# Preface

## Who This Book is For

This book is for the enthusiast who’s looking for an instruction set architecture (ISA) suitable for more complex and practical projects; an ISA capable of running a modern operating system. The ISA presented here is one similar to the Table888 ISA and it’s advisable that one have a read of the book Table888. It’s advisable that one have a fairly good background in digital electronics and computer systems before attempting a read. Examples are provided in the Verilog language, it would be helpful to have some understanding of HDL languages. The Table888 book looks at a number of design choices for Table888 in the context of what is known about computer architecture. This book (the Scarerob-V ISA) simply presents the Scarerob-V ISA without reviewing the architectural choices.

## Motivation

Table888 ISA is a reasonable starting place for a homebrew processor. The focus of Table888 was a processor design in order to support a simple compiler and tool chain, which results in a minimal amount of RTL code. Another motive behind Table888 was an easy to read instruction layout. The premise behind Table888 was making it easy understand and easy to implement. The RTL code for the processor itself is largely included in the text of the book.

While the simplicity of Table888 makes a good starting point, there is room for improvement of code density and performance at the cost of complexity and readability.Table888 may be good for embedded homebrew projects; I got it as far as reading files from a FAT file system using the Table888 tool chain. However, I desire to run a modern operation system such as Linux or MINIX on a 64 bit ISA. The ISA requirements for a modern operating system mandate an ISA more complex than that available with Table888. So the choice was between extending the Table888 or starting with a new design. The typical operating system has at least two privilege levels (user and supervisor) and uses some form of memory management. These features are not present in Table888 and so have to be added. Rather than just continue to extend the Table888 project a new project Scarerob-V has been started. Starting a new project allows the Scarerob-V to be significantly different than Table888 while starting from the same point.

The focus therefore of the Scarerob-V ISA is one of more complexity and greater practical usability. This necessitates a more complex tool chain, and a loss of readability.

## About the Author:

First a warning: I’m an enthusiastic hobbyist like yourself, with a ton of experience. I’ve spent a lot of time at home doing research and implementing several soft-core processors, almost maniacally. One of the first cores I worked on was a 6502 emulation. I then went on to develop the Butterfly32 core. Later the Raptor64. I have about 20 years professional experience working on banking applications at a variety of language levels including assembler. So I have some real world experience developing complex applications. I also have a degree in electronics engineering technology. Some of the cores I work on these days are really too complex and too large to do at home on an inexpensive FPGA. I await bigger, better, faster boards yet to come.

# Design Choices / Differences from Table888

The focus of the Scarerob-V ISA is one of more complexity and greater practical usability. The Scarerob-V ISA has enhancements over Table888’s ISA in order to support a modern operating system.

Why not just extend Table888 ? Well the Scarerob-V ISA is a kind of extension to the Table888’s ISA. A number of the instructions make use of the same opcodes. But Table888’s use of a fixed 40 bit instruction size is something altered by the Scarerob-V ISA.

## Instruction Size

For reasons of simplicity Table888 works with only 40 bit instructions. Limiting the instruction set to 40 bits results in some loss of code density. Many instructions do not require the full 40 bit encoding and could be implemented with fewer than 40 bits. In particular compares and branches, which make up a significant portion of the instruction stream, often don’t require anywhere near 40 bits to represent the desired operation. By contrast, the Scarerob-V ISA uses variable length instructions to significantly improve code density. Most instructions have 24-bit forms, about half the size of a Table888 instruction. This should result in better memory performance as the processor has to move fewer bytes per instruction executed.

## Register Arrays

Table888 uses a single 256 entry register set for simplicity, this is consistent with many RISC designs. The single register set is used to contain several different types of data including data and code target addresses, return addresses and flag values. The Scarerob-V ISA uses a single general purpose register array with 64 entries which is portioned into sections with specific purposes. The smaller register array helps with code density because it doesn’t take as many bits to represent registers in instructions. There is no significant loss in performance by using a smaller register array; it is simply more difficult to read off the registers when looking at machine code. The general register array is partitioned into flags registers, general purpose registers and code address registers. The flags array takes the place of several general purpose registers used in the Table888 design to store flag results by convention. The benefit of this is that only three bits rather than eight bits are required to specify a flag register. This makes it possible to encode a branch instruction using only 16 bits. A test instruction is also encodable in 16 bits. It’s a lot more compact than the 40 bits required by the Table888.

## Privilege Levels

The Scarerob-V ISA is a 16 level privilege system. 0 is the highest privilege level, 15 is the lowest. Some operations and registers are available only to higher privilege levels. Most modern systems are two level – supervisor and user modes. The x86 series has four privilege level of which only two are commonly used. If desired the Scarerob-V ISA can be used as a two level system (privilege levels 0 and 15). It is easy to use only two levels of a multi-level system, but quite hard to add multi-levels to a system that only supports two.

**General Purpose Register Array**

The ISA contains 64 general purpose registers (r0-r63). Several of the registers are reserved for predefined purposes. Instructions in the instruction set may make implicit use of several of the registers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Num. | Usage Convention | Num. | Usage Convention | |
| r0 | always zero | r32 |  | |
| r1 | first subroutine return value / argument | r33 |  | |
| r2 | second subroutine return value / argument | r34 |  | |
| r3 | subroutine argument | r35 |  | |
| r4 | subroutine argument | r36 |  | |
| r5 | subroutine argument | r37 |  | |
| r6 |  | r38 |  | |
| r7 |  | r39 |  | |
| r8 |  | r40 |  | |
| r9 |  | r41 |  | |
| r10 |  | r42 |  | |
| r11 |  | r43 |  | |
| r12 |  | r44 |  | |
| r13 |  | r45 |  | |
| r14 |  | r46 |  | |
| r15 |  | r47 |  | |
| r16 |  | r48 | flag 0 | These registers are identified by a three bit register code in compare and branch instructions. |
| r17 |  | r49 | flag 1 |
| r18 |  | r50 | flag 2 |
| r19 |  | r51 | flag 3 |
| r20 |  | r52 | flag 4 |
| r21 |  | r53 | flag 5 |
| r22 |  | r54 | flag 6 |
| r23 |  | r55 | flag 7 |
| r24 |  | r56 |  | |
| r25 |  | r57 |  | |
| r26 |  | r58 |  | |
| r27 |  | r59 | task register (associated with TSS) | |
| r28 |  | r60 | base pointer (assoc. SS) | |
| r29 |  | r61 | prev stack pointer | |
| r30 |  | r62 | stack pointer (assoc. SS) | |
| r31 |  | r63 | program counter alias (Assoc CS) | |

r61 – This register is the stack pointer for the previous privilege level.

r62 – This register is the stack pointer for the current privilege level.

r63 – This register is an alias for the program counter. It contains the address of the current instruction and is a read only register. It may be used in relative address formation.

## Flag Registers

The programming model contains eight flag registers which are also part of the general purpose register array. Flag registers are set by the compare and test instructions and are used for conditional branching. The flag register to use is specified by a three bit code in the instruction.

|  |  |  |
| --- | --- | --- |
| 3bit code | GPR | FlagR |
| 000 | r32 | f0 |
| 001 | r33 | f1 |
| 010 | r34 | f2 |
| 011 | r35 | f3 |
| 100 | r36 | f4 |
| 101 | r37 | f5 |
| 110 | r38 | f6 |
| 111 | r39 | f7 |

Each flag register holds four result status bits. These bits are the classic C,V,N, and Z bits.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 | 62 | 61 2 | 1 | 0 |
| N | V | ~ | Z | C |

## Status Register (SR - SPR07)

This register contains bits that control the overall operation of the processor or reflect the processor’s state. Bits are included for interrupt masking, and current operating privilege level. This register is split into two halves with both halves having the same format. The lower half of the register is what determines how the processor works. The upper half of the register maintains a backup copy of the lower half for interrupt processing. There are instructions provided for manipulating the interrupt mask.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31..16 | 15 | 14 | 13 | 12 | 11..4 | 3..0 |
| same format as 15..0 | Interrupt Mask | Reserved | Reserved | Float Except. Enable |  | Current Privilege Level |
|  | IM | ~ | ~ | FXE | ~ | CPL |

 IM = interrupt mask

Maskable interrupts are disabled when this bit is set.

# Additional Registers

These registers are accessible with the MFSPR and MTSPR instructions.

## Page Table Address (SPR 04 or CR3)

This register is described fully in the section on paged memory management. This register contains the base address of the page table in memory.

|  |  |  |
| --- | --- | --- |
| Address63..13 | ~10 | TS3 |

## Control Register Zero (SPR 05 or CR0)

This register contains bits to enable paged memory management and protected mode.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 | 62 9 6 1 | | | | | | | | 0 |
| Pg | Ce |  |  |  |  |  | BXWR |  | Pe |

Pg: Paging enable bit 1=enabled, 0 = disabled

PE: Protected Mode enable: 1 = enabled, 0 = disabled.

Ce: cache enable 1 = enabled, 0 = disabled

BXWR: triple mode redundancy bits

X: TMR on execute

W: TMR on writes

R: TMR on reads

## Clock Register (SPR 06)

The clock register controls clock gating to the processor to allow lower power consumption. Gating is controlled with a bit pattern which is fed to a clock enable gate. The pattern is 50 bits long, allowed clock control (or power control) in 2% increments. For example loading the register with h2AAAAAAAAAAAA will cause every other clock to be gated off, reducing the effective operating frequency of the core in half. Loading the register with a zero will stop the clock completely. However, a non-maskable interrupt or reset will reload the clock register with all ones, causing the processor to operate at maximum frequency.

|  |  |
| --- | --- |
| 63 50 | 49 0 |
| ~14 | clock gating pattern49..0 |

## Fault PC (SPR 08)

This register contains the instruction address at which a fault occurred.

|  |
| --- |
| 63 0 |
| Program Counter63..0 |

## SRAND1 (SPR16) and SRAND2 (SPR17)

This pair of registers contains the seed for the random number generator. They should be set to a non-zero value to generate random numbers. Using the same seeds results in the same sequence of random numbers generated.

## RAND (SPR 18)

This register contains a random value. The value is the latest value generated by a ‘GRAN’ (generate random number) instruction.

## SEGx (SPR 32 to 47)

These are the segment registers which are more fully described under the section on segmentation.

## Segmentation and Paging

Segmentation and paging are the two main choices for memory management beyond some simpler mechanisms like bank switching. The goal for Scarerob-V is to implement both a segmented and a paged system. Several commercial processors implement both segmentation and paging. Although segmentation has fallen out of favor somewhat it is still used. Typically the segmentation part of the cpu has a handful of segment registers loaded with a flat memory model, then is for the most part ignored.

### Segmentation Overview

As part of the memory management portion of a cpu segment registers are often provided. There are usually multiple segment registers in order to support multiple segments which are typically part of a program. Common program segment are: the code segment, the data segment, the uninitialized data segment and the stack segment. There are often other segments as well. 80x88 is famous for its segment registers, but other processors like IBM’s PowerPC also use them as well. Segment registers are a fairly easy to understand , and a low cost memory management approach. The memory address from an instruction is added to a value from a segment register in order to form a final address. The segment register is often shifted left as it is added in order to allow a greater physical memory range than the range directly supported by the architecture. Segment registers allow programs to be written as if they had specific memory addresses available to them, such as starting at location zero, while in reality the actual physical address of the program is much different.

One of the unfortunate consequences of using a segmented memory management system is that the segmentation model tends to be visible in the programming model for the cpu. Typically provisions must be made in the ISA in order to support a segmented system. For instance the x86 has numerous instructions for manipulating the segments including segment prefixes and near and far calls.

### Address Formation:

The virtual address is added to a segment base register in order to form a final address. Note that there is no shift associated with the segment addition in this case.

|  |
| --- |
| Virtual Address |
| + |
| segment base register |
| = |
| Segmented address |

### Paging Overview

Paging uses a set of tables to perform mapping of virtual addresses to physical ones. Unlike segmentation, paging cannot resolve maps right down to individual bytes. Instead memory is broken up into a number of pages and managed on that basis. A typical page size is 4kB. The virtual address is divided up and each part of the virtual address is used to index into a table.

The table at the highest level of the hierarchy is usually permanently resident in the computer’s memory for performance reasons. Because there is a fair amount of work to be done in mapping addresses, address mappings are usually cached in an additional until called a translation look-aside buffer (or TLB). This unit is also sometimes called an Address Translation Cache (ATC). A paging system tends to have more overhead associated with it compared to a segmented system.

# Segmentation

## Number of Registers:

The number of segment registers that are useful seems not to be quantified as closely as the number of general purpose registers. However, four registers was deemed not enough for the 80x86 architecture and two more segment registers were added. Also a couple of additional registers in the 80x86 design were added to support the segmentation architecture and they act a lot like segment registers. These include the task register and the local descriptor table register. So we have about eight segment registers in the 80x86 architecture. The PA-RISC uses an array of eight “Space Registers” which are roughly analogous to segment registers.

The Scarerob-V ISA has an array of sixteen segment registers available.

A number of segment registers are dedicated to specific uses and take the place of dedicated registers found in other architectures.

## Segment Usage Conventions

Segment register #15 is the code segment (CS) register. All program counter addresses are formed with the code segment register unless the upper 48 bits of the address are zero in which case the code segment is ignored. If the program counter alias register is used in a load / store instruction, then the code segment register is used as the base by default.

Segment register #14 is the stack segment (SS) register by convention. The stack segment is used for operations involving the stack, PUSH, POP, JSR, or RTS operations. If the base pointer or stack pointer registers are specified in a load / store instruction then the default is to use the stack segment as a base.

Segment register #1 is the data segment (DS) by convention.

Segment register #13 is the volatile data segment (VDS). Addresses formed using this segment register bypass the data cache.

Segment register #12 is task state segment register. This segment register points the task state save area. If the task register is used by a load / store instruction then the task state segment is used by default for address formation. It’s 80x86 equivalent is the task register.

Segment register #11 is Scarerob-V’s equivalent to the LDT register in the 80x86 series processors.

#### Segment Registers

|  |  |  |  |
| --- | --- | --- | --- |
| Num |  | Long name | Comment |
| 0 | NS | NULL segment | by convention contains zero |
| 1 | DS | data segment | by convention |
| 2 | TLS | thread storage | by convention |
| 3 | BSS | BSS segment | by convention |
| 4 | RS | read only segment | by convention |
| 5 | ES | extra segment | by convention |
| 6 | GS | global storage | by convention |
| 7 | seg7 |  | unassigned |
| 8 | seg8 |  | unassigned |
| 9 | seg9 |  | unassigned |
| 10 | seg10 |  | unassigned |
| 11 | LDT | local descriptor table | used by hardware to look up local descriptors |
| 12 | TSS | task state segment | associated with tr |
| 13 | VDS | volatile data segment | bypasses the cache (useful for I/O) |
| 14 | SS | Stack segment | by convention (associated with sp, bp) |
| 15 | CS | Code segment | always used for code addressing |

## Composition of Segment Registers

Segment registers are comprised of a program visible portion and an invisible portion. The visible portion of the segment register is a 24 bit selector. The invisible portion is a 256 bit register containing bounds and access rights information. The selector is used to access a segment description entry in the global or local segment descriptor tables. The invisible portion of a segment register is cached in the processor to improve performance.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | Visible portion of segment register -> | | | RPL4 | TI | Index19 | | |
|  |  |  |  | |  |  | |  |  | |
| h7 | Base127..96 | | | | | | | | | |
| h6 | Limit95..64 | | | | | | | | | |
| h5 | Base95..64 | | | | | | | | | |
| h4 | Limit63..32 | | | | | | | | | |
| h3 | Base63..32 | | | | | | | | | |
| h2 | Limit31..4 | | | | | | | | | F4 |
| h1 | Base31..4 | | | | | | | | | 04 |
| h0 | ACR8 | | DPL4 | ~20 | | | | | | |

The segment base and limit are 16 byte aligned in order to prevent address alignment problems with load and store operations.

Selector Format:

|  |  |  |
| --- | --- | --- |
| RPL4 | TI | Index19 |

RPL: request privilege level

TI: table indicator 0=global, 1=local

Index: index into the global or local descriptor table

Whenever the selector value is loaded into a segment register, the entire segment descriptor from the global or local table is loaded into hidden registers. The descriptor controls the accessibility and the location of the segment in memory.

### Register Access

Segment register access takes place in parallel with normal general purpose register access.

### Moving Register Values

Segment registers may be only moved to or from general purpose registers. There are no other instructions for manipulating a segment register. MTSPR is used to move a general purpose register to the segment register. MFSPR moves a segment register into a general purpose register.

Note that the only way to change the code segment is by executing an interrupt. An interrupt or the break instruction will load the code segment from a value in the interrupt vector table. The other instruction which loads the code segment register is the RTI instruction. Unlike some other architectures there are no far call / far return instructions.

## Merge Format:

It would be wasteful and time consuming to store the selector and program counter offset as separate words on the stack when a subroutine is called, when it isn’t needed most of the time. It would take two bus cycles for every subroutine call when really only a single cycle is needed. So usually the selector and offset are merged together into a single 64 bit format. Most of the time a program will consume far less than 38 bits of address space, meaning the upper bits of the program counter are zero. This fact can be used to merge the 24 bit selector into the upper address bits while storing a return address on the stack. When a subroutine return is done, the selector and program counter are split apart. In order to be able to use more than 38 bits for the program counter, a pair of far call, far return instructions could be used to store the selector and offset separately on the stack. Alternately, a bit flag can be stored on the stack to indicate that two words were used to store the selector and offset separately. Scarerob-V uses the alternate mechanism.

Return address, long format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| w1 | ~2 | Code Segment Selector24 | | ~38 |
| w0 | 10xx4 | | Program Counter59..38 | Program Counter37..0 |

Return address, short format with selector:

|  |  |  |  |
| --- | --- | --- | --- |
| w0 | 012 | Code Segment Selector24 | Program Counter37..0 |

Return address, short format without selector:

|  |  |  |  |
| --- | --- | --- | --- |
| w0 | 00xx4 | Program Counter59..38 | Program Counter37..0 |

### System Calls

The system call instructions (BRK, SWE) store the program counter, code segment and status register on the stack in a long address format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| w1 | ~2 | Code Segment Selector24 | | ~6 | Status Register32 |
| w0 | 11xx4 | | Program Counter59..38 | Program Counter37..0 | |

### Subroutine Calls

Calling a subroutine causes the selector and current program counter to be merged together and stored on the stack, as mentioned above.

In addition to a subroutine call which does not change the code segment, there is a prefix instruction provided in order to change the code segment. The format of this prefix and following subroutine call are shown below:

With Selector Prefix:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| RPL4 | TI | Index19 | 6F8h | JSP |
| Address24 | | | 728h | JSR address |

The prefix also works with the jump instruction.

### Call Gates

Call gates are used to switch to a higher level of privilege. Up to 31 word parameters may be copied from the caller’s stack to the callee’s stack.

Call gates have fallen into disuse in the 80x86 processors, in favor of sysenter/sysexit instructions which execute much faster.

#### Call Gate Descriptors

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| h7 | unused32 | | | |
| h6 | unused32 | | | |
| h5 | unused32 | | | |
| h4 | unused32 | | | |
| h3 | Offset63..32 | | | |
| h2 | Offset31..0 | | | |
| h1 | ~8 | Selector23..0 | | |
| h0 | ACR8 | DPL4 | ~15 | nCopy5 |

# Hardware Interrupts

Interrupt Descriptor Table Entries

The interrupt descriptor table contains only two types of gates (interrupt, or trap). Interrupt descriptor table entries are sixteen bytes in size with the following format:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| w1 | P | ~ | ~ | 0 | 0 | 1 | 1 | T | ~32 | Code Segment Selector24 |
| w0 | DPL4 | | | | Offset59..0 | | | | | |

T = 1: Trap Gate

T = 0: Interrupt Gate

The difference between a trap and an interrupt is that an interrupt automatically masks further interrupts from happening. A trap does not affect the interrupt flag.

The return from interrupt (RTI) instruction uses a long format to load both the program counter and status register from the stack. This format includes the code segment selector and is as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| w1 | ~2 | Code Segment Selector24 | | ~6 | Status Register32 |
| w0 | 11xx4 | | Program Counter59..38 | Program Counter37..0 | |

# Scarerob-V Paging System

One of the nice features of paging is that it is almost invisible from a software perspective. There aren’t any registers like segment registers, to worry about when paging is active. Paging simply works behind the scenes. The following section describes the paging system used by the Scarerob-V system.

The paging system is capable of mapping the entire 64 bit address space. A multi-level system of page directories and subdirectories is used. In most cases the address space mapped will be less than a full 64 bit address space. The paging system accommodates this by using a smaller directory hierarchy. For instance, if an application is less than 1GB in size, a two level page system is used. If the application can fit within 2MB only a single level is required. The depth of the directory system is controllable on an application basis. The page memory management unit takes care of walking the page tables in hardware in order to find a translation. Translations are stored in a translation look-aside buffer (TLB) which is a translation cache, so that the page tables don’t have to be walked for every translation.



## Registers:

The primary register that controls paging is the page table address register (PTA) or as it is alternately called control register number three. (CR3). This register contains the base address of the root page table in memory. Once the PTA (page table address) is set, the processor knows where to begin looking up virtual to physical address translations.

### Moving Register Values

Control registers for paging may be only moved to or from general purpose registers. There are no other instructions for manipulating a control register. MTSPR is used to move a general purpose register to the control register. MFSPR moves a control register into a general purpose register.

**Page Directories / Subdirectories**

Page directories are 4kB in size and contain 512, 8 byte entries.

**Page Directory Entries (PDE’s)**

The page directory entries identify where in memory further subdirectories or translation pages are.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Address63..12 | ~8 | d1 | a1 | ~1 | t1 |

**Page Tables**

Page tables are 8kB in size and contain 512, 16 byte entries.

**Page Table Entries (PTE’s)**

Page table entries map the virtual address to a physical one.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Address63..12 | | | | | | | | | ~8 | | | | | d1 | | a1 | | e1 | | t1 |
| p15 | p14 | p13 | p12 | p11 | p10 | p9 | p8 | p7 | | p6 | p5 | p4 | p3 | | p2 | | p1 | | p0 | |

a1: accessed

d1: dirty

e1: protection enabled

t1: page present

px: privilege bucket

|  |  |  |  |
| --- | --- | --- | --- |
| c1 | r1 | x1 | w1 |

c1: cacheable

r1: readable

x1: executable

w1: writeable

**The Page Table Address Register**

This register contains the base address of the page table in memory. The page table must be aligned on an 8kB boundary. The lowest three bits of the register identify the translation space.

|  |  |  |
| --- | --- | --- |
| Address63..13 | ~10 | TS3 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| TS3 | Translation Space | Address Processed | | | | | |
| 0 | 2MB |  | | | | | Address20..12 |
| 1 | 1GB |  | | | | Address29..12 | |
| 2 | 512 GB |  | | | Address38..12 | | |
| 3 | 256 TB |  | | Address47..12 | | | |
| 4 | 128 XB |  | Address56..12 | | | | |
| 5 | 2^64 | Address63..12 | | | | | |
| 6 | not used |  | | | | | |
| 7 | not used |  | | | | | |

## Load / Store Instructions

The following are the load / store instructions currently supported by the architecture.

### Loads

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Offs4 | Rt6 | Ra6 | 9x8h | Lx Rt,o4(Rn) |
| Displacement12 | | | Rt6 | Ra6 | 8x8h | Lx Rt,d12(Rn) |
| Disp6 | Rt6 | | Rb6 | Ra6 | 8x8h | Lx Rt,d6(Ra+Rb) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Register Indirect with Displacement | | Indexed | | 4-bit offset register indirect | |
| 80h | LB | 88h | LB | 90h | LB |
| 81h | LBU | 89h | LBU | 91h | LBU |
| 82h | LC | 8Ah | LC | 92h | LC |
| 83h | LCU | 8Bh | LCU | 93h | LCU |
| 84h | LH | 8Ch | LH | 94h | LH |
| 85h | LHU | 8Dh | LHU | 95h | LHU |
| 86h | LW | 8Eh | LW | 96h | LW |
| 87h | LEA | 8Fh | LEA |  |  |

### Stores

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Offs4 | Rs6 | Ra6 | Bx8h | Sx Rs,o4(Rn) |
| Displacement12 | | | Rs6 | Ra6 | Ax8h | Sx Rs,d12(Rn) |
| Disp6 | Rs6 | | Rb6 | Ra6 | Ax8h | Sx Rs,d6(Ra+Rb) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Register Indirect with Displacement | | Indexed | | 4-bit offset register indirect | |
| A0h | SB | A8h | SB | B0h | SB |
| A1h | SC | A9h | SC | B1h | SC |
| A2h | SH | AAh | SH | B2h | SH |
| A3h | SW | ABh | SW | B3h | SW |
| A4h | CACHE | ACh | CACHE |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| A5h | ~ | ADh | PUSH #i8 |
| A6h | PUSH | AEh | CAS |
| A7h | POP | AFh | ~ |

# Floating Point

Floating Point Register Array

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | fs |
|  |  | fd | |
| fq | | | |
|  |  |  | fp0 |
|  |  |  | fp1 |
|  |  |  | fp2 |
|  |  |  | fp3 |
|  |  |  | fp4 |
|  |  |  | fp5 |
|  |  |  | fp6 |
|  |  |  | fp7 |
|  |  |  | fp8 |
|  |  |  | fp9 |
|  |  |  | fp10 |
|  |  |  | fp11 |
|  |  |  | fp12 |
|  |  |  | fp13 |
|  |  |  | fp14 |
|  |  |  | fp15 |
|  |  |  | fp16 |
|  |  |  | fp17 |
|  |  |  | fp18 |
|  |  |  | fp19 |
|  |  |  | fp20 |
|  |  |  | fp21 |
|  |  |  | fp22 |
|  |  |  | fp23 |
|  |  |  | fp24 |
|  |  |  | fp25 |
|  |  |  | fp26 |
|  |  |  | fp27 |
|  |  |  | fp28 |
|  |  |  | fp29 |
|  |  |  | fp30 |
|  |  |  | fp31 |

|  |  |  |  |
| --- | --- | --- | --- |
| 15 11 | 10 8 | 7 0 |  |
| Disp5 | Flg3 | Opcode8 | Bcc |

|  |  |  |  |
| --- | --- | --- | --- |
| 23 11 | 10 8 | 7 0 |  |
| Displacement13 | Flg3 | Opcode8 | LBcc |

|  |  |  |  |
| --- | --- | --- | --- |
| 23 14 | 13 8 | 7 0 |  |
| Disp10 | Ra6 | Opcode8 | BRZ/BRNZ/BRMI/BRPL/DBNZ |

|  |  |  |  |
| --- | --- | --- | --- |
| 15 12 | 11 8 | 7 0 |  |
| Rs4 | Rd4 | Opcode8 | JALR |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 39 16 | 15 12 | 11 8 | 7 0 |  |
| Displacement24 | Rs4 | Rd4 | Opcode8 | JALR |

|  |  |
| --- | --- |
| 7 0 |  |
| Opcode8 | RET |

|  |  |  |  |
| --- | --- | --- | --- |
| 15 8 | 7 3 | 2 0 |  |
| Ra6 | Opcode5 | Flg3 | TST Ft, Ra |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 23 | 22 20 | 19 14 | 13 8 | 7 0 |  |
| ~ | Flg3 | Rb6 | Ra6 | Opcode8 | CMP Ft, Ra, Rb |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 14 | 13 8 | 7 3 | 2 0 |  |
| Imm10 | Ra6 | Opcode5 | Flg3 | CMPI Ft,#imm |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 24 | 23 16 | 15 8 | 7 0 |  |
| Imm8 | Rt8 | Ra8 | Opcode8 | SEQ Rt,Ra,#i8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 24 | 23 16 | 15 8 | 7 0 |  |
| Disp8 | Rt8 | Ra8 | Opcode8 | LB Rt,d8(Ra) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 24 | 23 16 | 15 8 | 7 0 |  |
| Rt8 | Rb8 | Ra8 | Opcode8 | LBX Rt,(Ra+Rb) |

# Instruction Set Description

A description of the instruction set follows.

## ADD - addition

ADD Rt, Ra, #imm

ADD Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 006h | Rt6 | | Rb6 | Ra6 | 028 | ADD Rt,Ra,Rb |
| Imm12 | | | Rt6 | Ra6 | 208 | ADD Rt,Ra,#imm |
| Imm4 | Rt6 | Ra6 | 268 | ADD Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra + immediate12

#### Register-Register Form

Rt = Ra + Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

This instruction may cause an overflow exception if used with the overflow trap (TRAPV) prefix.

## AND – bitwise logical ‘and’

AND Rt, Ra, #i12

AND Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 086 | Rt6 | Rb6 | Ra6 | 028 | AND Rt,Ra,Rb |
| Imm12 | | Rt6 | Ra6 | 288 | AND Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra & immediate16

#### Register-Register Form

Rt = Ra & Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

## ANDN – bitwise logical ‘and’ with complement

ANDN Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0B6 | Rt6 | Rb6 | Ra6 | 028 | AND Rt,Ra,Rb |

Operation:

#### Register-Register Form

Rt = Ra & ~Rb

Notes:

## Bcc – Branches

Bcc Fa,target\_address

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 15 11 | 10 8 | 7 0 |  |
| Disp5 | Fa3 | 4x8h | Bcc address |

Operation:

If (cond(Fa)) PC = PC + Disp5

Notes:

A branch is taken to the target address if the condition is true. Branches may branch forwards or backwards up to -16/+15 bytes in range.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | |  |  | |  |
| 40h | BEQ | branch if equal | 48h | BGT | branch if greater than |
| 41h | BNE | branch if not equal | 49h | BLE | branch if less or equal |
| 42h | BVS | branch if overflow set | 4Ah | BGE | branch if greater or equal |
| 43h | BVC | branch if overflow clear | 4Bh | BLT | branch if less than |
| 44h | BMI | branch if negative | 4Ch | BHI | branch if higher |
| 45h | BPL | branch if positive or zero | 4Dh | BLS | branch if lower or same |
| 46h | BRA | branch all the time | 4Eh | BHS | branch if higher or same |
| 47h | BNV | never branch | 4Fh | BLO | branch if lower |

## BRcc – Branches

BRcc Ra,target\_address

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 23 14 | 13 8 | 7 0 |  |
| Disp10 | Ra6 | 6x8h | BRcc address |

Operation:

If (cond(Ra)) PC = PC + Disp10

Notes:

A branch is taken to the target address if the condition is true. Branches may branch forwards or backwards up to -512/+511 bytes in range.

|  |  |  |
| --- | --- | --- |
|  | |  |
| 68h | BRZ | branch if zero |
| 69h | BRNZ | branch if not zero |
| 6Ah | BRMI | branch if MSB =1 |
| 6Bh | BRPL | branch if MSB = 0 |
| 6Ch | DBNZ | decrement, branch if not zero |

### BRK –Break

|  |
| --- |
| 7 0 |
| 00h |

#### Operation:

<none>

#### Description:

The Break exception is executed.

## BSR – Branch to Subroutine

BSR target

Instruction Formats:

|  |  |  |
| --- | --- | --- |
| 23 8 | 7 0 |  |
| Displacement16 | 6E8h | BSR address |
| Displacement24 | | 708h | BSR address |

Operation:

SP = SP - 8

memory[SP] = PC

PC = PC+sign extend(Displacement)

Notes:

The displacement may be extended up to 64 bits using a constant prefix instruction.

A new code segment may be called using the JSP instruction prefix.

### CAS – Compare And Swap

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Rt | Rc | Rb | Ra | Opcode |
| Rt6 | Rc6 | Rb6 | Ra6 | AEh8 |

#### Operation:

Rt = memory [Ra + displacement]

if memory[Ra+displacement] = Rb

memory[Ra + displacement] = Rc

#### Description:

If the contents of the addressed memory cell is equal to the contents of Rb then a sixty-four bit value is stored to memory from the source register Rc. The original contents of the memory cell are loaded into register Rt. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be word aligned. If the operation was successful then Rt and Rb will be the same value. The compare and swap operation is an atomic operation; the bus is locked during the load and potential store operation. This operation assumes that the addressed memory location is part of the volatile region of memory and bypasses the data cache.

#### Assembler:

CAS Rt,Rb,Rc,displacement[Ra]

### CLI – Clear Interrupt Mask

|  |
| --- |
| 7 0 |
| Opcode |
| FAh8 |

#### Operation:

im = 0

#### Description:

This instruction is used to enable interrupts.

### CMP Register-Register Compare

CMP Ft,Ra,Rb

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 20 | 19 14 | 13 8 | 7 0 |
| 0 | Ft3 | Rb6 | Ra6 | 188 |

#### Operation:

Ft = flags of (Ra – Rb)

#### Description:

The register compare instruction compares two registers and sets the flags in the target flag register as a result.

Compare performs both signed and unsigned comparison at the same time. A standard set of c,v,n and z flags are generated.

The most significant bit of the target register is set to the sign bit of the result. Branch instruction may branch on whether a register is minus or non-minus without performing a compare beforehand. Flags are stored in the target register as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 | 62 | 61 2 | 1 | 0 |
| n | v | 0 | z | c |

### CMP Float Double Register-Register Compare

CMP Ft,Da,Db

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 23 | 22 20 | 1918 | 17 13 | 12 8 | 7 0 |
| 1 | Ft3 | 22 | Db5 | Da5 | 188 |

#### Operation:

Ft = flags of (Da – Db)

#### Description:

The register compare instruction compares two float double registers and sets the flags in the target flag register as a result.

The most significant bit of the target register is set to the sign bit of the result. Branch instruction may branch on whether a register is minus or non-minus without performing a compare beforehand. Flags are stored in the target register as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 | 62 | 61 2 | 1 | 0 |
| < | U | 0 | = | 0 |

### CMPI Register-Immediate Compare

CMP Ft,Ra,#imm

|  |  |  |  |
| --- | --- | --- | --- |
| 23 14 | 13 8 | 7 3 | 2 0 |
| Imm10 | Ra6 | 25 | Ft3 |

#### Operation:

Ft = flags of (Ra – Rb)

#### Description:

The register immediate compare instruction compares a register to an immediate value and sets the flags in the target flag register as a result.

## COM – bitwise ones complement

COM Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 14 | 13 8 | 7 0 |  |
| 064 | Rt6 | Ra6 | 018 | COM Rt, Ra |

Operation:

#### Register-Register Form

Rt = ~Ra

Notes:

All the bits in Ra are inverted and placed into the target register Rt.

## CPUID – Central Processing Unit Identification

CPUID Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 14 | 13 8 | 7 0 |  |
| 0B4h | Rt6 | Ra6 | 018h | CPUID |

Operation:

#### Register Form

Rt = cpu info (Ra)

Notes:

The value in register Ra is used to determine which field of information on the cpu to return in register Rt.

|  |  |  |  |
| --- | --- | --- | --- |
| [Ra] | Field | Type |  |
| 0 | manufacturer lower | 8 character string |  |
| 1 | manufacturer upper | 4 character string |  |
| 2 | cpu class lower | 8 character string |  |
| 3 | cpu class upper | 4 character string |  |
| 4 | cpu\_name lower | 8 character string |  |
| 5 | cpu\_name upper | 4 character string |  |
| 6 | model number |  |  |
| 7 | serial number |  |  |
| 8 | features |  |  |
|  |  |  |  |
| 12 | location (rack,box,board, chip) |  |  |
| 13 | location – core # |  |  |
|  |  |  |  |
|  |  |  |  |

## DIV - Division

DIV Rt, Ra, #i12

DIV Rt, Ra, Rb

DIVU Rt, Ra, #i12

DIVU Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 036h | Rt6 | Rb6 | Ra6 | 028 | DIV Rt,Ra,Rb |
| Immediate12 | | Rt6 | Ra6 | 238 | DIV Rt,Ra,#imm |
| 136h | Rt6 | Rb6 | Ra6 | 028 | DIVU Rt,Ra,Rb |
| Immediate12 | | Rt6 | Ra6 | 338 | DIVU Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra / immediate12

#### Register-Register Form

Rt = Ra / Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

The signed division operation takes 69 cycles to complete. The unsigned operation takes 68 cycles to complete.

## EOR – bitwise logical exclusive ‘or’

EOR Rt, Ra, #imm

EOR Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0A6 | Rt6 | Rb6 | Ra6 | 028 | EOR Rt,Ra,Rb |
| Immediate12 | | Rt6 | Ra6 | 2A8 | EOR Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra ^ immediate16

#### Register-Register Form

Rt = Ra ^ Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions

## ENOR – complement bitwise logical exclusive ‘or’

ENOR Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0E6 | Rt6 | Rb6 | Ra6 | 028 | ENOR Rt,Ra,Rb |

Operation:

#### Register-Register Form

Rt = ~(Ra ^ Rb)

Notes:

There is no immediate form to this instruction.

## FADD – Floating Point Addition

FADD FPt, FPa, FPb

Instruction Formats:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ~4 | P2 | RM3 | FPt5 | FPb5 | FPa5 | C08 | FADD FPt,FPa,FPb |

Operation:

#### Register-Register Form

FPt = FPa + FPb

Notes:

P2 is the precision

|  |  |
| --- | --- |
| P2 |  |
| 0 | Single Precision |
| 1 | Double precision |
| 2 |  |
| 3 | Quad precision |

RM3 is the rounding mode

## FSUB – Floating Point Subtraction

FSUB FPt, FPa, FPb

Instruction Formats:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ~4 | P2 | RM3 | FPt5 | FPb5 | FPa5 | C18 | FSUB FPt,FPa,FPb |

Operation:

#### Register-Register Form

FPt = FPa - FPb

Notes:

P2 is the precision

|  |  |
| --- | --- |
| P2 |  |
| 0 | Single Precision |
| 1 | Double precision |
| 2 |  |
| 3 | Quad precision |

RM3 is the rounding mode

## GRAN – Generate Random Number

GRAN Rt

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 14 | 13 8 | 7 0 |  |
| F4h | Rt6 | ~6 | 018h | GRAN Rt |

Operation:

#### Register Form

Rt = random number

Notes:

Execution of the GRAN instruction generates a new random number according to George Marsaglia’s multiply method. The random number seed registers must be set to non-zero values before a number can be generated. The number returned by the GRAN instruction is the previous random number. The newly generated number is available in the RAND special purpose register.

### GS – Global Storage Prefix

GS: LW Rt,(Ra)

Instruction Formats:

|  |
| --- |
| 7 0 |
| 058 |

#### Operation:

#### Description:

The GS prefix causes the following load or store operation to use the global storage pointer (gs\_base) register during address formation . The gs\_base register is added onto the effective address generated by the following load / store instruction.

### IMM64,IMM56,IMM48,IMM40,IMM32,IMM24,IMM16,IMM8

### Immediate Extensions

The immediate extension prefixes are used to extend the immediate constant of the following instruction. The extensions may add from one to eight bytes more to the constant. Most, but not all instructions can accept an immediate prefix.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | Opcode8 |
| Immediate64 | | | | | | | | F8h |
| Immediate56 | | | | | | | F7h |
| Immediate48 | | | | | | F6h |
| Immediate40 | | | | | F5h |
| Immediate32 | | | | F4h |
| Immediate24 | | | F3h |
| Immediate16 | | F2h |
| Immediate8 | F1h |

### IO– Input / Output Storage Prefix

IO: LW Rt,(Ra)

Instruction Formats:

|  |
| --- |
| 7 0 |
| 068 |

#### Operation:

#### Description:

The IO prefix causes the following load or store operation to use the input / output storage pointer (ios\_base) register during address formation . The ios\_base is added onto the effective address generated by the following load / store instruction. Interrupts are disabled between this prefix and the following instruction.

## JGR – Jump Gate Routine

JGR selector

JGR [Ra]

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 28 | 27 | 26 8 | 7 0 |  |
| RPL4 | TI1 | Index19 | 778 | JGR |

|  |  |  |  |
| --- | --- | --- | --- |
| 15 | 8 | 7 0 |  |
| ~ | Ra6 | 788 | JGR |

Operation:

SP = SP[new privilege level]

Copy parameters from old stack

PUSH CS,PC

CS = Descriptor Table[selector].CS

PC = Descriptor Table[selector].PC

Notes:

This instruction is used to call code via a call gate. The selector must identify a call gate in either the local or global descriptor tables. The call gate contains the destination CS and PC. A number of parameters specified by the call gate may be copied between stacks at different privilege levels.

The JSP prefix may also be used to transfer to a routine via a call gate, however that prefix expects a following JMP or JSR instruction.

## JMP – Jump

JMP target

JMP [Ra]

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 8 | | | 7 0 |  |
| Offset16 | | | 738h | JMP address |
| Offset24 | | | | 748h | JMP address |
| ~2 | Ra6 | 758h | JMP [Ra] |
| Offset18 | | | Ra6 | 768h | JMP (address,Ra) |

Operation:

Direct address form: PC = zero extend(offset)

Register Indirect form: PC = Ra

Memory Indirect form: PC = memory[offset + Ra]

Notes:

The offset may be extended up to 64 bits using a constant prefix instruction.

A new code segment may be called using the JSP instruction prefix.

## JSP – Jump Selector Prefix

JSP selector

JSR offset

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 28 | 27 | 26 8 | 7 0 |  |
| RPL4 | TI1 | Index19 | 6F8 | JSP |

|  |  |  |  |
| --- | --- | --- | --- |
| 15 | 13 8 | 7 0 |  |
| ~2 | Ra6 | 798 | JSP |

Operation:

CS = Descriptor Table[selector].CS

Notes:

This instruction is used to transfer to a new code segment. The following jump or subroutine call will use the specified code selector in place of the current code selector. The return address stored will include the current code selector.

The selector may be either an immediate value or contained in a register.

If the JSP prefix points to a call gate, then the offset specified in the following JMP or JSR instruction is ignored. The program counter will be loaded from the offset specified in the call gate. The JGR instruction is a shorter form specifically for invoking call gates.

## JSR – Jump to Subroutine

JSR target

JSR [Ra]

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 8 | | | 7 0 |  |
| Offset16 | | | 718h | JSR address |
| Offset24 | | | | 728h | JSR address |
| ~2 | Ra6 | 608h | JSR [Ra] |
| Offset18 | | | Ra6 | 618h | JSR (address,Ra) |

Operation:

SP = SP - 8

IF (JSP prefix)

memory[SP] = code segment selector

SP = SP - 8

memory[SP] = PC

PC = zero extend(offset)

Notes:

The offset may be extended up to 64 bits using a constant prefix instruction.

A new code segment may be called using the JSP instruction prefix.

Note that the stack pointer is always decremented by sixteen. The return address stored on the stack may be in one of three different formats depending on whether or not a selector prefix is present, and whether or not the high order bits of the PC are zero. In any case the stack pointer is always decremented by a consistent amount.

Return address, long format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| w1 | ~2 | Code Segment Selector24 | | ~38 |
| w0 | 10xx4 | | Program Counter59..38 | Program Counter37..0 |

Return address, short format with selector:

|  |  |  |  |
| --- | --- | --- | --- |
| w0 | 012 | Code Segment Selector24 | Program Counter37..0 |

Return address, short format without selector:

|  |  |  |  |
| --- | --- | --- | --- |
| w0 | 00xx4 | Program Counter59..38 | Program Counter37..0 |

## LAR – Load Access Rights

LAR Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 14 | 13 8 | 7 0 |  |
| 024 | Rt6 | Ra6 | 1F8h | LAR Rt, Ra |

Operation:

#### Register-Register Form

Rt = segment ACR(selector in (Ra))

Notes:

The segment access rights field is loaded from the descriptor identified by the selector in Ra and placed into the target register Rt.

## LB – Load Byte with Sign Extend

LB Rt, d(Rn)

LB Rt, d(Ra + Rb)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Offs4 | Rt6 | Ra6 | 908h | LB Rt,d4(Ra) |
| Disp12 | | | Rt6 | Ra6 | 808h | LB Rt,d12(Ra) |
| Disp6 | Rt6 | | Rb6 | Ra6 | 888h | LB Rt,d6(Ra+Rb) |

Operation:

#### Register Indirect with Displacement Form

Rt = sign extend(memory[displacement + Ra])

#### Register-Register Form

Rt = sign extend(memory[displacement + Ra + Rb])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

## LBcc – Long Branches

LBcc Fa,target\_address

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 23 11 | 10 8 | 7 0 |  |
| Disp13 | Fa3 | 5x8h | Bcc address |

Operation:

PC = PC + Disp13

Notes:

A branch is taken to the target address if the condition is true. Branches may branch forwards or backwards up to -4096/+4095 bytes in range.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | |  |  | |  |
| 50h | LBEQ | branch if equal | 58h | LBGT | branch if greater than |
| 51h | LBNE | branch if not equal | 59h | LBLE | branch if less or equal |
| 52h | LBVS | branch if overflow set | 5Ah | LBGE | branch if greater or equal |
| 53h | LBVC | branch if overflow clear | 5Bh | LBLT | branch if less than |
| 54h | LBMI | branch if negative | 5Ch | LBHI | branch if higher |
| 55h | LBPL | branch if positive or zero | 5Dh | LBLS | branch if lower or same |
| 56h | LBRA | branch all the time | 5Eh | LBHS | branch if higher or same |
| 57h | LBNV | never branch | 5Fh | LBLO | branch if lower |

## LBU – Load Byte with Zero Extend

LBU Rt, d(Rn)

LBU Rt, d(Ra + Rb)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Offs4 | Rt6 | Ra6 | 918h | LBU Rt,d4(Rn) |
| Disp12 | | | Rt6 | Ra6 | 818h | LBU Rt,d12(Rn) |
| Disp6 | Rt6 | | Rb6 | Ra6 | 898h | LBU Rt, d6(Ra+Rb) |

Operation:

#### Register Indirect with Displacement Form

Rt = zero extend(memory[displacement + Ra])

#### Register-Register Form

Rt = zero extend(memory[displacement + Ra + Rb])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

## LC – Load Character with Sign Extend

LC Rt, d(Rn)

LC Rt, (Ra + Rb)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Offs4 | Rt6 | Ra6 | 928h | LC Rt,d4(Ra) |
| Disp12 | | | Rt6 | Ra6 | 828h | LC Rt,d12(Ra) |
| Disp6 | Rt6 | | Rb6 | Ra6 | 8A8h | LC Rt, d6(Ra+Rb) |

Operation:

#### Register Indirect with Displacement Form

Rt = sign extend(memory[displacement + Ra])

#### Register-Register Form

Rt = sign extend(memory[displacement + Ra + Rb])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The memory address must be 16 bit (two byte) aligned.

## LCU – Load Character with Zero Extend

LCU Rt, d(Rn)

LCU Rt, d(Ra + Rb)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Offs4 | Rt6 | Ra6 | 938h | LCU Rt,d4(Rn) |
| Disp12 | | | Rt6 | Ra6 | 838h | LCU Rt,d12(Rn) |
| Disp6 | Rt6 | | Rb6 | Ra6 | 8B8h | LCU Rt,d6(Ra+Rb) |

Operation:

#### Register Indirect with Displacement Form

Rt = zero extend(memory[displacement + Ra])

#### Register-Register Form

Rt = zero extend(memory[displacement + Ra + Rb])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The memory address must be 16 bit (two byte) aligned.

## LDI – Load Immediate

LDI Rt, #imm

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| Immediate10 | Rt6 | 258h | LDI Rt,#imm |
| Immediate18 | | Rt6 | 358h | LDI Rt,#imm |

Operation:

#### Register Immediate Form

Rt = immediate

Notes:

The immediate constant is sign extended to 64 bits and loaded into the target register. The constant may be extended up to 64 bits with immediate prefix instructions.

## LH – Load Half-Word with Sign Extend

LH Rt, d(Rn)

LH Rt, d(Ra + Rb)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Offs4 | Rt6 | Ra6 | 948h | LH Rt,d4(Rn) |
| Disp12 | | | Rt6 | Ra6 | 848h | LH Rt,d12(Rn) |
| Disp6 | Rt6 | | Rb6 | Ra6 | 8C8h | LH Rt, d6(Ra+Rb) |

Operation:

#### Register Indirect with Displacement Form

Rt = sign extend(memory32[displacement + Ra])

#### Register-Register Form

Rt = sign extend(memory32[displacement + Ra + Rb])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The memory address must be 32 bit (four byte) aligned.

## LHU – Load Half-Word with Zero Extend

LHU Rt, d(Rn)

LHU Rt, d(Ra + Rb)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Offs4 | Rt6 | Ra6 | 958h | LHU Rt,d4(Rn) |
| Disp12 | | | Rt6 | Ra6 | 858h | LHU Rt,d12(Rn) |
| Disp6 | Rt6 | | Rb6 | Ra6 | 8D8h | LHU Rt,d6(Ra+Rb) |

Operation:

#### Register Indirect with Displacement Form

Rt = zero extend(memory32[displacement + Ra])

#### Register-Register Form

Rt = zero extend(memory32[displacement + Ra + Rb])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The memory address must be 32 bit (four byte) aligned.

## LxDT – Load Descriptor Table Register

LIDT d(Rn)

LGDT d(Rn)

LLDT d(Rn)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Disp16 | DTt2 | Ra6 | 988h | LxDT d16(Ra) |

Operation:

#### Register Indirect with Displacement Form

DTt = memory[displacement + Ra]

Notes:

The descriptor register base and limit values are loaded from four consecutive memory words.

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The memory address must be 64 bit (word) aligned.

|  |  |  |
| --- | --- | --- |
| DTt | Register |  |
| 0 | IDT | Interrupt descriptor table |
| 1 | GDT | global descriptor table |
| 2 | LDT | local descriptor table |

## LSB – Load Segment Base

LSB Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 14 | 13 8 | 7 0 |  |
| 014 | Rt6 | Ra6 | 1F8h | LSB Rt, Ra |

Operation:

#### Register-Register Form

Rt = segment base(selector in (Ra))

Notes:

The segment base field is loaded from the descriptor identified by the selector in Ra and placed into the target register Rt.

## LSL – Load Segment Limit

LSL Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 14 | 13 8 | 7 0 |  |
| 004 | Rt6 | Ra6 | 1F8h | LSL Rt, Ra |

Operation:

#### Register-Register Form

Rt = segment limit(selector in (Ra))

Notes:

The segment limit field is loaded from the descriptor identified by the selector in Ra and placed into the target register Rt.

## LW – Load Word

LW Rt, d(Rn)

LW Rt, d(Ra + Rb)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Offs4 | Rt6 | Ra6 | 968h | LW Rt,d4(Rn) |
| Disp12 | | | Rt6 | Ra6 | 868h | LW Rt,d12(Rn) |
| Disp6 | Rt6 | | Rb6 | Ra6 | 8E8h | LW Rt,d6(Ra+Rb) |

Operation:

#### Register Indirect with Displacement Form

Rt = sign extend(memory64[displacement + Ra])

#### Register-Register Form

Rt = sign extend(memory64[offset + Ra + Rb])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The memory address must be 64 bit (eight byte) aligned.

## LWS – Load Special Purpose Register Word

LWS Sprt, d(Rn)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Displacement12 | Sprt6 | Ra6 | B58h | LWS Sprt,d12(Ra) |

Operation:

#### Register Indirect with Displacement Form

Sprt = memory[displacement + Ra]

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

## MFSPR – Move from Special Register

MFSPR Rt, Spra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 14 | 13 8 | 7 0 |  |
| D4h | Rt6 | Spr6 | 018h | MFSPR Rt,Spr |

Operation:

#### Register-Register Form

Rt = Spr

Notes:

The special purpose register is moved to the general purpose register.

|  |  |  |
| --- | --- | --- |
| Special Regno | Name | Description |
| 0 | tick | Tick register |
| 1 | vbr | vector table base address |
| 2 | bear | bus error address register |
| 4 | CR3 / PTA | paging table address register |
| 5 | CR0 | control register zero |
| 6 | CLK\_CTRL | clock rate control |
| 8 | FAULT\_PC | PC value when fault occurred |
| 9 | FAULT\_CS | CS value when fault occurred |
| 10h | SRAND1 | random seed register #1 |
| 11h | SRAND2 | random seed register #2 |
| 12h | RAND | random number register |
| 13h | prod\_high | high order product bits from a multiply |
| 2xh | SEG x | segment registers |

## MOD – Signed Modulus

MOD Rt, Ra, #i12

MOD Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 046h | Rt6 | Rb6 | Ra6 | 028 | MOD Rt,Ra,Rb |
| Immediate12 | | Rt6 | Ra6 | 248 | MOD Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra mod immediate12

#### Register-Register Form

Rt = Ra mod Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

## MODU – Unsigned Modulus

MODU Rt, Ra, #i12

MODU Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 146h | Rt6 | Rb6 | Ra6 | 028 | MODU Rt,Ra,Rb |
| Immediate12 | | Rt6 | Ra6 | 348 | MODU Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra mod immediate12

#### Register-Register Form

Rt = Ra mod Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

## MOV – Move Register to Register

MOV Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 044h | Rt6 | Ra6 | 018 | MOV Rt,Ra |

Operation:

#### Register-Register Form

Rt = Ra

Notes:

This instruction moves one register to another.

## MTSPR – Move to Special Register

MTSPR Sprt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C4h | Sprt6 | Ra6 | 018h | MTSPR Sprt,Ra |

Operation:

#### Register-Register Form

Sprt = Ra

Notes:

The general purpose register is moved to the special purpose register.

|  |  |  |
| --- | --- | --- |
| Special Regno | Name | Description |
| 0 | tick | Tick register |
| 1 | vbr | Vector table base address register |
| 2 | - | reserved |
| 5 | CR0 | control register zero |
|  |  |  |

## MUL – Signed Multiply

MUL Rt, Ra, #i12

MUL Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 026h | Rt6 | Rb6 | Ra6 | 028 | MUL Rt,Ra,Rb |
| Immediate12 | | Rt6 | Ra6 | 228 | MUL Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra \* immediate12

#### Register-Register Form

Rt = Ra \* Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

Only the low order 64 bits of the result are transferred to the target register. The upper 64 bits of the result are available from the product special purpose register.

This instruction may be used with the TRAPV prefix. In this use an overflow exception will result if the high order product bits are not equal to the sign extension of the most significant bit of the result.

## MULU – Unsigned Multiply

MULU Rt, Ra, #i12

MULU Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 126h | Rt6 | Rb6 | Ra6 | 028 | MULU Rt,Ra,Rb |
| Immediate12 | | Rt6 | Ra6 | 328 | MULU Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra \* immediate12

#### Register-Register Form

Rt = Ra \* Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

Only the low order 64 bits of the result are transferred to the target register. The upper 64 bits of the result are available from the product special purpose register.

This instruction may be used with the TRAPV prefix. In this use an overflow exception will result if the high order product bits are not equal to zero.

## NAND – Complement Bitwise Logical ‘And’

NAND Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0C6h | Rt6 | Rb6 | Ra6 | 028h | NAND Rt,Ra,Rb |

Operation:

#### Register-Register Form

Rt = ~(Ra & Rb)

Notes:

There is no immediate form to this instruction.

## NEG – Negate

NEG Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 054h | Rt6 | Ra6 | 018 | NEG Rt,Ra |

Operation:

#### Register-Register Form

Rt = -Ra

Notes:

## NOP – No Operation

NOP

Instruction Formats:

|  |  |
| --- | --- |
| EA8 | NOP |

Operation:

none

Notes:

## NOR – Complement Bitwise Logical ‘Or’

NOR Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0D6h | Rt6 | Rb6 | Ra6 | 028h | NOR Rt,Ra,Rb |

Operation:

#### Register-Register Form

Rt = ~(Ra | Rb)

Notes:

There is no immediate form to this instruction.

## NOT – Logical Not

NOT Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 078h | Rt6 | Ra6 | 018 | NOT Rt,Ra |

Operation:

#### Register-Register Form

Rt = !Ra

Notes:

If any bit in Ra is set the result in Rt is set to zero, if Ra is zero then Rt is set to one. The result in the register is thus either zero or one.

## OR – bitwise logical or

OR Rt, Ra, #i12

OR Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 096h | Rt6 | Rb6 | Ra6 | 028h | OR Rt,Ra,Rb |
| Immediate12 | | Rt6 | Ra6 | 298h | OR Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra | immediate12

#### Register-Register Form

Rt = Ra | Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

## ORN – Bitwise Logical Or with Complement

ORN Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0F6h | Rt6 | Rb6 | Ra6 | 028h | ORN Rt,Ra,Rb |

Operation:

#### Register-Register Form

Rt = Ra | ~Rb

Notes:

There is no immediate form to this instruction.

## PHP – Push Processor Status

PHP

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 02 | 076 | B6h | PHP |

Operation:

SP = SP - 8

memory[SP] = SR

Notes:

This instruction may be used to push the processor status register onto the stack.

This is an alternate mnemonic for the PUSHS SR instruction.

## PLP – Pull Processor Status

PLP

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 02 | 076 | B7h | PLP |

Operation:

SR = memory[SP]

SP = SP + 8

Notes:

This is an alternate mnemonic for POPS SR.

This instruction may be used to pop the status register from the stack. If the interrupt mask is to be cleared as a result there is a three cycle delay before interrupts are enabled.

## POP – Pop Register

POP Rt

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 02 | Rt6 | A7h | POP |

Operation:

If Rt <> 0

Rt = memory[SP]

SP = SP + 8

Notes:

This instruction may be used to pop a register from the stack.

## POP – Pop Float Double Register

POP Dt

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 43 | Dt5 | A7h | POP |

Operation:

If Rt <> 0

Dt = memory[SP]

SP = SP + 8

Notes:

This instruction may be used to pop a register from the stack.

## POPS – Pop Special Register

POPS Sprt

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 02 | Sprt6 | B7h | POP |

Operation:

If not readonly

Sprt = memory[SP]

SP = SP + 8

Notes:

This instruction may be used to pop a special purpose register from the stack. Note that read-only registers may not be modified by this instruction.

## PUSH – Push Register

PUSH Ra

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 02 | Ra6 | A6h | PUSH Ra |

Operation:

SP = SP - 8

memory[SP] = Ra

Notes:

This instruction may be used to push a register onto the stack.

## PUSH – Push Float Double Register

PUSH Da

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 43 | Da5 | A6h | PUSH Da |

Operation:

SP = SP - 8

memory[SP] = Da

Notes:

This instruction may be used to push a register onto the stack.

## PUSHC – Push Constant

PUSHC #imm

Instruction Formats:

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 0 |  |
| Constant8 | ADh | PUSHC #imm |

|  |  |  |
| --- | --- | --- |
| Constant16 | BDh | PUSHC #imm |

Operation:

SP = SP - 8

Memory[sp] = sign extend (immediate)

Notes:

This instruction pushes a constant value onto the stack. The constant value is sign extended to 64 bits before being pushed on the stack. The constant may be extended to 64 bits with a constant prefix instruction.

## PUSHS – Push Special Register

PUSH Ra

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 02 | Spra6 | B6h | PUSH Ra |

Operation:

SP = SP - 8

memory[SP] = Ra

Notes:

This instruction may be used to push a special purpose register onto the stack.

## ROL – Rotate Left

ROL Rt, Ra, #i6

ROL Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 216h | Rt6 | Rb6 | Ra6 | 028h | ROL Rt,Ra,Rb |
| 316h | Rt6 | Imm6 | Ra6 | 028h | ROL Rt,Ra,#i6 |

Operation:

#### Register Immediate Form

Rt = Ra << immediate6

#### Register-Register Form

Rt = Ra << Rb

Notes:

Most significant bits are rotated into the least significant bits.

## ROR – Rotate Right

ROR Rt, Ra, #i6

ROR Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 236h | Rt6 | Rb6 | Ra6 | 028h | ROR Rt,Ra,Rb |
| 336h | Rt6 | Imm6 | Ra6 | 028h | ROR Rt,Ra,#i6 |

Operation:

#### Register Immediate Form

Rt = Ra >> immediate6

#### Register-Register Form

Rt = Ra >> Rb

Notes:

The least significant bits are rotated to the most significant bits.

## RTD – Return and Deallocate

RTD 21

Instruction Format:

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 0 |  |
| N8 | 658 | RTD |

Operation:

PC = memory[SP]

if (type= 01)

code segment selector = memory[61:38][SP]

SP = SP + 8

if (type = 10xx)

code segment selector = memory[SP]

SP = SP + 8

SP = SP + N \* 8

Notes:

This instruction may be used to return from a subroutine and deallocate parameters from the stack frame. The program counter and possibly the code segment selector are loaded from the stack. The stack pointer is always incremented by sixteen ( two words) plus eight times the number of words to be deallocated.

Gate Routines:

Note that this is the only way to deallocate parameters copied to a stack at a higher privilege level when returning from that privilege level. A call gate routine should exit with an RTD in order to remove parameters from the stack. Parameter’s on the caller’s stack should be deallocated by the caller.

Return address, long format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| w1 | ~2 | Code Segment Selector24 | | ~38 |
| w0 | 10xx4 | | Program Counter59..38 | Program Counter37..0 |

Return address, short format with selector:

|  |  |  |  |
| --- | --- | --- | --- |
| w1 | ~64 | | |
| w0 | 012 | Code Segment Selector24 | Program Counter37..0 |

Return address, short format without selector:

|  |  |  |  |
| --- | --- | --- | --- |
| w1 | ~64 | | |
| w0 | 00xx4 | Program Counter59..38 | Program Counter37..0 |

## RTI – Return from interrupt subroutine

RTI

Instruction Format:

|  |  |
| --- | --- |
| 7 0 |  |
| 648 | RTI |

Operation:

PC = memory[SP]

SP = SP + 8

Code Segment Selector, status register = memory[SP]

SP = SP + 8

Notes:

This instruction may be used to return from an interrupt subroutine. The interrupt return address is loaded from the stack into the program counter and code segment selector. Additionally the status register is restored.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| w1 | ~2 | Code Segment Selector24 | | ~6 | Status Register32 |
| w0 | 11xx4 | | Program Counter59..38 | Program Counter37..0 | |

## RTS – Return from subroutine

RTS

Instruction Format:

|  |  |
| --- | --- |
| 7 0 |  |
| 638 | RTS |

Operation:

PC = memory[SP]

if (type= 01)

code segment selector = memory[61:38][SP]

SP = SP + 8

if (type = 10xx)

code segment selector = memory[SP]

SP = SP + 8

Notes:

This instruction may be used to return from a subroutine. The program counter and possibly the code segment selector are loaded from the stack. The stack pointer is always incremented by sixteen ( two words).

Return address, long format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| w1 | ~2 | Code Segment Selector24 | | ~38 |
| w0 | 10xx4 | | Program Counter59..38 | Program Counter37..0 |

Return address, short format with selector:

|  |  |  |  |
| --- | --- | --- | --- |
| w1 | ~64 | | |
| w0 | 012 | Code Segment Selector24 | Program Counter37..0 |

Return address, short format without selector:

|  |  |  |  |
| --- | --- | --- | --- |
| w1 | ~64 | | |
| w0 | 00xx4 | Program Counter59..38 | Program Counter37..0 |

## SB – Store Byte

SB Rs, d12(Rn)

SB Rs, d6(Ra + Rb)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| offs4 | Rs6 | Ra6 | B08h | SB Rs,d4(Rn) |
| Displacement12 | | | Rs6 | Ra6 | A08h | SB Rs,d12(Rn) |
| Disp6 | Rs6 | | Rb6 | Ra6 | A88h | SB Rs,d6 (Ra+Rb) |

Operation:

#### Register Indirect with Displacement Form

memory8[displacement + Ra] = Rs

#### Register-Register Form

memory[offset + Ra + Rb] = Rs

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

## SC – Store Character

SC Rb, d(Rn)

SC Rc, d(Ra + Rb)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Offs4 | Rs6 | Ra6 | B18h | SC Rs,o4(Rn) |
| Displacement12 | | | Rs6 | Ra6 | A18h | SC Rs,d12(Rn) |
| Disp6 | Rs6 | | Rb6 | Ra6 | A98h | SC Rs,d6 (Ra+Rb) |

Operation:

#### Register Indirect with Displacement Form

memory[displacement + Ra] = Rs

#### Register-Register Form

memory[offset + Ra + Rb] = Rs

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

## Scc – set instructions

SLT Rt, Ra, #i12

SGT Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Func6 | Rt6 | Rb6 | Ra6 | 028 | Scc Rt,Ra,Rb |
| Imm12 | | Rt6 | Ra6 | Opcode8 | Scc Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra cond immediate12

#### Register-Register Form

Rt = Ra cond Rb

Notes:

The set instructions set the target register to TRUE (1) or FALSE (0) depending on whether or not the comparison of the operands matches the test condition.

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode8 | Func6 | Mne. | Evaluation |
| 30 | 10 | SEQ | operands are equal |
| 31 | 11 | SNE | not equal |
| 38 | 18 | SGT | signed greater than |
| 39 | 19 | SLE | signed less than or equal |
| 3A | 1A | SGE | signed greater than or equal |
| 3B | 1B | SLT | signed less than |
| 3C | 1C | SHI | unsigned greater than |
| 3D | 1D | SLS | unsigned less than or equal |
| 3E | 1E | SHS | unsigned greater than or equal |
| 3F | 1F | SLO | unsigned less than |
|  |  |  |  |

### SEGx – Segment Prefix

SEG11: LW Rt,(Ra)

Instruction Formats:

|  |  |  |
| --- | --- | --- |
| 15 12 | 11 8 | 7 0 |
| ~4 | Sg4 | 078 |

#### Operation:

#### Description:

The SEGx prefix causes the following load or store operation to use the specified segment register during address formation . The segment base is added onto the effective address generated by the following load / store instruction. Interrupts are disabled between this prefix and the following instruction.

### SEI – Set Interrupt Mask

|  |
| --- |
| 7 0 |
| Opcode |
| FBh8 |

#### Operation:

im = 1

#### Description:

This instruction is used to disable interrupts.

## SH – Store Half-Word

SH Rb, d(Ra)

SH Rc, d(Ra + Rb)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Offs4 | Rs6 | Ra6 | B28h | SH Rs,o4(Rn) |
| Displacement12 | | | Rs6 | Ra6 | A28h | SH Rs,d12(Rn) |
| Disp6 | Rs6 | | Rb6 | Ra6 | AA8h | SH Rs,d6 (Ra+Rb) |

Operation:

#### Register Indirect with Displacement Form

memory[displacement + Ra] = Rb

#### Register-Register Form

memory[offset + Ra + Rb] = Rc

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

## SLL – Shift Left Logical

SLL Rt, Ra, #i6

SLL Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 206h | Rt6 | Rb6 | Ra6 | 028 | SHL Rt,Ra,Rb |
| 306h | Rt6 | Imm6 | Ra6 | 028 | SHL Rt,Ra,#i6 |

Operation:

#### Register Immediate Form

Rt = Ra << immediate6

#### Register-Register Form

Rt = Ra << Rb

Notes:

The least significant bits are loaded with zeros.

## SRA – Shift Right Arithmetic

SRA Rt, Ra, #i6

SRA Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 246 | Rt6 | Rb6 | Ra6 | 28 | SRA Rt, Ra, Rb |
| 346 | Rt6 | Imm6 | Ra6 | 28 | SRA Rt, Ra, #i6 |

Operation:

#### Register Immediate Form

Rt = Ra >> immediate6

#### Register-Register Form

Rt = Ra >> Rb

Notes:

Performs an arithmetic shift right, preserving the sign bit of the value.

## SRL – Shift Right Logical

SRL Rt, Ra, #i6

SRL Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 22h | Rt6 | Rb6 | Ra6 | 028h | SHR Rt,Ra,Rb |
| 32h | Rt6 | Imm6 | Ra6 | 028h | SHR Rt,Ra,#i6 |

Operation:

#### Register Immediate Form

Rt = Ra >> immediate6

#### Register-Register Form

Rt = Ra >> Rb

Notes:

The most significant bits are loaded with zeros.

## SUB - subtraction

SUB Rt, Ra, #imm

SUB Rt, Ra, Rb

Instruction Formats:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 016h | Rt6 | | Rb6 | Ra6 | 028 | SUB Rt,Ra,Rb |
| Imm12 | | | Rt6 | Ra6 | 218 | SUB Rt,Ra,#imm |
| Imm4 | Rt6 | Ra6 | 278 | SUB Rt,Ra,#imm |

Operation:

#### Register Immediate Form

Rt = Ra + immediate12

#### Register-Register Form

Rt = Ra + Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

This instruction may cause an overflow exception if used with the overflow trap (TRAPV) prefix.

## SW – Store Word

SW Rs, d(Rn)

SW Rs, d(Ra + Rb)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Offs4 | Rs6 | Ra6 | B38h | SW Rs,o4(Rn) |
| Displacement12 | | | Rs6 | Ra6 | A38h | SW Rs,d12(Rn) |
| Disp6 | Rs6 | | Rb6 | Ra6 | AC8h | SW Rs,d6 (Ra+Rb) |

Operation:

#### Register Indirect with Displacement Form

memory[displacement + Ra] = Rs

#### Register-Register Form

memory[displacement + Ra + Rb] = Rs

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

## SWS – Store Special Purpose Register Word

SWS Sprs, d(Rn)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Displacement12 | Sprs6 | Ra6 | B48h | SWS Sprs,d12(Rn) |

Operation:

#### Register Indirect with Displacement Form

memory[displacement + Ra] = Sprs

#### Register-Register Form

memory[displacement + Ra + Rb] = Sprs

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

### SWE –Software Exception

SWE 410

Instruction Formats:

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 1 | 0 |
| N7..0 | 0110011b | N8 |

#### Operation:

<none>

#### Description:

The interrupt routine is executed.

### SWE3 –Software Exception #3

SWE 3

Instruction Formats:

|  |
| --- |
| 7 0 |
| 6Dh |

#### Operation:

<none>

#### Description:

The interrupt routine #3 is executed.

## SXB – Sign Extend Byte

SXB Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 14 | 13 8 | 7 0 |  |
| 084h | Rt6 | Ra6 | 018h | SXB |

Operation:

#### Register Form

Rt = sign extend (Ra)

Notes:

The most significant bits (8 to 63) are loaded with the sign extension of bit 7.

## SXC – Sign Extend Character

SXC Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 14 | 13 8 | 7 0 |  |
| 094h | Rt6 | Ra6 | 018h | SXC |

Operation:

#### Register Form

Rt = sign extend (Ra)

Notes:

The most significant bits (16 to 63) are loaded with the sign extension of bit 15.

## SxDT – Store Descriptor Table Register

SIDT d(Rn)

SGDT d(Rn)

SLDT d(Rn)

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Disp16 | DTs2 | Ra6 | B88h | SxDT d16(Ra) |

Operation:

#### Register Indirect with Displacement Form

memory[displacement + Ra] = DTs

Notes:

The descriptor register base and limit values are stored to four consecutive memory words.

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The memory address must be 64 bit (word) aligned.

|  |  |  |
| --- | --- | --- |
| DTs | Register |  |
| 0 | IDT | Interrupt descriptor table |
| 1 | GDT | global descriptor table |
| 2 | LDT | local descriptor table |

## SXH – Sign Extend Half-Word

SXH Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 14 | 13 8 | 7 0 |  |
| 0A4h | Rt6 | Ra6 | 018h | SXH |

Operation:

#### Register Form

Rt = sign extend (Ra)

Notes:

The most significant bits (32 to 63) are loaded with the sign extension of bit 31.

### TLS – Thread Local Storage Prefix

TLS: LW Rt,(Ra)

Instruction Formats:

|  |
| --- |
| 7 0 |
| 048 |

#### Operation:

#### Description:

The TLS prefix causes the following load or store operation to use the thread local storage pointer (tls\_base) register during address formation . The tls\_base is added onto the effective address generated by the following load / store instruction. Interrupts are disabled between this prefix and the following instruction.

### TRAPV – Trap on overflow prefix

TRAPV: ADDI Rt,Ra,#1000

Instruction Formats:

|  |
| --- |
| 7 0 |
| 628 |

#### Operation:

#### Description:

The trapv prefix causes the following arithmetic operation to execute the overflow trap when an overflow condition is present.

Interrupts are disabled between this prefix and the following instruction.

### TST - Register Test Compare

TST Ft,Ra

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 15 14 | 13 8 | 7 3 | 2 0 |
| 02 | Ra6 | 15 | Ft3 |

#### Operation:

Ft = flags of (Ra – 0)

#### Description:

The register test compare compares a register against the value zero and sets the flags appropriately.

### TST – Float Double Register Test Compare

TST Ft,Da

Instruction Formats:

|  |  |  |  |
| --- | --- | --- | --- |
| 15 13 | 12 8 | 7 3 | 2 0 |
| 43 | Da5 | 15 | Ft3 |

#### Operation:

Ft = flags of (Da – 0.0)

#### Description:

The register test compare compares a float double register against the value zero and sets the flags appropriately.

## VERR – Verify Readable

VERR Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 14 | 13 8 | 7 0 |  |
| 044 | Rt6 | Ra6 | 1F8h | VERR Rt, Ra |

Operation:

#### Register-Register Form

IF IsReadable(segment[selector in Ra])

Rt = 1

else

Rt = 0

Notes:

The segment identified by the selector in Ra is tested for readability. A TRUE/ FALSE result is placed into the target register Rt.

## VERW – Verify Writable

VERW Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 14 | 13 8 | 7 0 |  |
| 054 | Rt6 | Ra6 | 1F8h | VERW Rt, Ra |

Operation:

#### Register-Register Form

IF IsWritable(segment[selector in Ra])

Rt = 1

else

Rt = 0

Notes:

The segment identified by the selector in Ra is tested for writability. A TRUE/ FALSE result is placed into the target register Rt.

## VERX – Verify Executable

VERX Rt, Ra

Instruction Formats:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 14 | 13 8 | 7 0 |  |
| 064 | Rt6 | Ra6 | 1F8h | VERX Rt, Ra |

Operation:

#### Register-Register Form

IF IsExecutable(segment[selector in Ra])

Rt = 1

else

Rt = 0

Notes:

The segment identified by the selector in Ra is tested for executability. A TRUE/ FALSE result is placed into the target register Rt.

Opcode

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | BRK | {r} | {rr} |  | TLS | GS | IO | SEG x | TST Ft,Ra / FDTST | | | | | | | |
| 1x | CMPI Ft,Ra,#i10 | | | | | | | | CMP |  |  |  |  |  |  | {SEGT} |
| 2x | ADD # | SUB # | MUL # | DIV # | MOD # | LDI #i10 | ADD #i4 | SUB #i4 | AND # | OR # | EOR # |  |  |  |  |  |
| 3x | SEQ# | SNE# | MULU # | DIVU # | MODU # | LDI #i18 |  |  | SGT# | SLE# | SGE# | SLT# | SHI# | SLS# | SHS# | SLO# |
| 4x | BEQ | BNE | BVS | BVC | BMI | BPL | BRA | BRN | BGT | BLE | BGE | BLT | BHI | BLS | BHS | BLO |
| 5x | LBEQ | LBNE | LBVS | LBVC | LBMI | LBPL | LBRA | LBRN | LBGT | LBLE | LBGE | LBLT | LBHI | LBLS | LBHS | LBLO |
| 6x | JSR [Rn] | JSR (,Rn) | TRAPV | RTS | RTI | RTD | SWI | | BRZ | BRNZ | BRMI | BRPL | DBNZ | SWI3 | BSR16 | JSP |
| 7x | BSR24 | JSR16 | JSR24 | JMP16 | JMP24 | JMP [Rn] | JMP (,Rn) | JGR |  |  |  |  |  |  |  |  |
| 8x | LB | LBU | LC | LCU | LH | LHU | LW | LEA | LBx | LBUx | LCx | LCUx | LHx | LHUx | LWx | LEAx |
| 9x | LB4 | LBU4 | LC4 | LCU4 | LH4 | LHU4 | LW4 |  | LxDT | LFS | LFSx | LFD | LFDx |  | LFQ | LFQx |
| Ax | SB | SC | SH | SW | CACHE |  | PUSH | POP | SBx | SCx | SHx | SWx | CACHEx | PUSHC8 | CAS |  |
| Bx | SB4 | SC4 | SH4 | SW4 | SWS | LWS | PUSHS | POPS | SxDT | SFS | SFSx | SFD | SFDx | PUSHC16 | SFQ | SFQx |
| Cx | FADD | FSUB | FMUL | FDIV |  |  |  |  |  |  |  |  |  |  |  |  |
| Dx |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Ex |  |  |  |  |  |  |  |  |  |  | NOP |  |  |  |  |  |
| Fx |  | IMM1 | IMM2 | IMM3 | IMM4 | IMM5 | IMM6 | IMM7 | IMM8 |  | CLI | SEI |  |  |  |  |

01 Group

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|  |  |  |  | SWAP | MOV | NEG | COM | NOT | SXB | SXC | SXH | CPUID | MTSPR | MFSPR | MOVS | GRAN |

02 Group

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | ADD | SUB | MUL | DIV | MOD |  | ADC | SBC | AND | OR | EOR | ANDN | NAND | NOR | ENOR | ORN |
| 1x | SEQ | SNE | MULU | DIVU | MODU |  |  |  | SGT | SLE | SGE | SLT | SHI | SLS | SHS | SLO |
| 2x | SLL | ROL | SRL | ROR | SRA |  |  |  |  |  |  |  |  |  |  |  |
| 3x | SLL # | ROL # | SRL # | ROR # | SRA # |  |  |  |  |  |  |  |  |  |  |  |

1F Group

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|  | LSL | LSB | LAR |  | VERR | VERW | VERX |  |  |  |  |  |  |  |  |  |