FT64v8c

## Rational

Why yet another architecture? Depending on a single architecture increases the odds that if something goes amiss with it, everyone is affected. Nature does not have a single architecture.

## The Beauty of ISA’s

Some of the best ISA’s aren’t very attractive, they have all kinds of features that the novice user asks: why did they do that? The ISA’s are not very attractive because they are functional and serve a purpose for which they were engineered.

## Separation of Register Sets

One of the things the 68k architecture does right is separate data and address calculations. Address calculations don’t require the same number and variability of operations as data calculations. Many modern processors have explicit address generation units separate from the data calculation ALU. FT64 takes this one step further and separates code and data addresses. Many modern processors contain branch calculation units, or flow control units. It appears to the author that the register set depends on the functional unit although it doesn’t have to. Floating point units often come with their own register set.

The PowerPC has several features that the author likes. One of the things the PowerPC architecture gets right is the provision for multiple condition code registers. A second feature the of PowerPC is indexed addressing.

Separating out sets of registers specific to functional units may make it possible to get away with fewer register ports, in particular register write ports, which tend to be expensive.

## Addressing Modes

The problem with indexed addressing is that when it’s needed it’s needed. Sure, it can be emulated using several simpler instructions and additional registers, but why not just include it directly in the ISA? There are code density benefits to supporting indexed addressing, if not clock cycle benefits. The base addressing mode supported by FT64 is indexed with displacement addressing mode. Other modes can be emulated as a subset of that mode.

## Compressed Instruction Sets

Compressed instruction sets are in the author’s view a kludge to increase code density. What’s really needed are variable instruction lengths. Part of the appeal of a compressed instruction set is that it makes the calculation of the instruction length simple. When active, compressed instructions are usually a fixed length. FT64v8 will have variable instruction lengths and the length of the instruction will be determined by the first byte of the instruction.

# Programming Model

# Registers

One thing that seems to be clear in modern computers is that memory is much slower than the cpu. This means that caching of values becomes extremely important. Machine registers are the level zero cache for values from memory. Having adequate types and numbers of registers is important.

## Data Registers (D0 to D31)

|  |  |  |
| --- | --- | --- |
| Xn6 |  |  |
| 0 | always zero |  |
| 1 | return value |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |
| 8 |  |  |
| 9 |  |  |
| 10 |  |  |
| 11 |  |  |
| 12 |  |  |
| 13 |  |  |
| 14 |  |  |
| 15 |  |  |
| 16 |  |  |
| 17 |  |  |
| 18 |  |  |
| 19 |  |  |
| 20 |  |  |
| 21 |  |  |
| 22 |  |  |
| 23 |  |  |
| 24 |  |  |
| 25 |  |  |
| 26 |  |  |
| 27 |  |  |
| 28 |  |  |
| 29 |  |  |
| 30 |  |  |
| 31 |  |  |

## Address Registers (A0 to A15)

Data address registers are used to locate data as opposed to instructions. The stack pointer is banked with a separate register for each operating level.

|  |  |  |  |
| --- | --- | --- | --- |
| Xn6 | Reg | 63 0 |  |
| 32 | A0 | always zero |  |
| 33 | A1 | return value |  |
| 34 | A2 |  |  |
| 35 | A3 |  |  |
| 36 | A4 |  |  |
| 37 | A5 |  |  |
| 38 | A6 |  |  |
| 39 | A7 |  |  |
| 40 | A8 |  |  |
| 41 | A9 |  |  |
| 42 | A10 |  |  |
| 43 | A11 |  |  |
| 44 | A12 |  |  |
| 45 | A13 |  |  |
| 46 | A14 | Frame Pointer |  |
| 47 | A15 | Stack Pointer |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 48 | RA | leaf return address |  |
| 49 | RA |  |  |
| 50 |  |  |  |
| 51 |  |  |  |
| 52 |  |  |  |
| 53 | CL | catch link |  |
| 54 | XP | exceptioned program counter |  |
| 55 | PC | Program Counter |  |
|  |  |  |  |
| 56 | A15 | Supervisor stack pointer |  |
| 57 | A15 | Hypervisor stack pointer |  |
| 58 | A15 | Machine stack pointer |  |

## Code Address Registers (CA0 to CA7)

The processor contains eight code address registers (CA0-CA7). Several of the registers are reserved for predefined purposes. A code address register is used in the formation and storage of code addresses. Code address registers may hold branch targets.

|  |  |  |  |
| --- | --- | --- | --- |
| Xn6 | Reg |  | Usage |
| 48 | RA |  | Subroutine return address |
| 49 | RA1 |  |  |
| 50 | LC |  |  |
| 51 | LC |  |  |
| 52 |  |  |  |
| 53 | CL | Catch Link Register | Used by the compiler to link to try/catch handlers. |
| 54 | XP | Exceptioned PC | This register is automatically set during a hardware interrupt or exception |
| 55 | PC | Program Counter | Relative address formation. |

Code address registers may be used to point to a block of code from which the JSR instruction can index into with its 24-bit offset. For instance, a register may contain a pointer to a class method jump list; the JSR instruction can then index into this list in order to invoke a method.

The presence of multiple code address registers allows multi-level return addresses to be used for performance. Leaf routines may use CA0 as the return address. Next to leaf routines may use CA1, etc. So that memory operations are avoided when implementing subroutine call and return.

The program counter register is read-only. The program counter cannot be modified by moving a value to this register.

## Program Counter (CA7)

|  |  |
| --- | --- |
| 63 40 | 39 0 |
| Program Bank | Program Counter |

The program counter is special in that it is always incrementing by the size of the instructions fetched as a program runs. Program code is byte aligned. To improve performance only the low order 40 bits of the program counter increment. The entire program counter may be loaded with a jump instruction. If the upper six bits of the program counter/ bank are all ones, then segmentation with the code segment is ignored.

## Condition Code (CC0 to CC7)

There are eight condition code registers. Each condition code register contains eight flags.

|  |  |
| --- | --- |
| Reg |  |
| CC0 |  |
| CC1 |  |
| CC2 |  |
| CC3 |  |
| CC4 |  |
| CC5 |  |
| CC6 |  |
| CC7 |  |

Each condition code register has the following organization:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| U | ~ | P | O | C | V | N | Z |

|  |  |  |
| --- | --- | --- |
| Bit | Flag | Meaning |
| 0 | Z | zero |
| 1 | N | negative |
| 2 | V | overflow |
| 3 | C | carry |
| 4 | O | odd |
| 5 | P | parity |
| 6 |  | reserved |
| 7 | U | unordered |

## Segment Registers (ZS, DS, ES, FS, GS, HS, SS, CS)

Segment selector registers are 24-selectors into either the global or local segment descriptor tables. Loading a selector register triggers a load of the segment descriptor information into a hidden descriptor cache.

|  |  |  |
| --- | --- | --- |
| Sn4 | Mne. | Description |
| 0 | ZS |  |
| 1 | DS | data selector |
| 2 | ES |  |
| 3 | FS |  |
| 4 | GS |  |
| 5 | HS |  |
| 6 | SS | stack selector |
| 7 | CS | code selector |
| 8 | XS | exception selector |
| 9 | RS | return code selector |
| 10 | TR | thread selector (thread register) |
| 11 | LDT | local descriptor table selector |

|  |  |  |
| --- | --- | --- |
|  | GDT | global descriptor table pointer |
|  | IDT | interrupt descriptor table pointer |

## Control and Status Registers

### CORENUM (0x001)

This register contains a number that is externally supplied on the corenum\_i input bus to represent the hardware thread id or the core number. No core should have the value zero as the corenum.

### TICK (0x002)

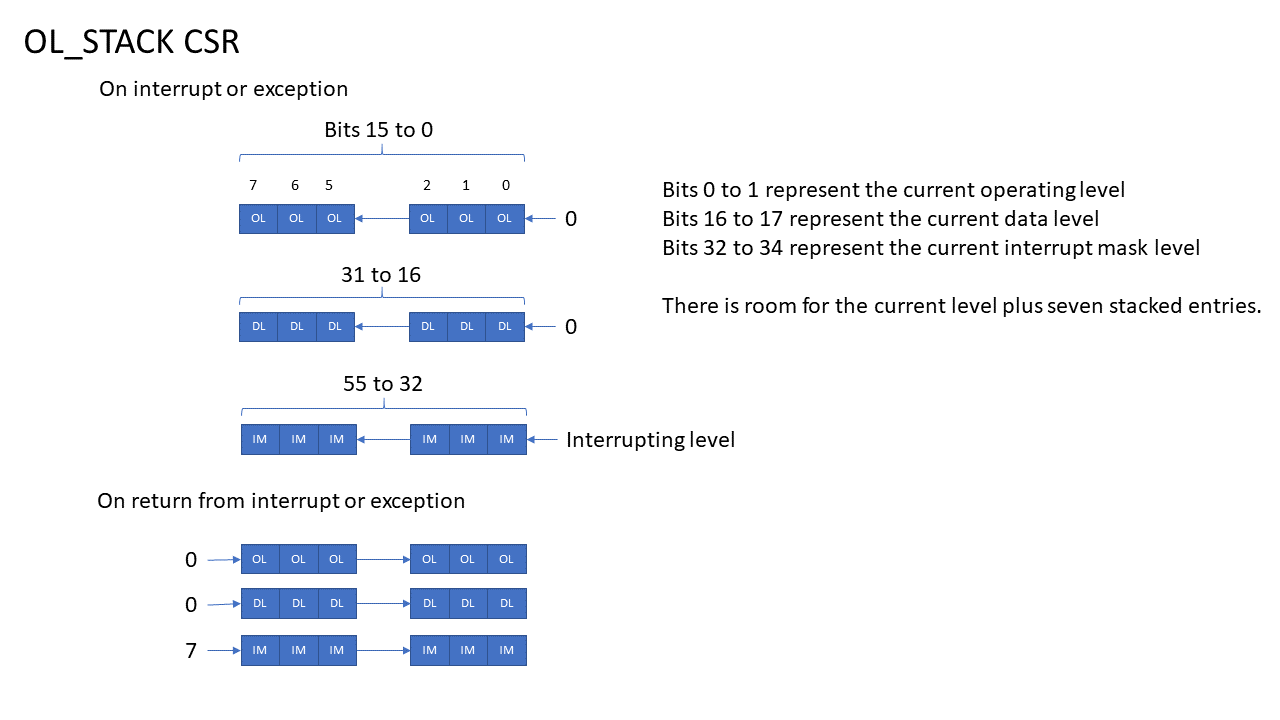
This register contains a tick count of the number of clock cycles that have passed since the last reset. Note that this register should not be used for precise timing as the processor’s clock frequency may vary for performance and power reasons. The TIME CSR may be used for wall-clock timing as it has its own timing source.

### BADADDR (CSR 0x007)

This register contains the effective address for a load / store operation that caused a memory management exception or a bus error. Note that the address of the instruction causing the exception is available in the XP register.

### OL\_STACK (0x041)

This register contains the operating, data level stack and interrupt mask stack. When an exception or interrupt occurs, this register is shifted to the left and zeros are inserted in the low order bits, when an RTI instruction is executed this register is shifted to the right restoring the current operating level. On RTI the last stack entry is set to zero which will select the machine operating level on stack underflow. The low order 32 bits of the register implement the operating and data level stack. The low order 16 bits are the code/stack operating level, the next 16 bits are the data operating level. The upper 32 bits implement the interrupt level stack.



### PL\_STACK (0x042)

This register contains the privilege level stack. When an exception or interrupt occurs, this register is shifted to the left and zeros loaded into the low order bits, when an RTI instruction is executed this register is shifted to the right. On RTI the last stack entry will be set to zero which will select privilege level zero on stack underflow.

### XPC (0x048 to 0x4F)

This sets of registers contains the interrupt or exception stack of the program counter register. The top of the stack is register 0x48. When an interrupt or exception occurs register 0x48 to 0x4E are copied to the next register and the program counter is placed into register 0x48. When an RTI instruction is executed the program counter is loaded from register 0x048 and registers 0x048 to 0x047 are loaded with the next register. Register 0x04F is loaded with the address of the break handler so that in the event of an underflow the break handler will be executed.

### XS (0x050 to 0x052)

This set of registers contains the interrupt or exception stack of code segment selectors. The top of stack is register 0x050 bits 0 to 23. This set of registers is treated as a single unit. When an interrupt or exception occurs the registers are shifted left by 24 bits. When an RTI instruction is executed this set of registers is shifted right 24 bits.

# Segmentation

## Overview

Segmentation is a low overhead means of memory protection and virtualization. Providing separate protected address spaces for different applications is the job of the operating system. Ideally segmentation hardware should not be visible to the application. The application should appear as though it has a flat memory model. The core contains eight segment registers. The segmentation system is managed via a combination of hardware and software. Up to 256 privilege levels are available.

## Privilege levels

Memory access is available according to privilege levels. The segmentation system allows up to 256 privilege levels.

## Usage

The segment register to use during address formation is identified by a three-bit field in the address. For code addresses segment register #7 (the CS) is always used.

* If segmentation is not desired then segmentation can effectively be ignored by setting all the segment registers to zero. The processor can also be built without segmentation by commenting out the ‘SEGMENTATION’ definition.

## Software Support

Segmentation is software supported. A software implementation allows a high degree of flexibility when implementing the segmentation model. Loading a value into a selector register causes a software segmentation exception to occur. The exception routine then loads the segment base, limit and access rights from a table in memory. It’s up to the system level software to determine if protection rules are violated.

Segment registers may only be transferred to or from one of the general-purpose registers. The [CSRRW](#_MTSPR_–Register-Special_Register) instruction can be used to perform the move. After loading a segment register the instruction stream should be synchronized with a memory barrier ([MEMSB](#_MEMSB_–_Memory)) to ensure the segment value can be ready for a following memory operation.

There are two cause codes in the cause table reserved for implementing far subroutine call and return instructions.

## Address Formation:

Address bits 0 to 5 pass through unchanged. Address bits 63 to 6 are added to the contents of the segment register to form the final segmented address. Note that there is a shift of six bits for the segment base address.

|  |  |  |
| --- | --- | --- |
| 0 | Address[63:6] | Address[5:0] |
| + | + | + |
| Segment register value[95:0] | | 0 |
| = | | |
| Segmented address[101:0] | | |

## Selecting a segment register

A specific segment register for a memory operation may be selected using a segment prefix in assembler code. Segment prefixes apply to data addresses only. Code addresses always use segment register #7 – the code segment. The segment prefix indicator is encoded by a three-bit field in the instruction.

## Selectors

The core uses selectors as a more compact way to represent segment registers. Rather than pass the entire segment descriptor to routines (256 bits) and have each routine check for privilege violations, the core uses 24-bit selectors. Privilege violations are checked for at the time the segment register components (base, limit and access rights) are loaded. The selector includes a field identifying the privilege level, and a second field identifying which segment descriptor the selector is associated with. The selector format is shown below.

### Selector Format

|  |  |  |
| --- | --- | --- |
| 23 16 | 15 | 14 0 |
| PL8 | T | Index15 |

PL8: the privilege level associated with the segment

Index23: the index into the descriptor table

T: 0 = global, 1 = local descriptor table

## Non-Segmented Code Area

The address range defined as 64’hFFxxxxxxxxxxxxxx (the top byte is ‘FF’) is a non-segmented code area. This area allows the operating system to work without paying attention to the code segment. Interrupt and exception vectors should vector into the non-segmented code area. The only way to change the code segment is by transferring to the operating system via a sys call instruction.

## Changing the Code Segment

The only way to change the code segment is by transferring to the operating system via a sys call instruction. The operating system, while operating in the non-segmented code area, can alter the code segment without causing a transfer of control. The operating system establishes the code segment for a thread while running in the non-segmented code area. To support far subroutine calls and returns there are vectors in the vector table that allow implementation of a far call or return.

## The Descriptor Table

The descriptor table is a table that contains information on the location and size for segments in the form of memory descriptors. Each descriptor is 32 bytes in size. Memory descriptor entries in the table have the following format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 255 244 | 243 192 | | 191 128 | 127 64 | 63 0 |
| w0 | ACR16 | ~32 | Base16 | Upper bound64 | Lower bound64 | Base64 |
| w1 | ACR16 | ~32 | Base16 | Upper bound64 | Lower bound64 | Base64 |
| … |  |  |  |  |  |  |

The descriptor table may contain other types of descriptors beyond basic memory descriptors, such as call gates.

The base address of, and the number of entries in the descriptor table is contained in the LDT or GDT special purpose registers. The descriptor table may be updated with regular load and store instructions when the processor is at privilege level zero.

24-bit selectors are used to index into the table in order to determine the characteristics of the segment.

#### Memory Descriptors

Memory descriptors describe the location and size of memory segments. They have the following format:

|  |  |  |  |
| --- | --- | --- | --- |
| n+3 | ACR16 | ~32 | Base79..64 |
| n+2 | Upper Bound63..0 | | |
| n+1 | Lower Bound63..0 | | |
| n | Base63..0 | | |

#### The Access Rights Field (ACR16) – Memory Descriptor

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 |  |  | 12 | 11 | 10 | 9 | 8 | 7 |  |  | 0 |
| P | ~ | ~ | 1/S | Ex | C/Stk | W/R | A | DPL8 | | | |

P: 1 = segment present, 0 = segment not present

S: 0 = system descriptor, 1 = memory descriptor

EX: 1 = executable, 0 = data

Code Segment Data Segment

C: 1= conforming Stk: 1=stack segment

R: 1 = readable W: 1=writeable

A: 1= accessed

DPL8 = descriptor privilege level

#### Typical Values for ACR

9A00 – executable, readable code segment, privilege level zero

9200 – read/writeable data segment, privilege level zero

9600 – read / writeable stack segment, privilege level zero

### System Segment Descriptors

System descriptors are identified by having bit 12 of the access rights character set to zero. There are potentially sixteen different system descriptor types.

#### The Access Rights Field (ACR16) – System Descriptor

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 |  |  | 12 | 11 | 10 | 9 | 8 | 7 |  |  | 0 |
| P | ~ | ~ | 0 | Type4 | | | | DPL8 | | | |

|  |  |  |
| --- | --- | --- |
| Type4 | Gate |  |
| 0 | unused |  |
| 2 | LDT descriptor |  |
| 4 | Call gate |  |
| 5 | Thread Gate |  |
| 6 | Interrupt Gate |  |
| 7 | Trap gate |  |

#### LDT Descriptor

The LDT descriptor establishes the location and size of the local descriptor table in memory.

|  |  |  |  |
| --- | --- | --- | --- |
| n+3 | ACR16 | ~48 | |
| n+2 | ~64 | | |
| n+1 | ~57 | | Pages6..0 |
| n | Base76..13 | | |

#### Call Gate Descriptor

|  |  |  |  |
| --- | --- | --- | --- |
| ACR16 | ~48 | | |
| ~64 | | | |
| ~35 | | N5 | Selector23..0 |
| Base63..0 | | | |

## Segment Load Exception

Moving a value to a selector register triggers a segment load exception in order to allow the segment descriptor to be loaded from one of the descriptor tables. This exception is triggered for a CSRRW instruction. There is a separate exception vector (vectors #232 to 239) to handle each segment register. The selector value being loaded into the segment register is reflected in the ARG1 special purpose register.

## Segment Bounds Exception

If an address is greater than the bound specified in the segment upper bound register or less than the address specified in the segment lower bound register then a segment bounds exception occurs. This applies for all segments including code and data segments.

## Segment Usage Conventions

Segment register #7 is the code segment (CS) register. All program counter addresses are formed with the code segment register unless the upper byte of the address is ‘FF’ in which case the code segment is ignored.

Segment register #6 is the stack segment (SS) register by convention. Future versions of the core may use this register implicitly for stack accesses. The assembler automatically selects the stack segment when one of the stack pointer registers is specified in the instruction. Segment register #1 is the data segment (DS) by convention. The data segment is selected as the segment register for memory operations when the stack segment is not selected.

## Gateway Table

The gateway table contains call, interrupt, and thread gate descriptors. The descriptors in this table are a different format than the memory descriptor table. The first 512 table entries are reserved for interrupts and exceptions.

#### Call Gate Descriptor

|  |  |  |  |
| --- | --- | --- | --- |
| ACR16 | ~19 | N5 | Selector23..0 |
| Offset63..0 | | | |

## Power-up State

On reset the value in the segment registers are undefined. Note that the processor begins executing instructions out of the non-segmented code area as the reset address is 64’hFFFFFFFFFFFC0100. One of the first tasks of the boot program would be to initialize the segment registers to known values. The segment register must be setup to perform data accesses properly.

### Segment Registers

|  |  |  |  |
| --- | --- | --- | --- |
| Num |  | Long name | Comment |
| 0 | ZS | zero (NULL) segment | by convention contains zero |
| 1 | DS | data segment | by convention – default for loads/stores |
| 2 | ES | extra segment | by convention |
| 3 | FS |  |  |
| 4 | GS |  |  |
| 5 | HS |  |  |
| 6 | SS | Stack segment | default for stack load/stores |
| 7 | CS | Code segment | always used for code addressing |

# Instruction Set Description

## ADD - Add

**Description**:

Add two values and store the result in a data register. The first operand must be in a data register. The second operand may be in a data register or an immediate value supplied in the instruction.

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 0 | Db5 | Da5 | Dt5 | 24h8 |

**Instruction Format**:

|  |  |  |  |
| --- | --- | --- | --- |
| 31 18 | 17 13 | 12 8 | 7 0 |
| Immediate14 | Da5 | Dt5 | 04h8 |

**Instruction Format**:

|  |  |  |  |
| --- | --- | --- | --- |
| 47 18 | 17 13 | 12 8 | 7 0 |
| Immediate30 | Da5 | Dt5 | 14h8 |

## ADDA – Add to Address

**Description**:

Add to an address register.

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 1 | Db5 | Aa5 | At5 | 24h8 |

**Instruction Format**:

|  |  |  |
| --- | --- | --- |
| 16 13 | 12 8 | 7 0 |
| Immed3 | At5 |  |

|  |  |  |
| --- | --- | --- |
| 23 13 | 12 8 | 7 0 |
| Immediate11 | At5 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 47 18 | 17 13 | 12 8 | 7 0 |
| Immediate30 | Aa5 | At5 | 44h8 |

## AND – Bitwise And

**Description**:

And two values and store the result in a data register. The first operand must be in a data register. The second operand may be in a data register or an immediate value supplied in the instruction.

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 0 | Db5 | Da5 | Dt5 | 28h8 |

**Instruction Format**:

|  |  |  |  |
| --- | --- | --- | --- |
| 31 18 | 17 13 | 12 8 | 7 0 |
| Immediate14 | Da5 | Dt5 | 08h8 |

**Instruction Format**:

|  |  |  |  |
| --- | --- | --- | --- |
| 47 18 | 17 13 | 12 8 | 7 0 |
| Immediate30 | Da5 | Dt5 | 18h8 |

## ANDA – Bitwise And to Address

**Description**:

And two values and store the result in a data register. The first operand must be in a data register. The second operand may be in a data register or an immediate value supplied in the instruction.

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 1 | Db5 | Aa5 | At5 | 28h8 |

**Instruction Format**:

|  |  |  |  |
| --- | --- | --- | --- |
| 31 18 | 17 13 | 12 8 | 7 0 |
| Immediate14 | Da5 | Dt5 | h8 |

**Instruction Format**:

|  |  |  |  |
| --- | --- | --- | --- |
| 47 18 | 17 13 | 12 8 | 7 0 |
| Immediate30 | Aa5 | At5 | 48h8 |

## BEQ – Branch if Equal

**Description**:

Branch if the condition code register indicates equality.

**Target Address Calculation**:

The lower 12 bits of the target address come from the offset field in the instruction. The middle 20 bits of the target address come from the sum of sign extended displacement field of the instruction shifted left by 12 bits and the code address register bits 12 to 31. The upper 32 bits of the target address come from the upper 32 bits of the code address register.

The program counter register may be used in the calculation.

**Instruction Format (DPO)**:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 29 | 28 24 | 23 12 | 11 | 10 8 | 7 0 |
| CA3 | Disp5 | Offset12 | ~ | CC3 | 36h8 |

## BIT – Bit Test

**Description**:

Test if a bit is set and store result in a condition code register. The zero flag of the condition register is updated with the value of the bit (zf = 1 means bit is clear). The negative, parity, and odd flags are set according to the value in the register.

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 18 | 17 13 | 1211 | 10 8 | 7 0 |
| Bn6 | Da5 | 02 | CC3 | 07h8 |

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 18 | 17 13 | 1211 | 10 8 | 7 0 |
| Bn6 | Aa5 | 12 | CC3 | 07h8 |

## BNE – Branch if Not Equal

**Description**:

Branch if the condition code register indicates inequality.

**Target Address Calculation**:

The lower 12 bits of the target address come from the offset field in the instruction. The middle 20 bits of the target address come from the sum of sign extended displacement field of the instruction shifted left by 12 bits and the code address register bits 12 to 31. The upper 32 bits of the target address come from the upper 32 bits of the code address register.

**Instruction Format (DPO)**:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 29 | 28 24 | 23 12 | 11 | 10 8 | 7 0 |
| CA3 | Disp5 | Offset12 | ~ | CC3 | 37h8 |

## BOD – Branch if Odd

**Description**:

Branch if the condition code register indicates odd.

**Target Address Calculation**:

The lower 12 bits of the target address come from the offset field in the instruction. The middle 20 bits of the target address come from the sum of sign extended displacement field of the instruction shifted left by 12 bits and the code address register bits 12 to 31. The upper 32 bits of the target address come from the upper 32 bits of the code address register.

The program counter register may be used in the calculation.

**Instruction Format (DPO)**:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 29 | 28 24 | 23 12 | 11 | 10 8 | 7 0 |
| CA3 | Disp5 | Offset12 | ~ | CC3 | 36h8 |

## BRA – Branch Always

**Description**:

Unconditionally branch.

**Target Address Calculation**:

The lower 12 bits of the target address come from the offset field in the instruction. The middle 20 bits of the target address come from the sum of sign extended displacement field of the instruction shifted left by 12 bits and the code address register bits 12 to 31. The upper 32 bits of the target address come from the upper 32 bits of the code address register.

**Instruction Format (DPO)**:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 29 | 28 24 | 23 12 | 11 | 10 8 | 7 0 |
| CA3 | Disp5 | Offset12 | ~ | ~3 | 40h8 |

## CMP - Compare

**Description**:

Compare two values and store the relationship result in a condition code register.

**Instruction Format**:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 1211 | 10 8 | 7 0 |
| ~ | Db5 | Da5 | 02 | CC3 | 23h8 |

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 18 | 17 13 | 1211 | 10 8 | 7 0 |
| Immediate14 | Da5 | 0 | CC3 | 03h8 |

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 47 18 | 17 13 | 1211 | 10 8 | 7 0 |
| Immediate30 | Da5 | 0 | CC3 | 13h8 |

## CMPA – Compare Addresses

**Description**:

Compare two address values.

**Instruction Format**:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 1211 | 10 8 | 7 0 |
| ~ | Ab5 | Aa5 | 12 | CC3 | 23h8 |

**Instruction Format**:

This format compares a data and an address register.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 1211 | 10 8 | 7 0 |
| ~ | Db5 | Aa5 | 22 | CC3 | 23h8 |

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 18 | 17 13 | 1211 | 10 8 | 7 0 |
| Immediate14 | Aa5 | 12 | CC3 | 03h8 |

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 47 18 | 17 13 | 1211 | 10 8 | 7 0 |
| Immediate30 | Aa5 | 12 | CC3 | 13h8 |

LW – Load word

Description:

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| 15 13 | 12 8 | 7 0 |  |
| Disp3 | Dt5 |  | LDW Dt,d[SP] |

|  |  |  |  |
| --- | --- | --- | --- |
| 15 13 | 12 8 | 7 0 |  |
| Disp3 | Dt5 |  | LDW Dt,d[FP] |

|  |  |  |  |
| --- | --- | --- | --- |
| 15 12 | 11 8 | 7 0 |  |
| Disp4 | At4 |  | LA At,d[SP] |

|  |  |  |  |
| --- | --- | --- | --- |
| 15 12 | 11 8 | 7 0 |  |
| Disp4 | At4 |  | LA At,d[FP] |

|  |  |  |  |
| --- | --- | --- | --- |
| 23 13 | 12 8 | 7 0 |  |
| Disp11 | Dt5 |  | LDW Dt,d[SP] |

|  |  |  |  |
| --- | --- | --- | --- |
| 23 13 | 12 8 | 7 0 |  |
| Disp11 | Dt5 |  | LDW Dt,d[FP] |

|  |  |  |  |
| --- | --- | --- | --- |
| 23 12 | 11 8 | 7 0 |  |
| Disp12 | At4 |  | LA At,d[SP] |

|  |  |  |  |
| --- | --- | --- | --- |
| 23 12 | 11 8 | 7 0 |  |
| Disp12 | At4 |  | LA At,d[FP] |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 20 | 19 17 | 16 13 | 12 8 | 7 0 |  |
| Disp12 | Seg3 | Aa4 | Dt5 |  | LDW Dt,d[An] |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 20 | 19 17 | 16 13 | 12 | 11 8 | 7 0 |  |
| Disp12 | Seg3 | Aa4 | ~ | At4 |  | LW At,d[An] |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 20 | 19 17 | 16 13 | 12 8 | 7 0 |  |
| Offset6 | AD | Xn5 | Seg3 | Aa4 | Dt5 |  | LW Dt,o[An+Xn] |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 20 | 19 17 | 16 13 | 12 | 11 8 | 7 0 |  |
| Offset6 | AD | Xn5 | Seg3 | Aa4 | ~ | At4 |  | LW At,o[An+Xn] |

## JMP – Jump to Address

**Description**:

The program counter is loaded with the target address specified in the instruction. The code segment is set to the specified segment.

**Instruction Format**:

|  |  |  |
| --- | --- | --- |
| 47 32 | 11 8 | 7 0 |
| Address36 | Sn4 | 11h8 |

**Operation**:

PC[35..0] = Address

CS = Sn

## JSR – Jump to Subroutine

**Description**:

The current program counter and code segment are copied to the return address and return code segment registers. The program counter is loaded with the target address specified in the instruction. The code segment is set to the segment specified in the instruction.

**Instruction Format**:

|  |  |  |
| --- | --- | --- |
| 47 12 | 11 8 | 7 0 |
| Address36 | Sn4 | 20h8 |

**Operation**:

RS:RA = CS:PC

PC[35..0] = Address

CS = Sn

## LB – Load Byte (8 bits)

**Description**:

Loads a byte (8-bit) value into a data register using indexed addressing. The primary instruction format is one of indexed addressing plus displacement. Four registers are summed in order to calculate the linear address. An address register, an index register, a displacement from the instruction and a segment base register.

Bytes may be loaded only into data registers.

**Instruction Format**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 32 | 31 29 | 2826 | 25 20 | 19 14 | 13 8 | 7 0 |
| Displacement16 | Sn3 | Sc3 | Xb6 | Xa6 | Xt6 | 80h8 |

## LGDT – Load Global Descriptor Table

**Description**:

Loads the global descriptor table pointer from a record in memory.

**Instruction Format**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 32 | 31 29 | 28 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Displacement16 | Sn3 | Sc3 | Xb6 | Xa6 | 06 |  |

**Memory Layout**:

|  |  |  |  |
| --- | --- | --- | --- |
| n | Table Address63..0 | | |
| n+8 | Table Length19..0 | ~12 | Table Address95..64 |

## LH – Load Half-word (16 bits)

**Description**:

Loads a half-word (16-bit) value into a data register using indexed addressing. The primary instruction format is one of indexed addressing plus displacement. Four registers are summed in order to calculate the linear address. An address register, an index register, an offset from the instruction and a segment base register.

Half-words may be loaded only into data registers.

**Instruction Format**:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 30 | 29 27 | 26 24 | 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| Displacement18 | Sn3 | Sc3 | AD | Xn5 | Aa5 | Dt5 | 82h8 |

## LIDT – Load Interrupt Descriptor Table

**Description**:

Loads the interrupt descriptor table pointer from a record in memory.

**Instruction Format**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 32 | 31 29 | 28 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Displacement16 | Sn3 | Sc3 | Xb6 | Xa6 | 16 |  |

**Memory Layout**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | Table Address63..0 | | | |
| n+8 | 06 | Length13..0 | ~12 | Table Address95..64 |

## LD – Load Double Word (64 bits)

**Description**:

Loads a double-word (64-bit) value into a data register using indexed addressing. The primary instruction format is one of indexed addressing plus displacement. Four registers are summed in order to calculate the linear address. An address register, an index register, an offset from the instruction and a segment base register.

There are few additional forms of this instruction that make assumptions about the registers in use in order to shorten the instruction.

**Instruction Format**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 32 | 31 29 | 28 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Displacement16 | Sn3 | Sc3 | Xb6 | Xa6 | Xt6 | 86h8 |

**Instruction Format**:

This instruction format is the same as the previous one except without a displacement field.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3129 | 2826 | 25 20 | 19 14 | 13 8 | 7 0 |
| Sn3 | Sc3 | Xn6 | Xa6 | Xt6 | 96h8 |

**Instruction Format LD Dt,d[SP]**:

This instruction formats assumes the stack pointer is used in the address calculation and assumes the stack segment is also used. The displacement is shifted left three times.

|  |  |  |
| --- | --- | --- |
| 23 14 | 13 8 | 7 0 |
| Disp10 | Xt6 | A6h8 |

**Instruction Format LD Dt,d[FP]**:

This instruction formats assumes the frame pointer is used in the address calculation and assumes the stack segment is also used. The displacement is shifted left three times.

|  |  |  |
| --- | --- | --- |
| 23 14 | 13 8 | 7 0 |
| Disp10 | Xt6 | B6h8 |

**Instruction Format**:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 23 | 22 20 | 19 14 | 13 8 | 7 0 |  |
| Disp9 | Seg3 | Xa6 | Xt6 |  | LD Dt,d[An] |

## LDIS – Load Immediate into Selector

**Description**:

Loads a 24-bit selector value into one of the selector registers. Also causes a load of the descriptor information from the descriptor table.

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 39 36 | 35 12 | 11 | 10 8 | 7 0 |
| ~ | Immediate24 | ~ | Sg3 |  |

## LUI – Load Upper Immediate

**Description**:

**Instruction Format**:

Loads bits 29 to 63 with a sign extended immediate constant and zeros out the lower 29 bits of the target register. A subsequent ADD or OR instruction may be used to build a 48-bit sign extended constant in a register.

|  |  |  |
| --- | --- | --- |
| 31 13 | 12 8 | 7 0 |
| Immediate19 | Dt5 | 05h8 |

**Instruction Format**:

Loads the upper 35 bits of a register with an immediate constant and zeros out the lower 29 bits of the target register. A subsequent ADD or OR instruction may be used to build a 64-bit constant in a register.

|  |  |  |
| --- | --- | --- |
| 47 13 | 12 8 | 7 0 |
| Immediate35 | Dt5 | 15h8 |

## LUIA – Load Upper Immediate Address

**Description**:

**Instruction Format**:

Loads the upper 35 bits of an address register with an immediate constant and zeros out the lower 29 bits of the target address register. A subsequent ADDA instruction may be used to build a 64-bit constant in a register.

|  |  |  |
| --- | --- | --- |
| 47 13 | 12 8 | 7 0 |
| Immediate35 | At5 | 16h8 |

MOV – Move Register

Description:

Moves values between registers.

Instruction Format:

This format moves from one data register to another. It is an alternate mnemonic for the ADD instruction.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 0 | 05 | Da5 | Dt5 | 24h8 |

Instruction Format:

This format moves from a data register to an address register. It is an alternate mnemonic for the ADD instruction.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 1 | Db5 | 05 | At5 | 24h8 |

## NOP – No Operation

**Description**:

NOP is just a placeholder byte that may be used to align code.

**Instruction Format**:

|  |
| --- |
| 7 0 |
| EAh8 |

## OR – Bitwise Or

**Description**:

Or two values and store the result in a data register. The first operand must be in a data register. The second operand may be in a data register or an immediate value supplied in the instruction.

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| ~ | Db5 | Da5 | Dt5 | 29h8 |

**Instruction Format**:

|  |  |  |  |
| --- | --- | --- | --- |
| 31 18 | 17 13 | 12 8 | 7 0 |
| Immediate14 | Da5 | Dt5 | 09h8 |

**Instruction Format**:

|  |  |  |  |
| --- | --- | --- | --- |
| 47 18 | 17 13 | 12 8 | 7 0 |
| Immediate30 | Da5 | Dt5 | 19h8 |

## ORA – Bitwise ‘Or’ to Address

**Description**:

Or two values and store the result in a data register. The first operand must be in a data register. The second operand may be in a data register or an immediate value supplied in the instruction.

**Instruction Format**:

|  |  |  |  |
| --- | --- | --- | --- |
| 47 18 | 17 13 | 12 8 | 7 0 |
| Immediate30 | Aa5 | At5 | 49h8 |

## PUSH – Push Register

**Description**:

Pushes a register onto the stack.

**Instruction Format**:

|  |  |  |
| --- | --- | --- |
| 1514 | 13 8 | 7 0 |
| 02 | Xn6 | C6h8 |

**Operation**:

SP = SP – 8

Memory[SP] = Xn

## RTS – Return from Leaf Subroutine

**Description**:

Return from a subroutine by restoring the program counter from the return address (RA) code address register and restoring the CS register from RS.

**Instruction Format**:

|  |
| --- |
| 7 0 |
| E2h8 |

**Operation**:

CS:PC = RS:RA

## RTI – Return from Interrupt

**Description**:

Perform a far return operation by restoring the previous code segment and program counter from the exception segment (XS) and exception return address (XP) registers. The internal interrupt stack is popped and the operating level, privilege level, and interrupt mask level are reset to values before the exception occurred. Also clears the least significant bit of the semaphore register (the reservation status bit).

**Instruction Format**:

|  |
| --- |
| 7 0 |
| E3h8 |

**Operation**:

CS:PC = XS:XP

## SB – Store Byte (8 bits)

**Description**:

Stores a byte (8-bit) value from a data register using indexed addressing. The primary instruction format is one of indexed addressing plus displacement. Four registers are summed in order to calculate the linear address. An address register, an index register, a displacement from the instruction and a segment base register.

Bytes may be stored only from data registers.

**Instruction Format**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 32 | 31 29 | 2826 | 25 20 | 19 14 | 13 8 | 7 0 |
| Displacement16 | Sn3 | Sc3 | Xb6 | Xa6 | Xs6 | C0h8 |

## SH – Store Half-word (16 bits)

**Description**:

Stores a half-word (16-bit) value from a data register using indexed addressing. The primary instruction format is one of indexed addressing plus displacement. Four registers are summed in order to calculate the linear address. An address register, an index register, a displacement from the instruction and a segment base register.

Half-words may be stored only from data registers.

**Instruction Format**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 32 | 31 29 | 2826 | 25 20 | 19 14 | 13 8 | 7 0 |
| Displacement16 | Sn3 | Sc3 | Xb6 | Xa6 | Xs6 | C1h8 |

## SD – Store Double Word (64 bits)

**Description**:

Stores a double-word (64-bit) value from a data register using indexed addressing. The primary instruction format is one of indexed addressing plus displacement. Four registers are summed in order to calculate the linear address. An address register, an index register, an offset from the instruction and a segment base register.

There are few additional forms of this instruction that make assumptions about the registers in use in order to shorten the instruction.

**Instruction Format**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 32 | 31 29 | 28 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Displacement16 | Sn3 | Sc3 | Xb6 | Xa6 | Xs6 | C3h8 |

**Instruction Format SD Ds,d[SP]**:

This instruction formats assumes the stack pointer is used in the address calculation and assumes the stack segment is also used. The displacement is shifted left three times.

|  |  |  |
| --- | --- | --- |
| 23 14 | 13 8 | 7 0 |
| Disp10 | Xs6 | D0h8 |

**Instruction Format SD Ds,d[FP]**:

This instruction formats assumes the frame pointer is used in the address calculation and assumes the stack segment is also used. The displacement is shifted left three times.

|  |  |  |
| --- | --- | --- |
| 23 14 | 13 8 | 7 0 |
| Disp10 | Xs6 | D1h8 |

**Instruction Format**:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 20 | 19 17 | 16 13 | 12 8 | 7 0 |  |
| Disp12 | Seg3 | Aa4 | Dt5 |  | LD Dt,d[An] |

## SUB - Subtract

**Description**:

Subtract two values and store the result in a data register. Both operands must be in a data register.

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| ~ | Db5 | Da5 | Dt5 | 25h8 |

**Operation**:

Dt = Da - Db

## SUBA

**Description**:

This instruction converts a difference between two pointers into an index.

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| ~ | Ab5 | Aa5 | Dt5 |  |

## XOR – Bitwise Exclusive Or

**Description**:

Exclusive Or two values and store the result in a data register. The first operand must be in a data register. The second operand may be in a data register or an immediate value supplied in the instruction.

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| ~ | Db5 | Da5 | Dt5 | 2Ah8 |

**Instruction Format**:

|  |  |  |  |
| --- | --- | --- | --- |
| 31 18 | 17 13 | 12 8 | 7 0 |
| Immediate14 | Da5 | Dt5 | 0Ah8 |

**Instruction Format**:

|  |  |  |  |
| --- | --- | --- | --- |
| 47 18 | 17 13 | 12 8 | 7 0 |
| Immediate30 | Da5 | Dt5 | 1Ah8 |

## XORA – Bitwise Exclusive ‘Or’ to Address

**Description**:

Exclusive or two values and store the result in a data register. The first operand must be in a data register. The second operand may be in a data register or an immediate value supplied in the instruction.

**Instruction Format**:

|  |  |  |  |
| --- | --- | --- | --- |
| 47 18 | 17 13 | 12 8 | 7 0 |
| Immediate30 | Aa5 | At5 | 4Ah8 |

MOV2SEGDESC

Description:

Instruction Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 0 | Dst5 | Aa5 | St5 | 24h8 |

|  |  |  |
| --- | --- | --- |
| Dst |  |  |
| 0 | Base low |  |
| 1 | Lower bound |  |
| 2 | Upper bound |  |
| 3 | Base high and access rights |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x32 | BRK |  |  | CMP #14 | ADD #14 | LUI #19 |  | BIT # | AND #14 | OR #14 | XOR #14 |  |  |  | XNOR #14 |  |
| 1x48 | JMP | JMF |  | CMP #30 | ADD #30 | LUI #35 | LUIA #35 |  | AND #30 | OR #30 | XOR #30 |  |  |  | XNOR #30 |  |
| 2x24 | JSR | JSF |  | CMP | ADD | SUB | MUL | DIV | AND | OR | XOR |  | NAND | NOR | XNOR |  |
| 3x | BOD | BNO | BPA | BNP | BVS | BVC | BEQ | BNE | BLT | BGE | BLE | BGT | BLTU | BGEU | BLEU | BGTU |
| 4x | BRA |  |  |  | ADDA #30 |  |  |  | ANDA #30 | ORA #30 | XORA #30 |  |  |  |  |  |
| 5x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 6x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 8x | LB | LBU | LH | LHU | LW | LWU | LD48 Xn | LDU | LQ | LQU | LO |  |  |  |  |  |
| 9x |  |  |  |  |  |  | LD32 Xn |  |  |  |  |  |  |  |  |  |
| Ax |  |  |  |  |  |  | LD24 dSP |  |  |  |  |  |  |  |  |  |
| Bx |  |  |  |  |  |  | LD24 dFP |  |  |  |  |  |  |  |  |  |
| Cx | SB | SH | SW | SD | SQ | SO | PUSH |  |  |  |  |  | DBG |  |  |  |
| Dx | SD24 dSP | SD24 dFP |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Ex8 |  | RTF | RTS | RTI |  |  |  |  |  |  | NOP |  |  |  |  |  |
| Fx |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

# Opcode Map