections, from ***Section 5.2.1. Interrupt Vector Configuration Interrupt*** to ***Section 5.2.8. Timer Expiration Interrupt***.

5.2.1. Interrupt Vector Configuration Interrupt

ID:	0x01
Type:	Software Interrupt
Instruction:	*INTR 0x01*
Parameters:	*RGP1*, *RGP2*, *RGP3*
Required Privilege:	*Privilege Level 0*
Faults:	*Permission Fault,* *Bad Operation Value Fault*
Trappable:	No

Description: Setup a new handler routine which code starts at Memory Address pointed by *RGP3* value, with flags field defined by the value of *RGP2*, for the Interrupt ID defined by the value of *RGP1*. If *Privilege Level* for the task causing this interrupt is different than 0, *Permission Fault* will occur. If the *Interrupt ID* specified in *RGP1* is invalid, a *Bad Operation Value Fault* will occur. To disable a previously configured *Interrupt Handler* for a given *Interrupt ID*, a *INTR 0x01* instruction must be performed with *RGP2* and *RGP3* set to zero (0).

Notes: If a *Fault* occur, the fourth octet of *RFF* stores the *Interrupt ID* and the *RFF* bit 13 is set in order to indicate the occurring fault was caused by an *Interrupt* (See ***Section 2.1.3. Fault Flags Register (RFF)*** and ***Section 6.1.14. Interrupt Fault***).

5.2.2. Halt System Interrupt

ID:	0x03
Type:	Software Interrupt
Instruction:	*INTR 0x03*
Parameters:	-

Required Privilege:	*Privilege Level 0*
Faults:	*Permission Fault*
Trappable:	No

Description: Halts the CPU. If *Privilege Level* for the task causing this interrupt is different than 0, *Permission Fault* will occur.

Notes: If a *Fault* occur, the fourth octet of *RFF* stores the *Interrupt ID* and the *RFF* bit 13 is set in order to indicate the occurring fault was caused by an *Interrupt* (See ***Section 2.1.3. Fault Flags Register (RFF)*** and ***Section 6.1.14. Interrupt Fault***).

5.2.3. Keyboard Input Interrupt

ID:	0x09
Type:	Hardware Interrupt
Instruction:	-
Parameters:	RGP1
Required Privilege:	-
Faults:	Input/Output Operation Fault
Trappable:	Yes

Description: Indicates that the input from the keyboard occurred. The input data is stored at register RGP1. An *Input/Output Operation Fault* (See ***Section 6.1.13. Input/Output Operation Fault***) will occur if a failure receiving the keyboard data is detected.

Notes: If a *Fault* occur, the fourth octet of *RFF* stores the *Interrupt ID* and the *RFF* bit 13 is set in order to indicate the occurring fault was caused by an *Interrupt* (See ***Section 2.1.3. Fault Flags Register (RFF)*** and ***Section 6.1.14. Interrupt Fault***).

5.2.4. Display Output Interrupt

ID:	0x0A
Type:	Software Interrupt
Instruction:	*INTR 0x0A*
Parameters:	RGP1
Required Privilege:	Privilege Level 0
Faults:	Permission Fault, Input/Output Operation Fault
Trappable:	No

Description: Reads the least significant byte (see ***Section 1.2. Endianess***) from *RGP1* as an *ASCII* code and sends it to *Display Output*. If *Privilege Level* for the task causing this interrupt is different than 0, *Permission Fault* will occur. If an error occur while communicating with display output, a *Input/Output Operation Fault* will occur (See ***Section 6.1.13. Input/Output Operation Fault***).

Notes: If a *Fault* occur, the fourth octet of *RFF* stores the *Interrupt ID* and the *RFF* bit 13 is set in order to indicate the occurring fault was caused by an *Interrupt* (See ***Section 2.1.3. Fault Flags Register (RFF)*** and ***Section 6.1.14. Interrupt Fault***).

5.2.5. Storage I/O Interrupt

ID:	0x0B
Type:	Software Interrupt
Instruction:	*INTR 0x0B*
Parameters:	RGP1, RGP2, RGP3, RGP4, RGP5
Required Privilege:	Privilege Level 0
Faults:	Permission Fault, Bad Operation Value Fault, Bad Memory Reference Fault, Input/Output Operation Fault
Trappable:	No

Description: Performs an Input/Output operation on a storage

device identified by the *Storage ID* in the 2 least significant bytes of the *RGP1* value. The 2 most significant bytes of *RGP1* are a bit flag field for interrupt parameterization. ***Table 5.2*** describes the acceptable flags:

RGP1 bit	Description
16	Read Operation
17	Write Operation
18	Extended Offset

(Table 5.2. Storage I/O Interrupt RGP1 flags)

The *Extended Offset* flag indicates that the data offset to be accessed is beyond the value that a 32bit unsigned integer can represent. If this bit isn't set, *RGP2* holds the data offset value. If it is set, *RGP5* extends the data offset value for an additional 32bits, meaning that a 64bit data offset can be represented (the RGP1 bit 18 state and the register RGP5 is ignored on *LEG64 Architecture*). *RGP3* value indicates the length of the data to be read or written. *RGP4* value represents the *Memory Address* from where the data should be read in the case of a storage write operation, or the *Memory Address* to where the data should be written in the case of a storage read operation.

Notes: If a *Fault* occur, the fourth octet of *RFF* stores the *Interrupt ID* and the *RFF* bit 13 is set in order to indicate the occurring fault was caused by an *Interrupt* (See ***Section 2.1.3. Fault Flags Register (RFF)*** and ***Section 6.1.14. Interrupt Fault***).

5.2.6. Page Cache Invalidation Interrupt

ID:	0x0D
Type:	Software Interrupt
Instruction:	*INT 0x0D*
Parameters:	RGP1
Required Privilege:	Privilege Level 0
Faults:	Permission Fault
Trappable:	No

Description: Informs the CPU that the page identified by the *Base Physical Address* (See ***Section 7.4. Paging***) value set on RGP1 should be invalidated if it resides in the CPU page caching mechanism.

Notes: If a *Fault* occur, the fourth octet of *RFF* stores the *Interrupt ID* and the *RFF* bit 13 is set in order to indicate the occurring fault was caused by an *Interrupt* (See ***Section 2.1.3. Fault Flags Register (RFF)*** and ***Section 6.1.14. Interrupt Fault***).

5.2.7. Timer Configuration Interrupt

ID:	0x0F
Type:	Software Interrupt
Instruction:	INTR 0x0F
Parameters:	RGP1,RGP2,RGP3
Required Privilege:	Privilege Level 0
Faults:	Bad Operation Value Fault, Permission Fault
Trappable:	No

Description: Configure an internal CPU *Timer* (See ***Section 9. Timers***) identified by the *Timer ID* value on *RGP1*, with *Granularity* specified on *RGP2* and *Time to Expiration* value on *RGP3*. The granularity may be expressed in nanoseconds (*RGP2* bit 0 set), microseconds (*RGP2* bit 1 set), milliseconds (*RGP2* bit 2 set) or seconds (*RGP2* bit 3 set). The granularity defines the magnitude of the *RGP3* value. Note that *RGP2* flags are mutual exclusive. If more than one flag is set, a *Bad Operation Value Fault* will occur. This fault will also occur if the ID set on *RGP1* value is invalid. *Permission Fault* occur if a task or process running at *Privilege Level 1* attempts to setup a Timer.

Notes: If a *Fault* occur, the fourth octet of *RFF* stores the *Interrupt ID* and the *RFF* bit 13 is set in order to indicate the occurring fault was caused by an *Interrupt* (See ***Section 2.1.3. Fault Flags Register (RFF)*** and ***Section 6.1.14. Interrupt Fault***).

5.2.8. Timer Expiration Interrupt

ID:	0x10
Type:	Hardware Interrupt
Instruction:	-
Parameters:	RGP1
Required Privilege:	-
Faults:	-
Trappable:	Yes

Description: Indicates that the Timer identified by the RGP1 value has expired.

5.3. User-defined Interrupts

User-defined Interrupts are a set of configurable *Software Interrupts*. A process running at *Privilege Level 0* may configure an *Interrupt Handler* routine in the *Interrupt Vector* that will be executed when the instruction *INTR* is performed for that *Interrupt ID* as its operand (See ***Section 4.11. INTR***). These interrupts can be configured to be permitted to be executed by tasks or processes running on any *Privilege Level* of the CPU. If it is intended that a *User-defined Interrupt Handler Routine* shall be permitted for either *Privilege Level 0* and *Privilege Level 1* tasks or processes, its flags field at the *Interrupt Vector* shall indicate so. See ***Section 5.1. Interrupt Vector*** for more information regarding this topic.

There are 96 User-defined Interrupts available, ranging from *Interrupt Vector* entry 32 to entry 127.

See ***Section 5. Interrupts***, ***Section 5.1. Interrupt Vector*** and ***Section 5.4. Interrupt Handling*** for more information regarding this topic.

5.4. Interrupt Handling

Interrupt Handling is the capability to handle *Interrupts* through *Interrupt Handler Routines*. An *Interrupt Handler* is a procedure that take actions to properly handle an *Interrupt*. *Interrupt Handlers* may be either internal CPU routines or user-defined routines. See ***Section 5.2. Architecture-specific Interrupts*** and ***Section 5.3. User-defined Interrupts***.

For user-defined routines the *Interrupt Vector* should be properly configured to cause the CPU to execute the custom *Interrupt Handler* when a given *Interrupt* occurs. See ***Section 5.1. Interrupt Vector*** for more information regarding this topic.

Notes: It is important to note that initial communication with external hardware devices is mostly performed via *Interrupt Handlers* on *LEG Architectures*. The full understanding of *Interrupt* mechanism for *LEG Architecture* is required in order to implement efficient operating systems based on this architecture. See ***Section 5. Interrupts*** and all its sub-sections for detailed information regarding *Interrupts*.

6. Faults

Faults occur when the CPU detects a state that it is unable to handle by itself. Usually these states can be interpreted and handled by the operating system, forcing the CPU to return to a state where it is possible to continue its execution in a non-fault state. A routine that interpret a Fault state and force the CPU to return to a non-fault state is called a *Fault Handler*. *Fault Handlers* can be configured through the *RST* and *RFA* register (See ***Section 2.1.2. Status Register (RST)*** and ***Section 5.1.4. Fault-Handler Address Register (RFA)***).

If a *Fault* occurs and *RST* bit 1 is set, the CPU execution is halted. This means that the CPU won't recover from the *Fault State* until it is restarted.

The next section describes the possible *Fault States* (***Section 6.1. Fault States***) that can be detected by the CPU.

6.1. Fault States

Fault States are machine state conditions that the CPU cannot handle by itself and require operating system procedures to allow the CPU to recover from a *Fault State* to a non-*Fault State* to avoid a halt state. Unhandled faults will cause the CPU to halt its execution and will require a CPU restart.

The following sub-sections (from ***Section 6.1.1. Machine Check Fault*** to ***Section 6.1.14. Interrupt Fault***) describe the possible *Fault States* implemented in *LEG Architecture*.

6.1.1. Machine Check Fault

RFF bit:	0
Caused By:	Bad Hardware
Handleable:	No

Description: This state may occur when an internal CPU component or external hardware device that communicate directly with the CPU behaves unexpectedly, or when an unrecoverable fault state is detected. Two practical examples that may cause this fault to occur are the CPU being unable to communicate with system memory due to a system bus failure, or when the *Task Structure* pointed by *RBT* isn't a *Privilege Level 0* task. Please refer to ***Section 2.1.2. Status Register (RST)***, ***Section 2.1.5.1. Base Task Register (RBT)***, ***Section 3. Privilege Levels*** and ***Section 8.1. Task Structure*** for more information regarding this topic.

6.1.2. Bad Memory Reference Fault

RFF bit: 1
Caused By: An invalid memory reference was used as an operand of an instruction.
Handleable: Yes

Description: This state occur when an invalid memory reference is used as an operand on a memory operation instruction. A practical example that cause this state is the use of a memory reference beyond the available system memory or the use of memory reference that points to a *Reserved Memory Region* (See ***Section 7.1. Reserved Memory Regions***).

6.1.3. Bad Register Reference Fault

RFF bit: 2
Caused By: Invalid *Register ID* detected on an *Instruction Opcode*.
Handleable: Yes

Description: This state occur when a valid *Instruction Opcode* is interpreted, but one of its operands references an invalid *Register ID*. See

Section 4. Instruction Set for more information regarding *Register IDs* and *Instruction Opcodes*.

6.1.4. Bad Register Value Fault

RFF bit: 3
Caused By: Unexpected Register Value
Handleable: Yes

Description: This state occur when a CPU *Register* is set with an unexpected value. A practical example that cause this state is the use of reserved flags or an unspecified combination of flags on *Control Registers* (See ***Section 2.1. Control Registers***)

6.1.5. Bad Operation Value Fault

RFF bit: 4
Caused By: An unexpected value was detected while performing an operation.
Handleable: Yes

Description: This state occur when an unexpected operand references a value for an instruction that is unable to handle it. A practical example that cause this state is the *INTR 0x01* instruction with *RGP2* register set to some value greater than 127. This instruction will request a modification in the Interrupt Vector at an entry greater than 127. The Interrupt Vector only addresses 127 entries. See ***Section 4.1.11. INTR*** and ***Section 5.1. Interrupt Vector***.

6.1.6. Illegal Interrupt Fault

RFF bit: 5
Caused By: Illegal operand value for *INTR* instruction.
Handleable: Yes

Description: This state occur when an invalid Software Interrupt ID is used as operand value for the INTR instruction.

6.1.7. Illegal Instruction Fault

RFF bit: 6
Caused By: An unspecified opcode was loaded into *RIP* register.
Handleable: Yes

Description: This state occur when an unspecified instruction opcode referenced by *RIP* register value was tried to be executed. See ***Section 4. Instruction Set*** for more information regarding instruction opcodes.

6.1.8. Floating Point Unit Fault

RFF bit: 7
Caused By: An exception occurred during a *Floating Point Operation* at the FPU.
Handleable: Yes

Description: This state occur when an exception is detected while performing a *Floating Point Operation* in the *FPU*. A practical example that cause this state is an arithmetic operation involve RFP registers that attempts to divide a value by zero. See ***Section 1.3. Floating-Point Numbers*** for more information regarding *Floating Point Operations*.

6.1.9. Arithmetic Logic Unit Fault

RFF bit: 8
Caused By: An exception occurred during an *Arithmetic* or

Logic Operation at the ALU.

Handleable: Yes

Description: This state occur when an exception is detected while performing an *Arithmetic* or *Logic Operation* at the ALU. A practical example that cause this state is an arithmetic operation involving RAL registers that attempts to divide a value by zero.

6.1.10. Page Permission Fault

RFF bit: 9

Caused By: Operation not permitted on a *Logical Address Reference*.

Handleable: Yes

Description: This state occur when the CPU attempts to perform an operation that is not permitted over a *Logical Address* due to its *Page Permissions* (See ***Section 7.4.3. Page Permissions***) or an invalid combination of permission were used in the *Flags* field of *Page Structure* (See ***Section 7.4.1. Page Structure***). A practical example that cause this state is the attempt to write a read-only *Page* or a code execution attempt on a non-executable *Page*.

6.1.11. Page Fault

RFF bit: 10

Caused By: A *Page*, needed to translate a *Logical Address*, wasn't found

Handleable: Yes

Description: This state occur when a *Logical Address* is referenced by an memory handling operation and the CPU was unable to find a *Page* to translate it to a *Physical Address*. See ***Section 7.4.2. Address Translation*** for more information regarding this topic.

6.1.12. Privilege Fault

RFF bit: 11
Caused By: Code running at *Privilege Level 1* attempted a *Privilege Level 0* operation.
Handleable: Yes

Description: This state occur when a task or process running at *Privilege Level 1* attempts to perform an operation that is only permitted at *Privilege Level 0*. A practical example that cause this state is an attempt to execute *LTSK* instruction (See ***Section 4.1.14. LTSK***) with *RST* bit 3 set. See ***Section 2.1.2. Status Register (RST)*** and ***Section 3. Privilege Levels*** for more information regarding this topic.

6.1.13. Input/Output Operation Fault

RFF bit: 12
Caused By: An Input/Output Operation has failed due to communication problems with an external hardware device.
Handleable: Yes

Description: This state occur when an Input/Output operation have failed due to a communication problem while interacting with an external hardware device. This Fault may be caused due to an Interrupt that was performing an Input/Output operation. See ***Section 5.3. Keyboard Input Interrupt***, ***Section 5.4. Display Output Interrupt*** and ***Section 5.5. Storage I/O Interrupt*** for more information regarding this fault state.

6.1.14. Interrupt Fault

RFF bit: 13
Caused By: -
Handleable: -

Description: This special *RFF* flag does not indicate a *Fault State*, but that some *Fault State* set at *RFF* was caused while executing an *Architecture-specific Interrupt Handler*. The *Interrupt ID* for which the handler was being executed is stored at the fourth octet of *RFF* register. See ***Section 2.1.3. Fault Flags Register (RFF)*** and ***Section 5.2. Architecture-specific Interrupts*** for more information regarding this topic.

6.2. Fault Handling

Fault Handling is the capability to handle *Fault States* through *Fault Handler Routines*. A *Fault Handler Routine* is responsible to recover the CPU from a *Fault State* into a non-*Fault State*. *Fault Handling* is disabled by default when the CPU is initialized. To enable *Fault Handling*, the *RST* bit 1 shall be set (See ***Section 2.1.1. Status Register (RST)***).

LEG Architecture implements a *Control Register* called *RFA* (See ***Section 2.1.4. Fault-Handler Address Register (RFA)***) which value shall point to a *Logical* or *Physical Address*, depending on the *RST* bit 4 state (See ***Section 2.1.1. Status Register (RST)***), that contain an Operating System routine responsible for handling *Fault States*. The Operating System will be able to identify the *Fault State* through the *RFF* register (See ***Section 2.1.3. Fault Flags Register (RFF)***) and take appropriate actions to recover CPU to a non-*Fault State*.

If *Fault Handling* is enabled and no action is taken in the routine pointed by *RFA* for a occurred *Fault*, the CPU will enter in an endless loop if the Fault was caused by an instruction, since this instruction will be restarted after the *Fault Handler* routine completes.

It is important to note that the Operating System is responsible to clear *RFF* register after the *Fault Handler* completes all its procedures.

See ***Section 6. Faults*** and ***Section 6.1. Fault States*** for more information regarding these topics.

7. Memory Management

This section intends to describe the implemented features on *LEG Architecture* for memory management. It will also describe recommended practices and implementation choices that can be implemented in an Operating System intended to run on a *LEG Architecture* CPU.

A *LEG Architecture* CPU can access memory through two different addressing modes: *Physical Address Access* and *Logical Address Access*.

Physical Addresses are memory references that directly point to a system memory address, while *Logical Addresses* are *Virtual Memory Addresses* that need to be translated to their respective *Physical Address* through an *Address Translation* (See ***Section 7.4.2. Address Translation***) mechanism before the data can be accessed.

The Operating System may choose whether a task should use or not *Virtual Memory* by properly configuring the *RST* bit 4 (See ***Section 2.1.2. Status Register (RST)***). *Paging*, or *Virtual Memory*, is enabled on a per *Task Context* basis, which means that the Operating System is able manage tasks using *Logical Addressing* and tasks using *Physical Addressing* depending on the *RST* bit 4 status for each task. See ***Section 2.1.6. Paging Address Register*** and ***Section 7.4. Paging*** for more information regarding this topic.

7.1. Reserved Memory Regions

LEG Architecture specifies memory regions that cannot be accessed directly by the Operating System. Attempts to do so will generate a *Bad Memory Reference Fault* (See ***Section 6.1.2. Bad Memory Reference Fault***).

Usually these regions are configurable CPU data structures that require a specific *Instruction* or *Interrupt* to perform operations over its contents.

Table 7.1 specifies the *Reserved Memory Regions* that cannot be accessed directly:

From	To	Description
0x000	0x3F8	Interrupt Vector (*LEG32)*
0x000	0x5F4	Interrupt Vector (*LEG64)*

(Table 7.1. Reserved Memory Regions)

For more information regarding Interrupt Vector, please refer to ***Section 5.1. Interrupt Vector***.

7.2. Physical Address Space

Physical Address Space is specified as being the total addressable space of the total available system memory. A *Physical Address* value addresses a real position in the system memory. Instructions executed from an Operating System task with *Paging* disabled (RST bit 4 unset) that reference a memory address, will instruct the CPU to directly access that address in system memory. (See ***Section 2.1.2. Status Register (RST)*** and ***Section 7.4. Paging***)

Physical Address Space management does not implement a *Permissions* mechanism to permit or deny accesses for a given memory reference being referenced by an *Instruction*. In order to grant certain access *Permissions* to specific memory region, *Paging* must be enabled (See ***Section 7.4. Paging***).

7.3. Logical Address Space

Logical Address Space, also known as *Virtual Memory*, is an address space whose referenced memory addresses do not point directly to a *Physical Address*. They instead refer to a logical value that needs to be translated through an *Address Translation* mechanism that is implemented as part of the *Paging* mechanism (See ***Section 7.4. Paging***).

Logical Address Translations (See ***Section 7.4.2. Address Translation***) are internally performed by the CPU which loads the *Paging Address Register* (See ***Section 2.1.6. Paging Address Register (RPA)***) value and looks for *Page Structures* (See ***Section 7.4.1. Page Structure***) in order to identify which *Page* describes the referenced *Logical Address*. Once identified, the CPU translates that *Logical Address* into a *Physical Address*. After translation routine is completed, the *Physical Address* is accessed and the requested operation over its contents is performed.

If the CPU fails to find no *Page* describing the referenced *Logical Address*, a *Page Fault* occur (See ***Section 6.1.11. Page Fault***).

7.4. Paging

Paging is a mechanism that allows the translation of *Logical Addresses* into their respective *Physical Addresses* through an *Address Translation* (See ***Section 7.4.2. Address Translation***) mechanism, validating the *Page Permissions* (See ***Section 7.4.3. Page Permissions***) for that *Logical Address*, based on the operation being performed over it.

It permits the implementation of *Virtual Memory Management* approaches, bringing the possibility to map non-contiguous physical memory blocks into linear virtual address blocks. This can greatly reduce the complexity of the Operating System *Memory Management* mechanism and allow improved security since *Pages* implement a permission mechanism (See ***Section 7.4.3. Page Permissions***).

The following sub-sections (from ***Section 7.4.1. Page Structure*** to ***Section 7.4.3. Page Permissions***) will describe, in detail, how the *Paging* mechanism is implemented in *LEG Architectures* and how it behaves.

7.4.1. Page Structure

A *Page Structure* is a data structure that instructs how the CPU will

translate *Logical Addresses* into *Physical Addresses* (See ***Section 7.4.2. Address Translation***) and evaluate the permitted operations over those memory addresses.

A *Page Structure* maps a *Virtual Memory Region* (or *Logical Address Region*) into a *Physical Memory Region* by describing base pointers to *Logical* and *Physical Addresses*, the respective *Size* of the region, the permissions of the region, and two pointers that will point to the next and previous *Page* belonging to same *Task Address Space* (See ***Section 8.2. Task Address Space***).

The Operating System is responsible for creating and managing *Page Structures*. Each time a *Logical Address* is referenced, the CPU will try to find the correct *Page Structure* by first loading a Page Structure from the *RPA* (See ***Section 2.1.6. Paging Address Register (RPA)***) value and lookup for an entry that describes that *Logical Address* in order to evaluate its permissions and to correctly translate it to a *Physical Address*.

Page Structure for *LEG32 Architecture* is specified in ***Table 7.2***.
Page Structure for *LEG64 Architecture* is specified in ***Table 7.3***.
The *Flags* field options are specified in ***Table 7.4***.

Field	Size	Description
Base Logical Address	32bit	32-bit value for Base Logical Address
Base Physical Address	32bit	32-bit value for Base Physical Address
Size	32bit	Size of the memory region
Flags	32bit	Bit flag field
Next Page	32bit	Pointer to the next page
Previous Page	32bit	Pointer to the previous page

(Table 7.2. Page Structure for LEG32 Architecture)

Field	Size	Description
Base Logical Address	64bit	64-bit value for Base Logical Address
Base Physical Address	64bit	64-bit value for Base Physical Address
Size	64bit	Size of the memory region
Flags	64bit	Bit flag field
Next Page	64bit	Pointer to the next page
Previous Page	64bit	Pointer to the previous page

(Table 7.3. Page Structure for LEG64 Architecture)

Bit	Description
0	Read-Only Permission
1	Read/Write Permission
2	Executable Permission

(Table 7.4. Page Structure Flags Description)

Note that bits 0 and 1 of *Page Structure Flags* are mutual exclusive (See ***Section 7.4.3. Page Permissions***). If a *Page Flags* is found to have both bit 0 and bit 1 set, a *Page Permission Fault* will occur (See ***Section 6.1.10. Page Permission Fault***).

7.4.2. Address Translation

Address Translation is a mechanism used by the CPU to translate *Logical Addresses* into *Physical Addresses*.

When a Task, with *Paging* enabled (See ***Section 2.1.2. Status Register (RST)***), references a *Logical Address* in a memory operation, the CPU will try to find a suitable *Page*, through all the *Page Structures* defined for that task, that maps the memory region containing that *Logical Address*. This mechanism is known as *Page Lookup*. It is important to note that the Operating System is responsible to properly set the *RPA* register (See ***Section 2.1.6. Paging Address Register (RPA)***) value to

point to a valid *Page Structure* (See ***Section 7.4.1. Page Structure***) entry belonging to the current *Task Address Space* (See ***Section 8.2. Task Address Space***).

After a suitable *Page* is identified, the CPU will first evaluate the Permissions for that Page and grant that the requested operation can be performed. If this sanity check fails, a *Page Permission Fault* will occur (See ***Section 6.1.10. Page Permission Fault***). If the CPU is unable to find a suitable *Page* containing the referenced *Logical Address*, a *Page Fault* will occur (See ***Section 6.1.11. Page Fault***).

In order to identify if a *Page* is suitable for the translation (*Page Lookup*), the CPU load the *Page Structure* pointed by the *RPA* register and performs the following operations:

- Verify if the referenced *Logical Address* value is greater than the *Base Logical Address* value found in the current *Page Structure*.
- Add the *Size* value of the current *Page Structure* to the *Base Logical Address*.
- Verify if the referenced *Logical Address* value is lesser than the computed value in the previous step.
- Return success if the last condition is true.
- If the last condition is false, it loads the *Next* pointer as a *Page Structure* and performs all the previous operations again.

Being a suitable *Page* identified and the permissions verification successfully passed (See ***Section 7.4.3. Page Permissions***), the CPU will perform the following operations:

- Compute the offset of the referenced *Logical Address* from the *Base Logical Address* value found in the current *Page Structure*.
- Add that offset to the *Base Physical Address* value found in the *Page Structure*.
- Return the computed value as the translated *Physical Address*.

7.4.3. Page Permissions

Each *Page Structure* contains a field named *Flags* (See ***Section 7.4.1. Page Structure***) that allow the Operating System to configure the memory operations that may be performed over the memory region described by that *Page*.

Three permission options can be configured for a Page:

- Read-Only Permission
- Read/Write Permission
- Executable Permission

The *Read-Only Permission* and *Read/Write Permission* are mutual exclusive. If both are set, a *Page Permission Fault* (See ***Section 6.1.10. Page Permission Fault***) will occur. For information regarding *Page Permissions* configuration, please refer to ***Section 7.4.1. Page Structure***.

In order to evaluate if the requested operation over a *Logical Address* is permitted, the CPU performs the following steps:

- Evaluate if the operation is a Read, a Write, or an Instruction execution.
- If the operation is a Read, the flag Read-Only or Read/Write must be set. If none of these flags are set, a *Page Permission Fault* will occur.
- If the operation is a Write, the flag Read/Write must be set. If not set, a *Page Permission Fault* will occur.
- If the operation is an Instruction execution, the flag Executable must be set. If not set, a *Page Permission Fault* occur.
- By successfully passing all the previous sanity checks, the operation is granted.

The steps described above are known as *Permission Check* mechanism. They are performed after *Page Lookup* and before the *Address Translation*. If the permission check fails, *Address Translation* won’t be performed.

8. Multi-Tasking

LEG Architecture specifies two *Control Registers*, called *Task Registers* (See ***Section 2.1.5. Task Registers***), intended to allow the implementation of *Multi-Tasking* Operating Systems through a *Context Switching* mechanism (See ***Section 8.3. Context Switching***).

These registers are intended to point to *Task Structures* (See ***Section 8.1. Task Structure***). The memory addresses whose Task Registers point to are managed by the Operating System.

The *RBT* register (See ***Section 2.1.5.1. Base Task Register (RBT)***) should point to the *Task Structure* of Operating System kernel task, while the *RCT* (See ***Section 2.1.5.2. Current Task Register (RCT)***) should point to the *Task Structure* of the current user-land task.

When a *Context Switch* occur, *Task Registers* are internally evaluated by the CPU in order to load or save the current registers' states into the respective *Task Structure* the register points to. Refer to ***Section 8.3. Context Switching*** for more information regarding this topic.

8.1. Task Structure

A *Task Structure* is a data structure that allows the CPU to save *Task Contexts* whenever a *Context Switch* occur. This concept allows the Operating System to manage different tasks without losing their contexts whenever a *Context Switch* occur (See ***Section 8.3. Context Switching***).

The data structure that describes a *Task Structure* for *LEG32 Architecture* is specified in ***Table 8.1***. For *LEG64 Architecture*, the *Task Structure* is specified in ***Table 8.2***.

Field	Size	Description
Registers	112 bytes	Registers' contents. Size: 28 registers times 32-bits
Process ID	4 bytes	Unique task or process identifier

(Table 8.1. Task Structure Description for LEG32 Architecture)

Field	Size	Description
Registers	224 bytes	Registers' contents. Size: 28 registers times 64-bits
Process ID	4 bytes	Unique task or process identifier

(Table 8.2. Task Structure Description for LEG64 Architecture)

The Registers Field can be viewed as a sub-structure that describe all the implemented registers. The order in which the registers must be placed is: *RIP, RST, RFF, RFA, RBT, RCT, RPA, RRA, RSA, RCMP, RLGIC, RARTH, RGP1, RGP2, RGP3, RGP4, RGP5, RGP6, RGP7, RGP8, RAL1, RAL2, RAL3, RAL4, RFP1, RFP2, RFP3 and RFP4*.

Task Structures location in system memory must be managed by the Operating System. The Operating System shall then properly set *RBT* and *RCT* registers values to correctly point to the *Task Structures*. See ***Section 2.1.5. Task Registers (RBT and RCT)*** for more information regarding this topic.

8.2. Task Address Space

A Task Address Space is considered to be all the memory regions used by a task or process. This includes all the Heap, Stack and Code memory regions.

When *Paging* (See ***Section 7.4. Paging***) is disabled, this concept may be ignored. When *Paging* is enabled for a given task, the *Task Address Space* is then considered as all the *Page Structures* (See ***Section***

7.4.1. Page Structure) belonging to that task and at least one *Page Structure* must be pointed by the *RPA* register (See ***Section 2.1.6. Paging Address Register (RPA)***) in order to the CPU identify at least one *Page Structure* belonging to that task.

8.3. Context Switching

A *Context Switch* is considered to be an interruption of code execution on the current context by the CPU in order to handle another event that requires the immediate attention of the CPU.

Context Switching occur whenever an *Interrupt* (See ***Section 5. Interrupts***) or *Fault* (See ***Section 6. Faults***) occur or when a *LTSK* instruction (See ***Section 4.14. LTSK***) is executed. In order to disable *Context Switches* when an *Interrupt* or *Fault* occur, *RST* bit 2 (See ***Section 2.1.2. Status Register (RST)***) must be cleared.

Whenever a *Context Switch* occur, the *Task Structures* referenced by the *Task Registers* (See ***Section 2.1.5. Task Registers (RBT and RCT)***) are loaded or updated by the CPU, depending on what caused that *Context Switch*. The operations performed by the CPU when a *Context Switch* occur are described below:

- The *Task Structure* pointed by *RBT* is loaded whenever a *Fault* or *Interrupt* occur and it is updated whenever a *LTSK* instruction is executed.
- The *Task Structure* pointed by *RCT* is loaded whenever a *LTSK* instruction is executed and it is updated whenever a a *Fault* or *Interrupt* occur.

Context Switching is an important part of the Operating System *Task Management* mechanism. It is strongly recommended, when implementing *Task Management* mechanism for an Operating System based on *LEG Architecture*, the full understanding of the ***Section 8. Multi-Tasking***, all its sub-sections and all the recommended sections or sub-sections that may be referenced.

9. Timers

LEG Architecture specifies eight (8) internal CPU Timers that can be used for time elapsing. These *Timers* can only be configured at *Privilege Level 0* (See ***Section 3.1 Privilege Level 0***). Attempts to setup a Timer at *Privilege Level 1* will cause a *Privilege Fault* (See ***Section 6.1.12. Privilege Fault***) to occur.

The following sub-sections (Section ***9.1. Timer Parameters*** and ***Section 9.2. Timer Configuration***) will describe the available timer parameters and how to setup them.

9.1. Timer Parameters

A *Timer* is data structure with three (3) elements: *Timer ID*, *Granularity* and *Time to Expiration* (*TTE*).

The *Timer ID* identifies the timer. Valid *Timer ID* values range from 0 to 7.

Granularity defines the magnitude of the *TTE* value. It can represent nanoseconds, microseconds, milliseconds and seconds. ***Table 9.1*** describes the implemented Granularity Flags for each available option.

Granularity Flag	Description
0x01	Nanoseconds (ns)
0x02	Microseconds (us)
0x04	Milliseconds (ms)
0x08	Seconds (s)

(Table 9.1. Granularity ID values)

Time to Expiration is the amount of time that will elapse since the Timer is activated, until it expires and causes a *Timer Expiration Interrupt* (See ***Section 5.2.8. Timer Expiration Interrupt***).

9.2. Timer Configuration

Timers are configured through a *Timer Configuration Interrupt*. Detailed information regarding this topic can be found on ***Section 5.2.7. Timer Configuration Interrupt***.

If an invalid value of *Timer ID* or an invalid *Granularity Flag* is used while configuring the *Timer*, a *Bad Operation Value Fault* will occur (See ***Section 6.1.5. Bad Operation Value Fault***).

VI. Appendixes

Appendix A – Bootloader Example

For the purpose of illustration of how to develop a basic bootloader on *LEG Architecture*, it is assumed that there's an external controller that reads the bootloader binary data from a storage device and loads it into system memory at the first memory address available that doesn't belong to a *Reserved Memory Region* (See ***Section 2.1.1. Instruction Pointer Register (RIP)*** and ***Section 7.1. Reserved Memory Regions***).

Being that granted and assuming that the bootloader binary data length doesn't exceed 2048 bytes, the following assembly code will load an Operating System kernel, which size is described by a 32-bit integer value stored at the address 0x800 on the system device identified as Storage ID 0, into the system memory address 0x1000 and perform a *JMP* (See ***Section 4.6. JMP***) instruction into this address after the bootloader process completes:

```
1. .start:
2.     cpvl 0x00, rst          # Disabled: Interrupts,FH,Tasking,Paging
3.     cpvl 0x10000, rgp1      # I/O Read from storage ID 0
4.     cpvl 0x800, rgp2        # Start Read at 0x800
5.     cpvl 4, rgp3            # Read 4 bytes
6.     cpvl 0x1000, rgp4       # Store at memory address 0x1000
7.     intr 0x0B               # Perform Storage I/O read
8.     cpr  rpg4, rgp3         # Load kernel size
9.     cpvl 0x804, rgp2        # Start read kernel binary at 0x804
10.    intr 0x0B               # Perform Storage I/O read
11.    cpvl 0x01, rcmp         # Enable JMP
12.    jmp  0x1000             # Start kernel code execution
```

Appendix B – Debugging Techniques

Although *LEG Architecture* does not specify debugging mechanisms, there are several techniques that can be used in order to do so.

Breakpoints are possible to be implemented by configuring a User-Defined Interrupt (See ***Section 5.3. User-Defined Interrupts***). A debugger can use the instruction that causes this interrupt to occur to replace the instruction where the breakpoint is required and the interrupt handler may be used to perform the analysis. After the analysis is completed, the interrupt handler shall replace the interrupt instruction with the original instruction and correctly adjust *RIP* (See ***Section 2.1.1. Instruction Pointer Register***) address in order to point to it.

More complex and robust mechanisms may be implemented for Multi-Tasking Operating Systems by implementing one or more system calls for debugging purposes.

Appendix C – Multi-Tasking Process Scheduler

The implementation of a simple Multi-Tasking Process Scheduler may be accomplished by using the Multi-Tasking mechanisms available on *LEG Architecture* (See ***Section 8. Multi-Tasking***) along with properly configured Timers (See ***Section 9. Timers***).

By configuring a *Timer* each time a *LTSK* instruction is used (See ***Section 4.14. LTSK***), it is possible to grant the amount of CPU time that is allocated to a specific task. A process scheduler in a round-robin fashion, may perform the following steps:

- Load *RCT* with the address of the next *Task Structure*.
- Setup a Timer with the amount of time the task should be running.
- Perform a *LTSK* instruction.
- When the *Timer* expires, a *Timer Expiration Interrupt* is caught.
- The interrupt handler informs the Process Scheduler.
- Repeat all the steps above.

This simplified algorithm allocates the same amount of processing time to every existing task.

For more information regarding this topic, please read the recommended sections referenced above and also the ***Section 2.1.5. Task Registers (RBT and RCT)*** and ***Section 5. Interrupts***.

Appendix D – Interrupts and Context Switching

Interrupt Handling must be designed properly in order to efficiently work on a Multi-Tasking Operating System, avoiding multiple context switches on the same task context.

To avoid interrupt occurrence inside an interrupt handler, it is required the interrupt handler code to disable interrupt handling at its very first instruction, by clearing the *RST* bit 0.

Also, if a fault occur during the interrupt handler execution, one should avoid a new context switch to *RBT Task Structure* when the current context is already the *Base Task Context.* It is recommended that the *Base Task* have the context switches disabled during the interrupt handler execution by clearing the *RST* bit 2.

By implementing this approach, it is granted that interrupts only occur on User-Space tasks, causing the desired context switch to *RBT Task Structure*. This approach also grants that if a fault occur inside an interrupt handler routine, a context switch won't occur.

Note that the context switching should be enabled by setting the *RST* bit 2 before the *LTSK* instruction is called, as this instruction will take no effect if task registers are disabled.

For a better understanding of this approach, it is strongly recommended the complete read and understanding of ***Section 2.1.2. Status Register (RST)***, ***Section 2.1.5. Task Registers (RBT and RCT)***, ***Section 4.1.14. LTSK***, ***Section 5. Interrupts*** and ***Section 8. Multi-tasking***.