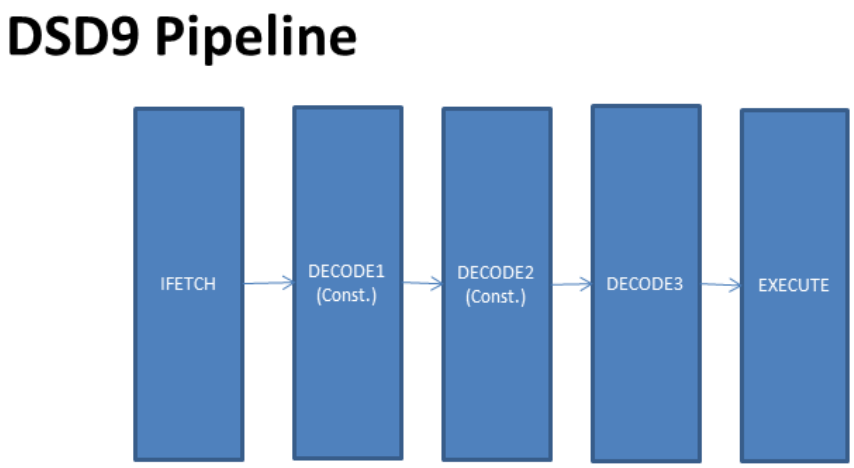
Dark Star * Dragon Nine

General Architecture

Pipeline

DSD9 has a short five stage overlapped pipeline that allows many instructions to execute in a single clock cycle. An I-cache miss will stall the entire pipeline so that no pipeline bubbles can occur in the middle of extended constant. Loads and stores stall the pipeline until the memory operation is complete. Multiply, divide and floating point operations all stall the pipeline for varying numbers of clock cycles depending on the instruction.



There are three stages for decode in order to directly support extended constants where needed. The first two stages of decode are just holding registers to hold onto the constant value for the third stage.

Caching

The core has L1 and L2 caches for instructions. The L1 cache is 2kB and accessible within a single clock cycle. The L2 cache is a 16kiB and requires two additional clock cycles for access. Both caches are direct mapped. The cache line size is 256 bits which is enough room for six instructions.

At reset the cache is loaded with the contents of block of memory ($FFFC0000 to $FFFC3FFF) via a 128 bit bus interface. The core outputs an address sequence on the bus in order to load the cache during reset.

The core has a 16kiB direct mapped data cache. At reset the cache is loaded with the contents of block of memory ($FFFC0000 to $FFFC3FFF). The core outputs an address sequence on the bus in order to load the cache during reset.

Instructions

Instructions are a fixed 40 bits in size. Three forty bit instructions are fit into the lower 120 bits of a 128 bit, 16 byte aligned memory cell. The upper eight bits of the memory are not used and are reserved. Code should be aligned on 16 byte boundaries.

Register File

The core has a 64 entry, 80 bit general purpose register file. r0 always reads as a zero. Several registers have specific uses as outlined later in the document. Registers 60 to 63 are banked for each operating mode.

Floating Point

The core supports basic extended double precision (80 bit) floating point. The floating point unit is clocked at ½ the frequency of the integer portion of the core so that timing delays in the FP unit don't affect the integer portion.

Data / Instruction Granularity

Data and instructions both use a minimum parcel size of 8 bits. Addresses refer to 8 bit quantities.

Endian

The core is a little-endian machine. The least significant byte is stored at the lowest address.

Bus Interface

The core uses an 128 bit wide WISHBONE bus. There are a number of additional signals not part of the WISHBONE spec. WISHBONE signals are identified with an '*' in the table below.

The WISHBONE bus operates as a bus master as detailed in the WISHBONE spec.

WISHBONE	Signal	Width	I/O	Purpose	
	hartid_i	80	I	identifies the core in a multi-core environment	
*	rst_i	1	I	resets the core (synchronous)	
*	clk_i	1	I	clock	
	irq_i	1	I	interrupt request	
	icause_i	9	I	interrupt cause code	
*	cyc_o	1	O	bus cycle is valid	
*	stb_o	1	O	valid data strobe	
*	ack_i	1	I	bus acknowledge	
*	err_i	1	I	bus error occurred (timeout)	
*	lock_o	1	O	indicates the bus should be locked	
*	wr_o	1	O	indicates a write cycle is taking place	

*	sel_o	16	O	byte lane select	
*	adr_o	32	O	address bus	
*	dat_i	128	I	input data	
*	dat_o	128	O	output data	
	sr_o	1	O	set address reservation	
	cr_o	1	O	clear address reservation	
	rb_i	1	I	address reservation status	

hartid_i

This input bus is used to identify the core in a multi-core system. It should be a non-zero value and remain constant while the core is running. The value of this input is reflected in the hartid CSR register.

clk_i

The leading edge of the clock signal is the active edge. Core outputs become valid sometime after the leading edge of the clock. Data is latched into the core on the leading edge of the clock.

sel_o

sel_o identifies which byte of the data bus is active. Data should be reflected across the bus for peripherals that are less than 128 bits wide during a read transaction. During an 8, 16 or 32 bit write transaction the core will reflect data across the bus. For example during a byte write operation the same eight data bits will appear on every byte lane of the bus. Bus reflections are not provided for 40 or 80 bit data. Instead the inactive portions of the bus are set to zero.

adr_o

The adr_o signal indicates which address should be transacted for the memory / I/O system. All addresses are 8 bit references. The core may address up to 4Gi bytes of memory or I/O.

sr_o

sr_o indicates that the memory system should place a reserved status on the address when a read operation takes place. Older reservations may be lost if the memory system depending on how many reservations the memory system can track. This signal is provided for use in multi-core systems.

cr_o

The cr_o signal indicates that the memory system should clear the reserved status on the address when a write operation takes place. This signal is provided for use in multi-core systems.

rbi_i

This signal indicates to the core that the addressed memory cell has a reserved status and if a write to memory was successful. It will be true (1) if the address was still reserved during a write cycle. The core latches the status of the rb_i signal into the SEMA CSR during the SWC instruction.

Operating Modes

The core has four primary operating modes (User, Supervisor, Hypervisor, and Machine). All features (CRS's and instructions) are available to Machine mode. Other modes of operation have more limited capabilities depending on the operating mode.

Privilege Levels

The core supports a 256 level privilege level system. Privilege level zero is assigned to operating mode zero. Privilege level one is assigned to operating mode one. Privilege levels 2 to 7 are assigned to operating mode two. The remaining privilege levels are assigned to operating mode three.

Programming Model

General Purpose Register Array

DSD9 has an array of 64, 80 bit general purpose registers. The registers may hold either integer or floating point data.

	Usage
r0	always zero
r1	return value / exception code
r2	return value / exception type
r3	temporary register
r4	temporary register
r5	temporary register
r6	temporary register
r7	temporary register
r8	temporary register
r9	temporary register
r10	temporary register
r11	register var
r12	register var
r13	register var
r14	register var
r15	register var
r16	register var
r17	register var
r18	register var
r19	
r20	
r21	
r22	
r23	
r24	
r25	
r26	
r27	
r28	
r29	
r30	
r31	

r32	
r33	return value
r34	
r35	temporary register
r36	temporary register
r37	temporary register
r38	temporary register
r39	temporary register
r40	temporary register
r41	temporary register
r42	temporary register
r43	
r44	
r45	
r46	
r47	
r48	
r49	
r50	
r51	
r52	
r53	
r54	
r55	constant
r56	thread pointer
r57	global pointer
r58	exception link register
r59	frame pointer
r60	banked
r61	banked
r62	banked
r63	stack pointer - banked

Register Banking

For purposes of high-speed exception processing a portion of the register file is banked. The last four registers are banked according to the core's operating level. The supervisor has access to the user level registers which are mapped as CSR's . At the machine level all registers are accessible.

Register Usage

Several of the general purpose registers have specific uses.

r0 – always reads as zero. This register may be used where the constant zero is required. For floating point this register reads as positive zero.

r1 – assigned usage is as a return value register. This register also contains the exception code for a locally handled exception. The core loads r1 with the cause code during exception processing.

r2 – has an assigned usage as a return value register. This register is also set to the exception type (24) during exception processing.

r58 – is assigned to be used as the catch handler address register (or exception linkage register). When an exception is routed to local handlers, the core jumps to the address contained in this register.

r59 – has a dedicated use as the stack frame pointer (BP). When referenced in a memory operation causes the core to check the effective address against stack bounds.

r60,r61,r62, and r63 – are a banked registers. There is a separate physical register for each operating level.

r63 – is dedicated to being used as the stack pointer. When referenced in a memory operation causes the core to check the effective address against stack bounds. This register is also automatically updated by stack operations such as push / pop / call / ret.

Instruction Alignment

Instructions must be aligned on an address ending in ‘0’, ‘5’ or ‘A’.

Data Alignment

Data may by aligned on byte address boundaries.

Instruction Cache

The instruction cache line size is 256 bits or about six 40 bit instructions. It may be desirable to align code with the start of the cache line for short loops which will fit entirely in a cache line, to improve performance. Cache lines begin every 16 bytes (address mod 16 = 0).

Special Purpose Registers (Control and Status Registers)

Overview

Most special purpose registers are accessible only at more privileged operating modes. A privilege violation will result if attempting to access a special purpose register in user mode that is not available to that mode.

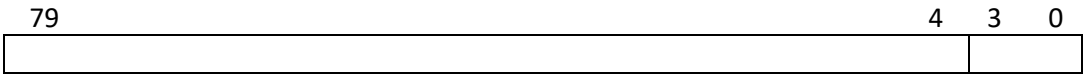
The special purpose or control or status register is identified by a 14 bit constant. The upper two bits of the constant specify which operating mode register is desired. Depending on the specific register requested it may be “banked” for different operating modes. For instance there are three separate banked TVEC (register #004) registers, one for each operating mode above user mode.

There are no results forwarding on the update of a special purpose register. If the value of the register is required immediately in the following few instructions, then some provision must be made to allow the special purpose register to update. This can be done by following a move to the spr with a couple of NOP instructions. Alternately a branch to a delay subroutine could be performed.

Program Counter

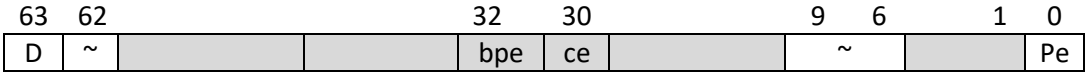
The program counter identifies which instruction to execute. The program counter increments with the lower four bits changing from 0 to 5, 5 to Ah and then back to 0.

The increment may be overridden using one of the flow control instructions. The program counter typically addresses 40 bit instruction parcels using byte addresses.



Control Register Zero (CSR #000)

This register contains a bit to enable protected mode.



D: debug mode status. this bit is set during an interrupt routine if the processor was in debug mode when the interrupt occurred.
PE: Protected Mode enable: 1 = enabled, 0 = disabled.
CE: cache enable: 1=enabled, 0 = disabled
bpe: branch predictor enable: 1=enabled, 0=disabled

Tick Count Register (CSR #002h or TICK)

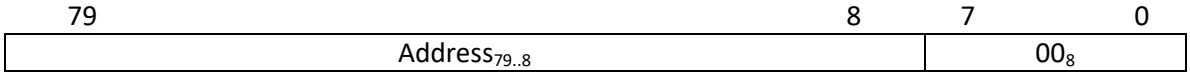
This register contains a count of the number of clock cycles that have passed since the last time the processor was reset. Tick may be used for high-resolution timing or performance measurement.

PCR Paging Control (CSR #003h)

This register controls the paged memory management unit. A more detailed description is available under the section on memory management.

Trap Vector Address (CSR #004h or TVEC)

This register contains the physical base address of the interrupt vector table in memory. There is a separate register for each operating mode. For machine mode the Table is 256B aligned. For hypervisor and supervisor modes this register is byte aligned.



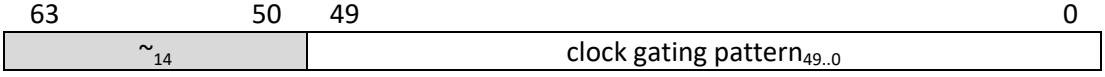
EXROUT (CSR #005h)

This register contains bitfields controlling routing of exceptions in the core. Some exceptions may be routed to a local exception handler in addition to the global exception handler.

Bit		
0	0 = global, 1 = local routing for check instruction	
1	0 = global, 1 = local routing for floating point exceptions	
3-2	00 = ignored, 10 = global routing, 11 = local routing for divide by zero	
5-4	00 = ignored, 10 = global routing, 11 = local routing for multiplier overflow	
7-6	00 = ignored, 10 = global routing, 11 = local routing for add/subtract overflow	

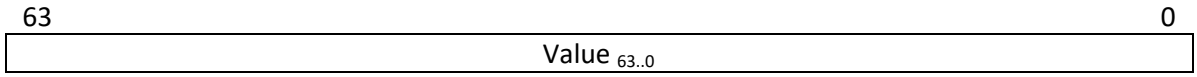
Clock Register (SPR 06)

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.



DBPC (SPR07)

This register stores the return address for a debug interrupt processing routine. This register is automatically loaded when a debug interrupt occurs. The program counter is loaded from this register automatically as part of the RTD instruction processing.



PCR2 Paging Control (CSR #008h)

This register controls the paged memory management unit. A more detailed description is available under the section on memory management.

SEMA (CSR #00Ch) Semaphores

This register is available for system semaphore or flag use. The least significant bit is tied to the reservation address status input (rb_i). It will be set if a STDCR instruction was successful. The least significant bit is also cleared automatically when an interrupt (BRK) or interrupt return (RTI) instruction is executed. Any one of the remaining bits may also be cleared by an RTI instruction. This could be a busy status bit for the interrupt routine. Bits in this CSR may be set or cleared with one of the CSRxx instructions. This register has individual bit set / clear capability.

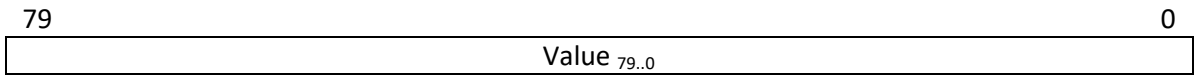
The following is sample code for entrance into a system function.

```
    asm {
                    csrrs   r1,#$0C,#2                  // read status bit and set it (bit mask)
                    and     r1,r1,#2                    // check bit #1
                    beq     r1,r0,.0002                 // if it wasn't already set, okay to process
                    csrrw    r1,#$40,r0                 // get exceptioned PC
                    add      r1,r1,#1                   // increment to skip over static parameter
                    csrrw   r0,#$40,r1                  // write it back
                    csrrw   r0,#$42,#E_Busy             // store busy status in ER1 to be returned in r1
                    rti                                 // leave system busy status bit set

.0002:
        }
… <more code>
    asm {
                    rti    #1                   // clear the system busy bit (bit number)
        }
```

EPC (CSR #040h) Exception Program Counter

This register stack stores the return address for an exception processing routine (OVERFLOW / privilege violation). This register is automatically loaded when an exception occurs. The program counter is loaded from this register automatically as part of the IRET instruction processing. This register stack is five deep. When stack underflow occurs the program counter will be set to the exception stack underflow vector.



STATUS (CSR #041h) Machine Status

This register contains a stack of machine states in bits zero to fifty-five that need to be recovered after an exception occurs. Included in this register are fields for the operating level, privilege level and interrupt mask. This register allows for a five deep stack of machine status bits.

The top of the stack are bits zero to ten. When an interrupt or exception occurs this register is shifted left by eleven bits, and the current interrupt mask, operating level, and privilege level are copied to the top of the stack.

On return from exception or interrupt this register is shifted right by eleven bits and the top of stack copied to the current interrupt mask, operating level and privilege level.

79	7877	76 72	71	7068	6766	6564	63 55	54 11	10 3	2 1	0
SD_1	$\sim_2$	VM_5	$MPRV_1$	$\sim_3$	XS_2	FS_2	$\sim_9$	Stack	PL_8	OL_2	IM_1

VM_5

These bits control virtual memory options. Note that multiple options may be present at the same time. At reset all the bits are set to zero.

Bit		
0	1 = single bound	
1	1 = separate program and data bounds	
2	1 = lot protection system	
3	1 = simplified paged unit	
4	1 = paging unit	

MPRV

This bit when true (1) causes memory operations to use the first stack privilege level when evaluating privilege and protection rules. (Bits 0 to 10 in the status reg).

FS2

These two bits can be used to keep track of the floating point register state.

XS2

These two bits can be used to keep track of an additional core extension state.

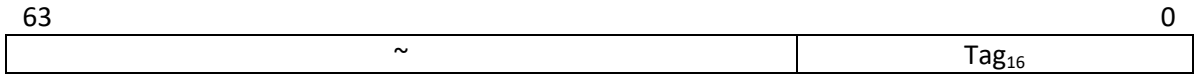
EA (CSR #028h)

This register holds the effective address associated with a memory tag. The tag number is contained in bits 16 to 26. The tag associated with this address will be accessible in the TAGS special purpose register. Note that this register and following tag access should be executed with interrupts disabled to prevent the effective address from changing before the tag is updated or read. Also no memory operation should occur between setting this register and updating or reading the tag. This register also reflects the latest effective address calculated by the processor and will be automatically updated when a memory operation occurs.

63		0
~	tag $number_{11}$	$Offset_{16}$

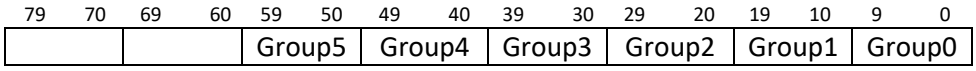
TAGS (CSR #029h)

This register makes the tag value accessible for update or read-back. It is used in association with the EA special purpose register. Writing this register will update the tag identified in the EA register.



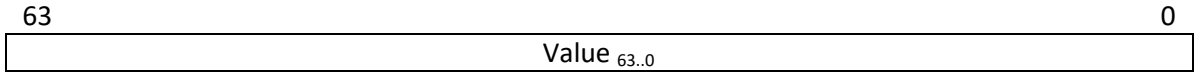
LOTGRP (CSR #02Ah)

This register contains a list of memory groups that the process belongs to. The owning group associated with a memory tag is compared to this list during a memory access. If the group is in the list then the memory access is allowed, otherwise a memory fault exception occurs. This comparison takes place only in user mode; in machine mode the machine owns all of memory so the memory access is always allowed.



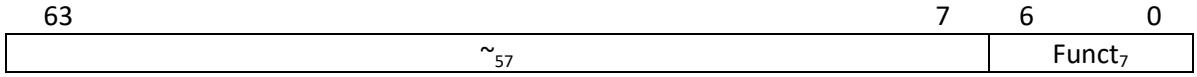
Compare and Swap (SPR44 or CAS)

This register is to support the compare and swap (CAS) instruction. If the value in the addressed memory location identified by the CAS instruction is equal to the value in the CAS register, then the source register is written to the memory location, and the source register is loaded with the value 1. Otherwise if the value in the addressed memory location doesn't match the value in this register, then value at the memory location is loaded into the CAS register, and the source register is set to zero. No write to memory occurs if the match fails.



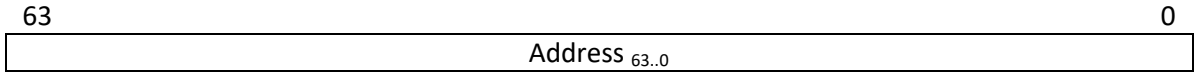
MYST (SPR45)

This register is to supports the MYST instruction. During execution of the MYST instruction the function code of the operation to be performed is loaded from this register. The MYST register is available to both user and kernel modes.



Debug Address Register (SPR50 to SPR53 or DBAD0 to DBAD3)

These registers contain addresses of instruction or data breakpoints.



Debug Control Register (SPR54)

These registers contains bits controlling the circumstances under which a debug interrupt will occur.

<table>
<tr><td>bits</td><td></td><td></td><td></td></tr>
<tr><td>3 to 0</td><td>Enables a specific debug address register to do address matching. If the corresponding bit in this register is set and the address (instruction or data) matches the address in the debug address register then a debug interrupt will be taken.</td><td></td><td></td></tr>
<tr><td>17, 16</td><td>This pair of bits determine what should match the debug address register zero in order for a debug interrupt to occur.
<table>
<tr><td>17:16</td><td></td><td></td></tr>
<tr><td>00</td><td>match the instruction address</td><td></td></tr>
<tr><td>01</td><td>match a data store address</td><td></td></tr>
<tr><td>10</td><td>reserved</td><td></td></tr>
<tr><td>11</td><td>match a data load or store address</td><td></td></tr>
</table></td><td></td><td></td></tr>
<tr><td>19, 18</td><td>This pair of bits determine how many of the address bits need to match in order to be considered a match to the debug address register. These bits are ignored when matching instruction addresses, which are always half-word aligned.
<table>
<tr><td>19:18</td><td></td><td>Size</td></tr>
<tr><td>00</td><td>all bits must match</td><td>byte</td></tr>
<tr><td>01</td><td>all but the least significant bit should match</td><td>char</td></tr>
<tr><td>10</td><td>all but the two LSB’s should match</td><td>half</td></tr>
<tr><td>11</td><td>all but the three LSB’s should match</td><td>word</td></tr>
</table></td><td></td><td></td></tr>
<tr><td>23 to 20</td><td>Same as 16 to 19 except for debug address register one.</td><td></td><td></td></tr>
<tr><td>27 to 24</td><td>Same as 16 to 19 except for debug address register two.</td><td></td><td></td></tr>
<tr><td>31 to 28</td><td>Same as 16 to 19 except for debug address register three.</td><td></td><td></td></tr>
<tr><td>62</td><td>This bit is a history bit for single stepping mode. The debug interrupt records bit 63 into bit 62 when a debug interrupt occurs. Then turns off SSM by writing a zero to bit 63. On return from debug routine (RTD) this bit is restored into bit 63 re-enabling SSM.</td><td></td><td></td></tr>
<tr><td>63</td><td>This bit enables SSM (single stepping mode)</td><td></td><td></td></tr>
</table>

Debug Status Register (SPR55)

This register contains bits indicating which addresses matched. These bits are set when an address match occurs, and must be reset by software.

bit	
0	matched address register zero
1	matched address register one
2	matched address register two
3	matched address register three
63 to 4	not used, reserved

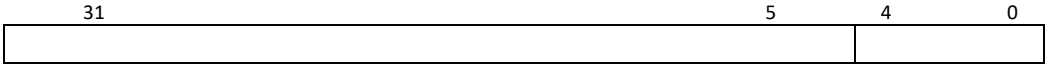
Floating Point Status and Control Register –SPR20

The floating point status and control register may be read using the MFSPR instruction.

Bit		Symbol	Description
31:29	**RM**	rm	rounding mode (unimplemented)
28	**E5**	inexe	- inexact exception enable
27	**E4**	dbzxe	- divide by zero exception enable
26	**E3**	underxe	- underflow exception enable
25	**E2**	overxe	- overflow exception enable
24	**E1**	invopxe	- invalid operation exception enable
23	**NS**	ns	- non standard floating point indicator
Result Status			
22		fractie	- the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception)
21	**RA**	rawayz	rounded away from zero (fraction incremented)
20	**SC**	C	denormalized, negative zero, or quiet NaN
19	**SL**	neg <	the result is negative (and not zero)
18	**SG**	pos >	the result is positive (and not zero)
17	**SE**	zero =	the result is zero (negative or positive)
16	**SI**	inf ?	the result is infinite or quiet NaN
Exception Occurrence			
15	**X6**	swt	{reserved} - set this bit using software to trigger an invalid operation
14	**X5**	inerx	- inexact result exception occurred (sticky)
13	**X4**	dbzx	- divide by zero exception occurred
12	**X3**	underx	- underflow exception occurred
11	**X2**	overx	- overflow exception occurred
10	**X1**	giopx	- global invalid operation exception – set if any invalid operation exception has occurred
9	**GX**	gx	- global exception indicator – set if any enabled exception has happened
8	**SX**	sumx	- summary exception – set if any exception could occur if it was enabled - can only be cleared by software
Exception Type Resolution			
7	**X1T**	cvt	- attempt to convert NaN or too large to integer
6	**X1T**	sqrtx	- square root of non-zero negative
5	**X1T**	NaNCmp	- comparison of NaN not using unordered comparison instructions
4	**X1T**	infzero	- multiply infinity by zero
3	**X1T**	zerozero	- division of zero by zero
2	**X1T**	infdiv	- division of infinities
1	**X1T**	subinfx	- subtraction of infinities
0	**X1T**	snanx	- signaling NaN

CONFIG (CSR #EFFh)

This register contains information controlling the configuration of the core.



CPU Info Registers (CSR #FF0 to #FF7h)

Regno		
FF0h	Manufacturer name – first 10 chars	“Finitron”
FF1h	Manufacturer name – next 10 chars	
FF2h	CPU Class	80 Bit
FF3h	CPU Class	
FF4h	CPU Name	“DSD9”
FF5h	CPU Name	
FF6h	Model Number	1
FF7h	Serial Number	1234
FF8h	Bits 0 to 39 – Instruction cache size	16384
FF8h	Bits 40 to 79 – Data cache size	16384

Floating Point Number Format

The floating point number format used by DSD9 is the extended double precision format:

1	1	14	64 + 1 hidden bit
S_m	S_e	EEEEEEEEEE	.MMMMMMMM…..MMMMMMMM

S_m = sign of mantissa

S_e = sign of exponent

The exponent and mantissa are both represented as two's complement numbers, however the sign bit of the exponent is inverted.

S_eEEEEEEEEE	
11111111111	Maximum exponent
….	
01111111111	exponent of zero
….	
00000000000	Minimum exponent

The exponent ranges from -16382 to +16382

Exceptions

External Interrupts

There is very little difference between an externally generated exception and an internally generated one. An externally caused exception will force a BRK instruction into the instruction stream. The BRK instruction contains a cause code identifying the external interrupt source.

Effect on Machine Status

The operating mode is always switched to the machine mode on exception. It's up to the machine mode code to redirect the exception to a lower operating mode when desired. Further exceptions at the machine mode are disabled automatically. Machine mode code must enable interrupts at some point. This can be done automatically when the exception is redirected to a lower level by the REX instruction. The IRET instruction will also automatically enable further machine level exceptions.

Exception Stack

The program counter and status bits are pushed onto an internal stack when an exception occurs. This stack is only five entries deep as that is the maximum amount of nesting that can occur. Further nesting of exceptions can be achieved by saving the state contained in the exception registers.

Exception Vectoring

Exceptions are handled through a single vector for a given operating level. More specific exception information is supplied in the cause register.

Exception Vector Table

The exception vector table contains instructions used to vector to handling routines. The instructions are typically a jump or branch instruction. The vector table is located by the trap vector address (TVEC) register. This register is set to $FFFC0000 on reset. Note that the reset vector is fixed and cannot be relocated.

Vector Address			
0xFFFC0000	Exception from user level		
0xFFFC0040	Exception from supervisor level		
0xFFFC0080	Exception from hypervisor level		
0xFFFC00C0	Exception from machine level		
0xFF…FC0100	Non-maskable interrupt		
0xFF…FC0140	Reset		

When an exception is redirected to a lower level by the REX instruction the core will jump to the address contained in one of the lower level vector register (TVEC for hypervisor or supervisor level).

Reset

On a reset the core vectors to address $FFFC0000.

Exception Cause Codes

The following table outlines the cause code for a given purpose. These codes are specific to DSD9. Under the HW column an ‘x’ indicates that the exception is internally generated by the processor; the cause code is hard-wired to that use. An ‘e’ indicates an externally generated interrupt, the usage may vary depending on the system.

Cause Code		HW	Description	
0				
1				
2			FMTK Scheduler	
432		e		
433	KRST	e	Keyboard reset interrupt	
434	MSI	e	Millisecond Interrupt	
435	TICK	e	FMTK Tick Interrupt	
…				
463	KBD	e	Keyboard interrupt	
482	TGT	x	call target exception	
483	MEM	x	memory fault	
484	IADR	x	bad instruction address	
485	UNIMP	x	unimplemented instruction	
486	FLT		floating point exception	
487	CHK		bounds check exception	
488	DBZ	x	divide by zero	
489	OFL	x	overflow	
493	FLT	x	floating point exception	
497	EXF	x	Executable fault	
498	DWF	x	Data write fault	
499	DRF	x	data read fault	
500	SGB	x	segment bounds violation	
501	PRIV	x	privilege level violation	
504	STF	x	stack fault	
505	CPF	x	code page fault	
506	DPF	x	data page fault	
508	DBE	x	data bus error	
510	NMI	x	Non-maskable interrupt	

Simplified Paged Memory Management Unit

Overview

The memory management unit is a simplified paged memory management unit. Memory management by the MMU includes virtual to physical address mapping and read/write/execute permissions. The MMU divides memory into 64kB or 4MiB pages depending on the setting in PCR2.

64kiB pages

Processor address bits 16 to 25 are used as a ten bit index into a mapping table to find the physical page. The MMU remaps the ten address bits into a sixteen bit value used as address bits 16 to 31 when accessing a physical address. The lower sixteen bits of the address pass through the MMU unchanged. The maximum amount of memory that may be mapped in the MMU is 64MiB per map out of a pool of 4GiB. Addresses with the most significant six bits set are not mapped.

4MiB pages

Some tasks require a lot of memory and a 64MB map isn't sufficient. For instance, while in machine mode the core requires access to the entire address range. A memory page size of 4MiB may be selected by setting the bit corresponding to the memory map in PCR2.

Processor address bits 22 to 31 are used as a ten bit index into a mapping table to find the physical page. The MMU remaps the ten address bits into a ten bit value used as address bits 22 to 31 when accessing a physical address. The lower 22 bits of the address pass through the MMU unchanged. The maximum amount of memory that may be mapped in the MMU is 4GiB per map out of a pool of 4GiB. Addresses with the most significant six bits set are not mapped.

Map Tables

The mapping tables for memory management are stored directly in the MMU rather than being stored in main memory as is commonly done. The MMU supports up to 64 independent mapping tables. Only a single mapping table may be active at one time. The active mapping table is set in the paging control register (CSR #3) bits 0 to 5 – called the operate key. Mapping tables may be shared between tasks.

Map Caching / TLB

There isn't a need for a TLB or ATC as the entire mapping table is contained in the MMU. A TLB isn't required. Address mapping is still only two cycles.

Operate Key

The operate key controls which mapping table is actively mapping the memory space. The operate key is located in CSR #3 bits 0 to 5. The operate key is similar to an ASID (address space identifier). The operate key is also used as part of the cores cache tags.

Access Key

The MMU mapping tables are present at I/O address $FFDC4000 to $FFDC4FFE. All the mapping tables share the same I/O space. Only one mapping table is visible in the address space at one time. Which table is visible is controlled by an access key. The access key is located in the paging control register (CSR #3) bits 8 to 13.

Address Pass-through

Addresses pass through the MMU unaltered until the mapping enable bit is set. Until mapping is enabled, the physical address will match the virtual address. Additionally address bits 0 to 15 pass through the MMU unaltered.

Mapping Table Layout

	D18	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	
000	W	R	X	PA31	PA30	PA29	PA28	PA27	PA26	PA25	PA24	PA23	PA22	PA21	PA20	PA19	PA18	PA17	PA16	
004	W	R	X	PA31	PA30	PA29	PA28	PA27	PA26	PA25	PA24	PA23	PA22	PA21	PA20	PA19	PA18	PA17	PA16	
	...																			
FFC	W	R	X	PA31	PA30	PA29	PA28	PA27	PA26	PA25	PA24	PA23	PA22	PA21	PA20	PA19	PA18	PA17	PA16	

PAnn = physical address

X = executable page indicator.

W = writeable data page indicator.

R = readable data page indicator.

Note the low order six bits are not used for 4MiB pages.

PCR- Paging Control Register Layout

31	30 14	13 8	7 6	5 0
PE	$\sim_{18}$	$AKey_6$	~	$OKey_6$

PE = Paging Enable (1=enabled, 0 = disabled)

AKey = Access Key

OKey = Operate Key

PCR2 – Page Size

This register controls the memory page size. Each bit in the register corresponds to a memory map. Memory may be paged in either 64kiB or 4MiB pages. All pages in a map have the same size.

Latency

The address map operation when enabled has two cycles of latency.

Memory

Memory Bus Size

The core uses a 128 bit data-bus to interface to external devices. This size was chosen to match the typical size of the dynamic RAM interface. It also allows loading the caches with a minimal number of bus cycles to improve performance. For un-cached data most data is available with just a single bus cycle. When un-cached data is not suitably aligned the core will perform a second memory operation to fetch or store the data.

Bus Organization

The byte, wyde, and tetra-byte store operations fill the data-bus with contiguous multiple copies of the data in the inactive portions of the data-bus. The penta-byte and deci-byte store operations pad the data-bus with zeros for the inactive portions. For instance a byte store operation places the same byte of data on every byte lane during the store. There are then 15 copies + 1 valid byte on the data-bus. A wyde operation would place eight copies of the sixteen bit data on the data-bus. This was done to facilitate the use of the bus by input / output devices. The I/O device has to look only at the low order portion of the data-bus that's relevant to it during a store operation. Otherwise it would be necessary to externally shift the data onto the low order bits for the I/O device. While there are tons of 8, 16, or 32 bit peripheral cores there are not many 40 or 80 bit cores so output to storage for those sizes was simplified.

For input the I/O subsystem should reflect the data from the I/O device across the width of the data-bus. For instance eight copies of wyde data should be placed on the 128 bit data bus by the I/O subsystem. This is required because the core looks only at the portion of the data-bus that is active according to the address and data size.

Interrupts

FISA64 uses a vectored interrupt system with support for 512 interrupt vectors.

Interrupt Vector Table Usage

The following table outlines which vector is used for a given purpose. These vectors are specific to FISA64. Under the HW column an 'x' indicates that the interrupt is internally generated by the processor; the vector is hard-wired to that use. An 'e' indicates an externally generated interrupt, the usage may vary depending on the system.

Vecno		HW	Description	
0				
1				
2			FMTK Scheduler	
3			debug interrupt	

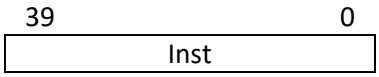
4			OS API call	
449	KRST	e	Keyboard reset interrupt	
450	MSI	e	Millisecond Interrupt	
451	TICK	e	FMTK Tick Interrupt	
463	KBD	e	Keyboard interrupt	
487	BND	x	Bounds check exception	
488	DBZ	x	divide by zero	
489	OFL	x	overflow	
493	FLT	x	floating point exception	
494	TAP	x	debug tap interrupt	
495	SSM	x	single-step interrupt	
496	BPT	x	breakpoint	
497	EXF	x	Executable fault	
498	DWF	x	Data write fault	
499	DRF	x	data read fault	
501	PRIV	x	privilege level violation	
508	DBE	x	data bus error	
509	IBE	x	instruction bus error	
510	NMI	x	Non-maskable interrupt	

Instruction Set Description

A description of the instruction set follows.

Instruction Size

Instructions are primarily 40 bits in size.



Constant Extensions

Constants may be extended up to 80 bits using constant post-words.

Shown below are the instruction formats for the 'ADD' immediate instruction with 20, 56, and 92 bits.

20 bit constant

39 — 20	19 — 14	13 — 8	7 — 0	
$Immediate_{20}$	Rt_6	Ra_6	04_8	ADD Rt,Ra,#imm

56 bit constant

$Immediate_{19..0}$	Rt_6	Ra_6	04_8		ADD Rt, #imm
$Immediate_{55..24}$			Ch_4	I_4	

92 bit constant

$Immediate_{19..0}$	Rt_6	Ra_6	04_8		ADD Rt, #imm
$Immediate_{55..24}$			Ch_4	I_4	
$Immediate_{91..60}$			Ch_4	I_4	

If a constant is encountered in the instruction stream it will be treated as a NOP operation. Normally the processor advances past constants as they are encountered.

Arithmetic Instructions

Arithmetic instructions include add, subtract, multiply and divide. There are signed and unsigned versions of all arithmetic instructions.

Logical / Bitwise Instructions

There is a group of bitwise instructions which may also be used for logical operations. These instructions include and, or, exclusive or, and their inverses.

Branches

Branches integrate a compare and a branch operation into a single instruction. Both compare to register and compare to immediate value are supported for both signed and unsigned values. The branch architecture is quite powerful and although the instructions are 40 bits they may replace up to two instructions on other architectures. Consider an architecture that doesn't support branching based on compare to immediates. It must first use a compare instruction, then a branch instruction. This may actually require 64 bits and two clock cycles to do in a 32 bit architecture.

Branches make use of a (2,2) correlating branch predictor with a 512 entry history table. Most branches can be completed in a single cycle.

Branch based on compare to register have the option of specifying the branch prediction. If the branch prediction is specified then the branch does not consume room in the branch predictor tables.

ADD - Addition

ADD Rt, Ra, #i20
ADD Rt, Ra, Rb

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
$04h_8$	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	ADD Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	04_8	ADD Rt,Ra,#imm
		$Immed_{10}$		Rt_6	$31h_8$	ADD Rt,#imm10

Operation:

Register Immediate Form

Rt = Ra + $immediate_{20}$

Register-Register Form

Rt = Ra + Rb

Notes*:*

The immediate constant may be extended up to 80 bits.

Note there is a compressed 24 bit version of the immediate add instruction.

ADDO – Addition with Overflow Detect

ADD Rt, Ra, #i20
ADD Rt, Ra, Rb

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
$14h_8$	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	ADD Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	14_8	ADD Rt,Ra,#imm

Operation:

Register Immediate Form

$Rt = Ra + immediate_{20}$

Register-Register Form

$Rt = Ra + Rb$

Exceptions: adder/subtracter overflow if enabled in EXROUT

Notes:

The immediate constant may be extended up to 80 bits.

Note there is a compressed 24 bit version of the immediate add instruction.

AND – Bitwise ‘and’

AND Rt, Ra, #i20
AND Rt, Ra, Rb

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
$08h_8$	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	AND Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	08_8	AND Rt,Ra,#imm

Operation:

Register Immediate Form

Rt = Ra & immediate

Register-Register Form

Rt = Ra & Rb

Notes*:*

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

ASL – Arithmetic Shift Left Register

Description:

Shift register to the left by an amount specified in either a second register or a constant in the instruction and place the result in the target register. A Zero is shifted into bit 0.

Instruction Format:

39 32	31 26		25 20	19 14	13 8	7 0	
$32h_8$	$\sim_6$		Rt_6	Rb_6	Ra_6	02_8	ASL Rt,Ra,Rb
$42h_8$	$\sim_5$	I_1	Rt_6	$Immed_6$	Ra_6	02_8	ASL Rt,Ra,#i6

Operation:

Rt = Ra << Rb

Clock Cycles: 1

Exceptions: overflow if the sign bit changes

Notes:

This instruction performs the same operation as the SHL instruction except that it may cause an overflow exception.

ASR – Arithmetic Shift Right

ASR Rt, Ra, #i7
ASR Rt, Ra, Rb

Description:

Shift register to the right preserving the sign bit, by an amount specified in either a second register or a constant in the instruction and place the result in the target register. The value of the most significant bit is unchanged.

Instruction Formats:

39 32	31 26		25 20	19 14	13 8	7 0	
$33h_8$	$\sim_6$		Rt_6	Rb_6	Ra_6	02_8	ASR Rt,Ra,Rb
$43h_8$	$\sim_3$	I_1	Rt_6	$Immed_6$	Ra_6	02_8	ASR Rt,Ra,#i6

Operation:

Register Immediate Form

Rt = Ra >> $immediate_7$

Register-Register Form

Rt = Ra >> Rb

Notes:

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

BFCHG – Bitfield Change

Description:

Inverts the bitfield in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

Instruction Format:

39 36	35	34 28	27	26 20	19 14	13 8	7 0
2_4	$\sim_1$	me_7	~	mb_7	Rt_6	Ra_6	$13h_8$

BFCLR – Bitfield Clear

Description:

Sets the bits to zero of the bitfield in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

Instruction Format:

39 36	35	34 28	27	26 20	19 14	13 8	7 0
1_4	$\sim_1$	me_7	~	mb_7	Rt_6	Ra_6	$13h_8$

BFEXT – Bitfield Extract

Description:

Extracts a bitfield from register Ra located between the mask begin (mb) and mask end (me) bits and places the sign extended result into the target register. This instruction may be used to sign extend a value beginning at any bit.

Instruction Format:

39 36	35	34 28	27	26 20	19 14	13 8	7 0
5_4	$\sim_1$	me_7	~	mb_7	Rt_6	Ra_6	$13h_8$

BFEXTU – Bitfield Extract Unsigned

Description:

Extracts a bit-field from register Ra located between the mask begin (mb) and mask end (me) bits and places the zero extended result into the target register. This instruction may be used to zero extend a value beginning at any bit.

39 36	35	34 28	27	26 20	19 14	13 8	7 0
6_4	$\sim_1$	me_7	~	mb_7	Rt_6	Ra_6	$13h_8$

BFINS – Bitfield Insert

Description:

Inserts a bitfield into the target register located between the mask begin (mb) and mask end (me) bits from the low order bits of Ra or an immediate value.

Instruction Format:

39 36	35	34 28	27	26 20	19 14	13 8	7 0
3_4	$\sim_1$	me_7	~	mb_7	Rt_6	Ra_6	$13h_8$
4_4	$\sim_1$	me_7	~	mb_7	Rt_6	Imm_6	$13h_8$

BFSET – Bitfield Set

Description:

Sets the bits to one of the bitfield in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

Instruction Format:

39 36	35	34 28	27	26 20	19 14	13 8	7 0
0_4	$\sim_1$	me_7	~	mb_7	Rt_6	Ra_6	$13h_8$

BBC – Branch on Bit Clear

Description:

Branch if a bit in a register is false. The branch range is approximately +/- 32k bytes.

Instruction Formats:

$Target_{15..0}$	P_2	$\sim_1$	$Bitno_7$	Ra_6	$54h_8$

Operation:

```
if (~Ra[bitno])
        pc <= pc + sign extended Target;
```

Clock Cycles: 1 if prediction is true, 3 otherwise

BBS – Branch on Bit Set

Description:

Branch if a bit in a register is true. The branch range is approximately +/- 128k bytes.

Instruction Formats:

$Target_{15..0}$	P_2	$\sim_1$	$Bitno_7$	Ra_6	$55h_8$

Operation:

```
if (~Ra[bitno])
        pc <= pc + sign extended Target;
```

Clock Cycles: 1 if prediction is true, 3 otherwise

Bcc – Branch on Compare to Register

Description:

Branch if a comparison condition between a register and another register value is true. The branch range is approximately +/- 32k bytes.

These instructions allow a static branch prediction to be specified. If the branch prediction is statically specified then the branch does not consume room in the branch history table. This may improve the accuracy of predicted branches.

Instruction Formats:

$Target_{15..0}$	P_2	$\sim_2$	Rb_6	Ra_6	$Opcode_8$

Operation:

```
if (Ra cond Rb)
        pc <= pc + sign extended Target;
```

Clock Cycles: 1 if prediction is true, 3 otherwise

Conditions:

$Opcode_8$	Mne.	Description
46	BEQ	branch if equal
47	BNE	not equal
48	BLT	signed less than
49	BGE	signed greater than or equal
4A	BLE	signed less than or equal
4B	BGT	signed greater than
4C	BLTU	unsigned less than
4D	BGEU	unsigned greater than or equal
4E	BLEU	unsigned less than or equal
4F	BGTU	unsigned greater than

P_2	Meaning
0	don't predict
1	reserved
2	predict not taken
3	predict taken

BccI – Branch on Compare to Immediate

Description:

Branch if a comparison condition between a register and an eight bit immediate value is true. The branch range is approximately +/- 32k bytes. The immediate value is sign extended before use. Constant postfix words may be used to extend the immediate to 64 bits.

Instruction Formats:

$Target_{15..0}$	P_2	$Imm_{7..0}$	Ra_6	$Opcode_8$

Operation:

```
if (Ra cond #imm)
        pc <= pc + displacement
```

Clock Cycles: 1 if prediction is true, 3 otherwise

Conditions:

$Opcode_8$	Mne.	Description
56	BEQ #	branch if equal
57	BNE #	not equal
58	BLT #	signed less than
59	BGE #	signed greater than or equal
5A	BLE #	signed less than or equal
5B	BGT #	signed greater than
5C	BLTU #	unsigned less than
5D	BGEU #	unsigned greater than or equal
5E	BLEU #	unsigned less than or equal
5F	BGTU #	unsigned greater than

BRA – Branch Unconditionally

BRA target_address

Instruction Formats:

$Target_{15..0}$	3_2	$\sim_2$	0_6	0_6	$46h_8$

Operation:

Notes:

This instruction is typically encoded with a static prediction of taken.

BRK – Breakpoint Exception

BRK address

Instruction Formats:

23	22 17	16 8	7 0
N	$\sim_6$	$Cause_9$	00_8

Operation:

Notes:

The ‘N’ bit indicates whether to return to the next address (1) or the address of the breakpoint (0).

Perform a global interrupt, exception or debug handler. The BRK instruction is used by hardware interrupts to call a hardware interrupt processing routine. The appropriate return instruction (RTI) should be used to return from the BRK handler. The BRK instruction causes the processor to switch to machine mode. Operating level zero is set along with privilege level zero. The BRK handler may redirect exceptions to a lower operating level (numerically higher) using the REX instruction.

The status register and program counter are pushed to internal stacks. Further interrupts are automatically disabled. Additionally the cause register is loaded with the value specified in the instruction.

CALL – Call Subroutine

CALL (abs[Rn])

CALL d(Rn)

Description:

This instruction performs a subroutine call operation. First the return address is pushed onto the stack (the address of the next instruction). Next the program counter is loaded from the specified operand.

If register 63 is specified for Ra, then the address of the instruction is substituted for use in calculating the target address. This allows program counter relative addressing to be used.

Instruction Formats:

$Immed_{20}$	0_6	Ra_6	$50h_8$	CALL
$Immed_{20}$	0_6	Ra_6	$51h_8$	CALLI

Operation:

Memory Indexed Indirect Form

SP = SP – 10
memory[SP] = return address
PC = memory[address + Rn]

Register Indirect with Displacement Form

SP = SP – 10
memory[SP] = return address
PC = displacement + Rn

Notes:

The address constant may be extended up to 80 bits with immediate postfix instructions.

CAS – Compare and Swap

CAS R1,R2,d[R4]

Instruction Format:

$Disp_{15}$	Rst_5	Ra_5	$6C_7h$	CAS

Operation:

```
if memory[Ra+displacement] = casreg
        memory[Ra + displacement] = Rst
        Rst = 1
else
        casreg = memory [Ra + displacement]
        Rst = 0
```

Description:

If the contents of the addressed memory cell is equal to the contents of CAS special purpose register then a sixty-four bit value is stored to memory from the source register Rst and Rst is set equal to one. Otherwise Rst is set to zero and the contents of the memory cell is loaded into CAS. The memory address is the sum of the sign extended displacement and register Ra. 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. Note that the memory system must support bus locks in order for this instruction to work as expected.

This instruction is typically used to implement semaphores. The LWAR and SWCR may also be used to perform a similar function where the memory system does not support bus locks, but support address reservations instead.

Assembler:

CAS Rt,Rt,displacement[Ra]

CHK – Check and Exception

CHK Rt, Ra, Bn

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
$0Bh_8$	$\sim_6$	Rc_6	Rb_6	Ra_6	02_8	CHK Ra,Rb,Rc
$Immediate_{20}$			Rb_6	Ra_6	$0B_8$	CHK Ra,Rb,#imm

Operation:

```
if not (Ra >= Rb and Ra < Rc or immediate)
   take bounds exception
```

Notes:

This instruction may be used to validate a pointer or array index.

A register is checked against the upper and lower bounds contained in a register identified by the instruction. If the register is not between the upper and lower bounds then a bounds check exception is taken.

The immediate constant may be extended up to 80 bits with immediate prefix instructions.

This exception may be routed to either a local or global exception handler depending on the setting in the EXROUT CSR.

CMP - Comparison

CMP Rt, Ra, #i20
CMP Rt, Ra, Rb

Description

CMP performs a signed comparison of operands and sets the target register to -1, 0, or +1 if the first operand is less than, equal to, or greater than the second respectively.

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

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
$06h_8$	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	CMP Rt,Ra,Rb
$Immediate_{20}$			Rb_6	Ra_6	06_8	CMP Rt,Ra,#imm

Operation:

Register Immediate Form

```
if (Ra < immediate)
   Rt = -1
else if (Ra = immediate)
   Rt = 0
else
   Rt = 1
```

Register-Register Form

```
if (Ra < Rb)
   Rt = -1
else if (Ra = Rb)
   Rt = 0
else
   Rt = 1
```

CMPU – Unsigned Comparison

CMPU Rt, Ra, #i20
CMPU Rt, Ra, Rb

Description

CMPU performs a unsigned comparison of operands and sets the target register to -1, 0, or +1 if the first operand is less than, equal to, or greater than the second respectively.

The immediate constant may be extended up to 80 bits with immediate postfix instructions.

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
$06h_8$	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	CMP Rt,Ra,Rb
$Immediate_{20}$			Rb_6	Ra_6	06_8	CMP Rt,Ra,#imm

Operation:

Register Immediate Form

```
if (Ra < immediate)
   Rt = -1
else if (Ra = immediate)
   Rt = 0
else
   Rt = 1
```

Register-Register Form

```
if (Ra < Rb)
   Rt = -1
else if (Ra = Rb)
   Rt = 0
else
   Rt = 1
```

COM – Bitwise Ones Complement

COM Rt, Ra

Description

All the bits in Ra are inverted and placed into the target register Rt. This is an alternate mnemonic for the XOR instruction.

Instruction Formats:

39 20	19 14	13 8	7 0	
$FFFFFh_{20}$	Rt_6	Ra_6	$0A_8$	XOR Rt,Ra,#imm

Operation:

Register Immediate Form

Rt = Ra ^ -1

Notes*:*

CSEV – Conditional Set if Even

CSEV Rt, Ra, #i20
CSEV Rt, Ra, Rb

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
67_8	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	CSEV Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	67_8	CSEV Rt,Ra,#imm

Operation:

Register Immediate Form

```
If Ra mod 2 = 0
    Rt = immediate
```

Register-Register Form

```
If Ra mod 2 = 0
    Rt = Rb
```

Notes*:*

The immediate constant may be extended up to 80 bits.

CSN – Conditional Set if Negative

CSN Rt, Ra, #i20
CSN Rt, Ra, Rb

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
62_8	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	CSN Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	62_8	CSN Rt,Ra,#imm

Operation:

Register Immediate Form

If Ra<0
Rt = immediate

Register-Register Form

If Ra<0
Rt = Rb

Notes*:*

The immediate constant may be extended up to 80 bits.

CSNN – Conditional Set if Non-Negative

CSN Rt, Ra, #i20
CSN Rt, Ra, Rb

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
63_8	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	CSN Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	63_8	CSN Rt,Ra,#imm

Operation:

Register Immediate Form

```
If Ra >= 0
        Rt = immediate
```

Register-Register Form

```
If Ra >= 0
        Rt = Rb
```

Notes*:*

The immediate constant may be extended up to 64 bits.

CSNP – Conditional Set if Not Positive

CSNP Rt, Ra, #i20
CSNP Rt, Ra, Rb

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
65_8	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	CSNP Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	65_8	CSNP Rt,Ra,#imm

Operation:

Register Immediate Form

```
If Ra<=0
    Rt = immediate
```

Register-Register Form

```
If Ra<=0
    Rt = Rb
```

Notes*:*

The immediate constant may be extended up to 80 bits.

CSNZ – Conditional Set if Non-Zero

CSNZ Rt, Ra, #i20
CSNZ Rt, Ra, Rb

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
61_8	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	CSZ Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	61_8	CSZ Rt,Ra,#imm

Operation:

Register Immediate Form

```
If Ra<>0
    Rt = immediate
```

Register-Register Form

```
If Ra<>0
    Rt = Rb
```

Notes*:*

The immediate constant may be extended up to 80 bits.

CSOD – Conditional Set if Odd

CSO Rt, Ra, #i20
CSO Rt, Ra, Rb

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
66_8	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	CSOD Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	66_8	CSOD Rt,Ra,#imm

Operation:

Register Immediate Form

```
If Ra mod 2 = 1
    Rt = immediate
```

Register-Register Form

```
If Ra mod 2 = 1
    Rt = Rb
```

Notes*:*

The immediate constant may be extended up to 80 bits.

CSP – Conditional Set if Positive

CSP Rt, Ra, #i20

CSP Rt, Ra, Rb

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
64_8	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	CSP Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	64_8	CSP Rt,Ra,#imm

Operation:

Register Immediate Form

```
If Ra>0
    Rt = immediate
```

Register-Register Form

```
If Ra>0
    Rt = Rb
```

Notes*:*

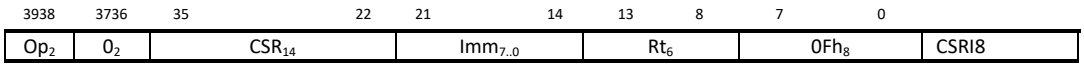
The immediate constant may be extended up to 80 bits.

CSR – Control and Status Register Update

Description:

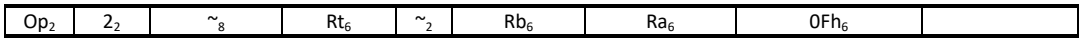
This instruction atomically reads the CSR into a target register then sets it to either an immediate value supplied by the instruction or a value supplied by a register. Individual bits in the CSR may be set or cleared by the CSRRSI and CSRRCI instructions. Which CSR register to access may be specified by an immediate constant in the instruction, or by the contents of a register.

Instruction Formats:

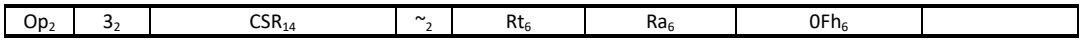


Immediate to CSR #

Immediate to CSR Ra



Register to CSR Ra



Register to CSR #

Op_2	Mne.	Description
0	CSRRWI	Write the entire value of immediate to the CSR
1	CSRRSI	Set the bits in the CSR according to the bits set in the immediate
2	CSRRCI	Clear the bits in the CSR according to the bits set in the immediate
3		not used

Note that not all CSR's support the CSRRS and CSRRC instructions.

CSR's are determined by the lower 12 bits of the CSR field in the instruction. The upper two bits of the CSR field are reserved, and may be used in the future to resolve the core's operating level.

CSZ – Conditional Set if Zero

CSZ Rt, Ra, #i20
CSZ Rt, Ra, Rb

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
60_8	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	CSZ Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	60_8	CSZ Rt,Ra,#imm

Operation:

Register Immediate Form

If Ra=0
Rt = immediate

Register-Register Form

If Ra=0
Rt = Rb

Notes*:*

The immediate constant may be extended up to 64 bits.

DIV – Signed Division

DIV Rt, Ra, #i20
DIV Rt, Ra, Rb

Description:

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
$18h_8$	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	DIV Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	18_8	DIV Rt,Ra,#imm

Operation:

Register Immediate Form

Rt = Ra / immediate

Register-Register Form

Rt = Ra / Rb

Notes*:*

The immediate constant may be extended up to 80 bits.

The signed registered form of the instruction may generate a divide by zero exception if the divisor is zero. All other forms of the instruction including signed division by a constant never generate any exceptions.

DIVU – Unsigned Division

DIVU Rt, Ra, #i20
DIVU Rt, Ra, Rb

Description:

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
$19h_8$	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	DIVU Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	19_8	DIVU Rt,Ra,#imm

Operation:

Register Immediate Form

Rt = Ra / immediate

Register-Register Form

Rt = Ra / Rb

Notes*:*

The immediate constant may be extended up to 80 bits.

FBcc – Float Branch on Compare to Register

Description:

Branch if the floating point comparison condition between a register and another register value is true. The branch range is approximately +/- 32k bytes.

Instruction Formats:

$Target_{15..0}$	P_2	S_2	Rb_6	Ra_6	$Opcode_8$

Operation:

```
if (Ra cond #imm)
        pc <= pc + displacement
```

Clock Cycles: 3 if branch is taken, otherwise 1

Conditions:

$Opcode_8$	Mne.	Description
36	FBEQ	branch if equal
37	FBNE	not equal
38	FBLT	less than
39	FBGE	greater than or equal
3A	FBLE	less than or equal
3B	FBGT	greater than
3C	FBOR	ordered
3D	FBUN	unordered

S_2	Precision (Size)
0	single
1	double
2	reserved
3	quad

IMM – Immediate Postfix

IMM #i36

Instruction Formats:

$Constant_{35..4}$	Ch_4	C_4	IMM

Operation:

Notes:

The IMM prefix appends 36 bits onto the 20 bit or 8 bit constant field of the previous instruction then sign extends the resulting 56 bit constant out to 80 bits. Two immediate prefix instructions may be used in succession in order to append up to 80 bits onto the constant field of the following instruction. Thus a full 80 bit constant may be used by most instructions.

The immediate postfix may not be used to extend the range of a branch instruction. If there is an immediate postfix applied to an instruction that doesn't use a constant, then the postfix will be ignored.

INC – Increment memory word

INC d(Rn),#n

Instruction Formats:

$Displacement_{15}$	Imm_5	Ra_5	64_7h	INC d15(Rn),#n

Operation:

Register Indirect with Displacement Form

memory[displacement + Ra] = memory[displacement + Ra] + n

Notes:

Increments the memory word by a signed five bit immediate constant. The displacement constant may be extended up to 64 bits with immediate prefix instructions.

JAL – Jump and Link

JAL Rd,(abs,Rn)
JAL Rd,d(Rn)

Instruction Formats:

$Immed_{20}$	Rt_6	Ra_6	$40h_8$	JAL
$Immed_{20}$	Rt_6	Ra_6	$41h_8$	JALI

Operation:

Memory Indexed Indirect Form

PC = memory[address + Rn]

Register Indirect with Displacement Form

PC = displacement + Rn

Notes:

The address constant may be extended up to 80 bits with immediate postfix instructions.

JMP – Jump

JMP (abs,Rn)
JMP d(Rn)

Instruction Formats:

$Immed_{20}$	0_6	Ra_6	$40h_8$	JMP
$Immed_{20}$	0_6	Ra_6	$41h_8$	JMPI

Operation:

Memory Indexed Indirect Form

PC = memory[address + Rn]

Register Indirect with Displacement Form

PC = displacement + Rn

Notes:

The address constant may be extended up to 80 bits with immediate postfix instructions.

This instruction is an alternate mnemonic for the JAL / JALI instruction, where the target register is specified as zero.

If Ra is specified as 63 then the current instruction address is used in forming the target address. This allows program counter relative addresses to be formed.

LDB – Load Byte with Sign Extend

LDBX – Load Byte with Sign Extend

LDB Rt, d(Rn)
LDB Rt, d(Ra + Rb * scale)

Description:

This instruction loads an eight bit quantity into a register then sign extends the value to the width of the machine.

Instruction Formats:

$Displacement_{20}$			Rt_6	Ra_6	$80h_8$	LDB Rt,d18(Rn)
$Displacement_{11}$	Sc_3	Rt_6	Rb_6	Ra_6	$A0h_8$	LDB Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

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

Register-Register Form

Rt = sign extend(memory[offset + Ra + Rb * scale])

Notes:

The displacement constant may be extended up to 80. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8
4	16
5	5
6	10

LDBU – Load Byte with Zero Extend

LDBUX – Load Byte with Zero Extend

LDBU Rt, d(Rn)
LDBU Rt, d(Ra + Rb * scale)

Description:

This instruction loads an eight bit quantity into a register then zero extends the value to the width of the machine.

Instruction Formats:

$Displacement_{20}$			Rt_6	Ra_6	$81h_8$	LDBU Rt,d20(Rn)
$Displacement_{11}$	Sc_3	Rt_6	Rb_6	Ra_6	$A1h_8$	LDBU Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

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

Register-Register Form

Rt = sign extend(memory[offset + Ra + Rb * scale])

Notes:

The displacement constant may be extended up to 80. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8
4	16
5	5
6	10

LDD – Load Deci-byte with Sign Extend

LDDX – Load Deci-byte with Sign Extend

LDD Rt, d(Rn)
LDD Rt, d(Ra + Rb * scale)

Description:

This instruction loads an 80 bit value into a register and sign extends it to the register width.

Instruction Formats:

$Displacement_{20}$			Rt_6	Ra_6	$86h_8$	LDD Rt,d20(Rn)
$Displacement_{11}$	Sc_3	Rt_7	Rb_6	Ra_6	$A6h_8$	LDD Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

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

Register-Register Form

Rt = sign extend(memory[offset + Ra + Rb * scale])

Notes:

The displacement constant may be extended up to 80 bits with immediate prefix instructions. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8
4	16
5	5
6	10

LDLI – Load Lower Immediate

LDLI Rn,#i27

Description:

This instruction loads a 27 bit immediate constant into the low order bits of a register and sign extends the result to the width of the register.

Instruction Formats:

$Constant_{26..1}$	Rt_6	$60h_7$	I_0	IMM

LDMI – Load Middle Immediate

LDMI Rn,#i27

Description:

This instruction loads a 27 bit immediate constant into the bits 27 to 53 of a register and sign extends the result to the width of the register. The current low order bits of the register are not affected.

Instruction Formats:

$Constant_{53..28}$	Rt_6	$61h_7$	I_{27}	IMM

LDUI – Load Upper Immediate

LDUI Rn,#i27

Description:

This instruction loads a 27 bit immediate constant into the bits 54 to 79 of a register. The current low order bits of the register are not affected.

Instruction Formats:

$Constant_{79..55}$	Rt_6	$62h_7$	I_{54}	IMM

LDP – Load Pentabyte with Sign Extend

LDPX – Load Pentabyte with Sign Extend

LDP Rt, d(Rn)
LDP Rt, d(Ra + Rb * scale)

Description:

This instruction loads a 40 bit value into a register and sign extends it to the register width.

Instruction Formats:

$Displacement_{20}$			Rt_6	Ra_6	$84h_8$	LDT Rt,d20(Rn)
$Displacement_{11}$	Sc_3	Rt_7	Rb_6	Ra_6	$A4h_8$	LDT Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

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

Register-Register Form

Rt = sign extend(memory[offset + Ra + Rb * scale])

Notes:

The displacement constant may be extended up to 80 bits with immediate prefix instructions. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8
4	16
5	5
6	10

LDPU – Load Pentabyte with Zero Extend

LDPUX – Load Pentabyte with Zero Extend

LDPU Rt, d(Rn)
LDPU Rt, d(Ra + Rb * scale)

Description:

This instruction loads a 40 bit value into a register and zero extends it to the register width.

Instruction Formats:

$Displacement_{20}$			Rt_6	Ra_6	$85h_8$	LDTU Rt,d19(Rn)
$Displacement_{11}$	Sc_3	Rt_7	Rb_6	Ra_6	$A5h_8$	LDTU Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

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

Register-Register Form

Rt = sign extend(memory[offset + Ra + Rb * scale])

Notes:

The displacement constant may be extended up to 80 bits with immediate prefix instructions. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8
4	16
5	5
6	10

LDDAR – Load Deci-byte with Sign Extend and Reserve

LDDARX – Load Deci-byte with Sign Extend and Reserve

LDDAR Rt, d(Rn)
LDDAR Rt, d(Ra + Rb * scale)

Description:

This instruction loads an 80 bit value into a register and sign extends it to the register width. In addition the memory reservation signal is activated.

Instruction Formats:

$Displacement_{20}$			Rt_6	Ra_6	$8Eh_8$	LDDAR Rt,d20(Rn)
$Displacement_{11}$	Sc_3	Rt_7	Rb_6	Ra_6	AEh_8	LDDAR Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

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

Register-Register Form

Rt = sign extend(memory[offset + Ra + Rb * scale])

Notes:

The displacement constant may be extended up to 80 bits with immediate prefix instructions. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8
4	16
5	5
6	10

LDW – Load Wyde with Sign Extend

LDWX – Load Wyde with Sign Extend

LDW Rt, d(Rn)
LDW Rt, d(Ra + Rb * scale)

Description:

This instruction loads a sixteen bit quantity into a register then sign extends it to the width of the register.

Instruction Formats:

$Displacement_{20}$			Rt_6	Ra_6	$82h_8$	LDW Rt,d20(Rn)
$Displacement_{11}$	Sc_3	Rt_7	Rb_6	Ra_6	$A2h_8$	LDW Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

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

Register-Register Form

Rt = sign extend(memory[offset + Ra + Rb * scale])

Notes:

The displacement constant may be extended up to 80. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8
4	16
5	5
6	10

LDWU – Load Wyde with Zero Extend

LDWUX – Load Wyde with Zero Extend

LDWU Rt, d(Rn)
LDWU Rt, d(Ra + Rb * scale)

Description:

This instruction loads a sixteen bit quantity into a register then zero extends it to the width of the register.

Instruction Formats:

$Displacement_{20}$			Rt_6	Ra_6	$83h_8$	LDWU Rt,d20(Rn)
$Displacement_{11}$	Sc_3	Rt_7	Rb_6	Ra_6	$A3h_8$	LDWU Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

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

Register-Register Form

Rt = sign extend(memory[offset + Ra + Rb * scale])

Notes:

The displacement constant may be extended up to 80. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8
4	16
5	5
6	10

LEA – Load Effective Address

LEA Rt,d(Ra)
LEA Rt, d(Ra + Rb * scale)

Description

This instruction loads the effective address of a memory operand into a register.

Instruction Formats:

$Displacement_{20}$			Rt_6	Ra_6	$26h_8$	LEA Rt,d19(Rn)
$Displacement_{11}$	Sc_3	Rt_6	Rb_6	Ra_6	$27h_8$	LEA Rt,d(Ra+Rb*sc)

Operation:

Indexed Form

Rt = address of ($memory_{32}$[offset + Ra + Rb * scale])

Notes:

This instruction loads the target register with the address of the memory determined by the indexing operation.

The displacement constant may be extended up to 80 bits with immediate postfix instructions.

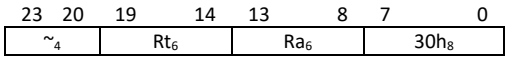
Sc_3 Code	Multiply By
0	1
1	2
2	4
3	8
4	16

MOV – Move Register to Register

Description:

The contents of the source register Ra are moved to the target register Rt.

Instruction Format:



Clock Cycles: 1

Exceptions: none

MUL - Multiplication

MUL Rt, Ra, #i20
MUL Rt, Ra, Rb

Description:

This instruction multiplies two values using signed arithmetic. The first value is in a register, the second value may be either in a register or an immediate constant. The signed result is placed in the target register.

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
$10h_8$	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	MUL Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	10_8	MUL Rt,Ra,#imm

Operation:

Register Immediate Form

Rt = Ra * $immediate_{20}$

Register-Register Form

Rt = Ra * Rb

Clock Cycles: 8

Exceptions:

Multiplier overflow (if enabled in the EXROUT CSR). By default the multiplier overflow exception is not enabled.

Multiplier overflow occurs if the upper bits of the product (bits 159 to 80) are not equal to the most significant bit (bit 79).

Notes*:*

The immediate constant may be extended up to 64 bits.

MULU – Unsigned Multiplication

MULU Rt, Ra, #i20
MULU Rt, Ra, Rb

Description:

This instruction multiplies two values using unsigned arithmetic. The first value is in a register, the second value may be either in a register or an immediate constant. The unsigned result is placed in the target register.

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
$11h_8$	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	MUL Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	11_8	MUL Rt,Ra,#imm

Operation:

Register Immediate Form

Rt = Ra * $immediate_{20}$

Register-Register Form

Rt = Ra * Rb

Clock Cycles: 8

Exceptions:

Multiplier overflow (if enabled in the EXROUT CSR). By default the multiplier overflow exception is not enabled.

Multiplier overflow occurs if the upper bits of the product (bits 159 to 80) are not equal zero.

Notes*:*

The immediate constant may be extended up to 64 bits.

MUL - Multiplication

MUX – Bitwise Multiplex

MUX Rt, Ra, Rb, Rc

Desription:

This instruction copies bits to the target register from either register Rb or Rc depending on the setting of the corresponding bit in register Ra. If a bit in Ra is set then the corresponding bit from Rb is copied to Rt, otherwise the corresponding bit from Rc is copied to Rt.

Instruction Formats:

39 32	31 26	25 20	19 14	13 8	7 0	
$5Fh_8$	Rt_6	Rc_6	Rb_6	Ra_6	02_8	MUX Rt,Ra,Rb,Rc

Operation:

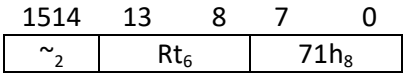
Register-Register Form

for bit = 0 to 79

Rt[bit] = Ra[bit] ? Rb[bit] : Rc[bit]

POP – Pop Register from Stack

Instruction Format:



Example:

POP r1

PUSH – Push Register on Stack

Instruction Format:

1514	13 8	7 0
$\sim_2$	Ra_6	$70h_8$

Example:

```
PUSH    r1
```

ROL – Rotate Left Register

Description:

Rotates the register to the left by an amount specified in either a second register or a constant in the instruction and place the result in the target register. Bit 79 is shifted into bit 0.

Instruction Format:

39 32	31 26		25 20	19 14	13 8	7 0	
$34h_8$	$\sim_6$		Rt_6	Rb_6	Ra_6	02_8	ROL Rt,Ra,Rb
$44h_8$	$\sim_5$	I_1	Rt_6	$Immed_6$	Ra_6	02_8	ROL Rt,Ra,#i7

Operation:

Rt = Ra << Rb

Clock Cycles: 1

Exceptions: none

Notes:

ROR – Rotate Right Register

Description:

Rotates the register to the right by an amount specified in either a second register or a constant in the instruction and place the result in the target register. Bit 0 is shifted into bit 79.

Instruction Format:

39 32	31 26		25 20	19 14	13 8	7 0	
$35h_8$	$\sim_6$		Rt_6	Rb_6	Ra_6	02_8	ROR Rt,Ra,Rb
$45h_8$	$\sim_5$	I_1	Rt_6	$Immed_6$	Ra_6	02_8	ROR Rt,Ra,#i7

Operation:

Rt = Ra >> Rb

Clock Cycles: 1

Exceptions: none

Notes:

RET – Return From Subroutine

RET #i16

Description

This instruction performs a return from subroutine operation. The program counter is loaded from the stack, then the stack pointer incremented by the amount specified in the instruction.

Instruction Formats:

$Immediate_{16}$	EFh_8	RET #i16

Operation:

PC = memory[SP]
SP = SP + Immediate

Notes:

The stack pointer may be adjusted in order to remove parameters from the stack. In assembler code if an immediate value is specified it must include ten bytes for popping the return address. By default the immediate value is set to ten.

REX – Redirect Exception

Description:

This instruction redirects an exception from an operating level to a lower operating level and privilege level. If the target operating level is hypervisor then the hypervisor privilege level (1) is set. If the target operating level is supervisor then one of the supervisor privilege levels must be chosen (2 to 7). This instruction if successful jumps to the target exception handler and does not return. If this instruction fails execution will continue with the next instruction.

This instruction may fail if exceptions are not enabled at the target level.

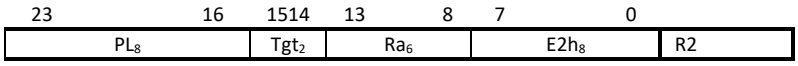
When redirecting to the supervisor level, the target privilege level is set to the bitwise 'or' of an immediate constant specified in the instruction and register Ra. One of these two values should be zero. The result should be a value in the range 2 to 7.

The location of the target exception handler is found in the trap vector register for that operating level (stvec, htvec).

The cause (cause) and bad address (badaddr) registers of the originating level are copied to the corresponding registers in the target level.

The REX instruction automatically enables exceptions for operating levels higher (numerically lower than) than the target level.

Instruction Format:



Tgt_2	
0	not used
1	redirect to hypervisor level
2	redirect to supervisor level
3	not used

Clock Cycles: 3

Example:

```
REX super,2,r0   ; redirect to supervisor handler, privilege level two
; If the redirection failed, exceptions were likely disabled at the target level.
; Continue processing so the target level may complete it's operation.
IRET             ; redirection failed (exceptions disabled ?)
```

Notes:

Since all exceptions are initially handled at the machine level the machine level handler must check for disabled lower level exceptions.

RTI – Return from Interrupt

Description:

This instruction returns the processor from an exception handler back to the exceptioned code. The program counter and machine status are unstacked from internal registers. This instruction may also clear a semaphore register bit. Semaphore register bit #0 (the reservation status bit) is always cleared by this instruction.

Instruction Format:

15	14 8	7 0
~	$Sema_7$	$E4h_8$

SHL – Shift Left Register

Description:

Shift register to the left by an amount specified in either a second register or a constant in the instruction and place the result in the target register. A Zero is shifted into bit 0.

Instruction Format:

39 32	31 26		25 20	19 14	13 8	7 0	
$30h_8$	$\sim_6$		Rt_6	Rb_6	Ra_6	02_8	SHL Rt,Ra,Rb
$40h_8$	$\sim_5$	I_1	Rt_6	$Immed_6$	Ra_6	02_8	SHL Rt,Ra,#i7

Operation:

Rt = Ra << Rb

Clock Cycles: 1

Exceptions: none

STB – Store Byte

STBX – Store Byte

STB Rt, d(Rn)
STB Rt, d(Ra + Rb * scale)

Description

This instruction stores a byte from a register to memory using either register indirect with displacement or scaled indexed addressing.

Instruction Formats:

$Displacement_{20}$			Rs_6	Ra_6	90_8h	STB Rs,d15(Rn)
$Displacement_{11}$	Sc_3	Rb_6	Rs_6	Ra_6	$B0_8h$	STB Rs,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

memory[displacement + Ra] = Rs

Register-Register Form

memory[offset + Ra + Rb * scale] = Rs

Notes:

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

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8
4	16
5	5
6	10

STD – Store Deci-byte

STDX – Store Deci-byte

STD Rt, d(Rn)
STD Rt, d(Ra + Rb * scale)

Description

This instruction stores a deci-byte (80 bits) from a register to memory using either register indirect with displacement or scaled indexed addressing.

Instruction Formats:

$Displacement_{20}$			Rs_6	Ra_6	93_8h	STD Rs,d15(Rn)
$Displacement_{11}$	Sc_3	Rb_6	Rs_6	Ra_6	$B3_8h$	STD Rs,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

memory[displacement + Ra] = Rs

Register-Register Form

memory[offset + Ra + Rb * scale] = Rs

Notes:

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

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8
4	16
5	5
6	10

STDCR – Store Deci-byte and Clear Reservation

STDCR Rt, d(Rn)

Description

This instruction conditionally stores a deci-byte (80 bits) from a register to memory using register indirect with displacement addressing. Memory is updated only if a reservation exists on the address. If memory was updated the least significant bit of the semaphore CSR is set, otherwise the bit is cleared. This instruction sets the cr_o signal during execution. The memory system must be capable of aborting the store if there is no reservation present. The rb_i status input bit is latched by this instruction.

Instruction Formats:

$Displacement_{20}$			Rs_6	Ra_6	95_8h	STDCR Rs,d20(Rn)
$Displacement_{11}$	Sc_3	Rb_6	Rs_6	Ra_6	$B5_8h$	STDCR Rs,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

```
if (address reserved)
        memory[displacement + Ra] = Rs
        clear address reservation
```

Register-Register Form

```
If (address reserved)
        memory[offset + Ra + Rb * scale] = Rs
        clear address reservation
```

Notes:

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

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8
4	16

STOP – Stop Processor

STOP

This instruction stops the processor placing it in low power mode by stopping the processor clock. The clock rate register is loaded with zero. The processor may begin processing again once a non-maskable interrupt occurs or a reset occurs. The processor may be slowed down without stopping the clock by adjusting the value in the clock rate register.

STP – Store Penta

STPX – Store Penta

STP Rt, d(Rn)
STP Rt, d(Ra + Rb * scale)

Description

This instruction stores a penta-byte (40 bits) from a register to memory using either register indirect with displacement or scaled indexed addressing.

Instruction Formats:

$Displacement_{20}$			Rs_6	Ra_6	92_8h	STP Rs,d15(Rn)
$Displacement_{11}$	Sc_3	Rb_6	Rs_6	Ra_6	$B2_8h$	STP Rs,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

memory[displacement + Ra] = Rs

Register-Register Form

memory[offset + Ra + Rb * scale] = Rs

Notes:

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

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8
4	16
5	5
6	10

STW – Store Wyde

STWX – Store Wyde

STW Rt, d(Rn)
STW Rt, d(Ra + Rb * scale)

Description

This instruction stores a wyde from a register to memory using either register indirect with displacement or scaled indexed addressing.

Instruction Formats:

$Displacement_{20}$			Rs_6	Ra_6	91_8h	STW Rs,d15(Rn)
$Displacement_{11}$	Sc_3	Rb_6	Rs_6	Ra_6	$B1_8h$	STW Rs,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

memory[displacement + Ra] = Rs

Register-Register Form

memory[offset + Ra + Rb * scale] = Rs

Notes:

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

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8
4	16
5	5
6	10

SXB – Sign Extend Byte

SXB Rt, Ra

Description

This instruction sign extends a byte in a register to the width of the register. This instruction is an alternate mnemonic for the BFEXT instruction.

Instruction Formats:

39 36	35	34 28	27	26 20	19 14	13 8	7 0
5_4	$\sim_1$	7_7	~	0_7	Rt_6	Ra_6	$13h_8$

Operation:

Register Form

Rt = sign extend (Ra)

Notes:

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

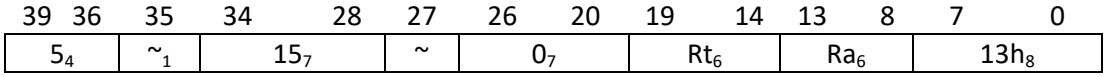
SXW – Sign Extend Wyde

SXW Rt, Ra

Description

This instruction sign extends a wyde in a register to the width of the register. This instruction is an alternate mnemonic for the BFEXT instruction.

Instruction Formats:



Operation:

Register Form

Rt = sign extend (Ra)

Notes:

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

SXT – Sign Extend Tetra

SXT Rt, Ra

Description

This instruction sign extends a tetra in a register to the width of the register. This instruction is an alternate mnemonic for the BFEXT instruction.

Instruction Formats:

39 36	35	34 28	27	26 20	19 14	13 8	7 0
5_4	$\sim_1$	31_7	~	0_7	Rt_6	Ra_6	$13h_8$

Operation:

Register Form

Rt = sign extend (Ra)

Notes:

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

TGT – Jump / Call Target

TGT

Description:

This instruction identifies the location of a jump or call target. If the target exception is enabled the core will generate an exception if the destination of a call or jump operation doesn't contain the TGT instruction. The TGT instruction is treated as a NOP.

Instruction Formats:

53_8	TGT

Operation:

Exceptions: target exception

Notes:

WAIT – Wait For Interrupt

WAIT

Instruction Formats:

$E3_8$	WAI

Operation:

```
if (no interrupt)
   PC = PC
else
   PC = PC + 1
```

Notes:

This instruction waits for an interrupt to occur before proceeding..

XOR – Bitwise Exclusive 'Or'

XOR Rt, Ra, #i20
XOR Rt, Ra, Rb

Instruction Formats:

39 33	31 26	25 20	19 14	13 8	7 0	
$0Ah_8$	$\sim_6$	Rt_6	Rb_6	Ra_6	02_8	XOR Rt,Ra,Rb
$Immediate_{20}$			Rt_6	Ra_6	$0A_8$	XOR Rt,Ra,#imm

Operation:

Register Immediate Form

Rt = Ra ^ immediate

Register-Register Form

Rt = Ra ^ Rb

Notes*:*

The immediate constant may be extended up to 80 bits.

Floating Point Instruction Set

FABS – Absolute Value

Description:

This instruction takes the absolute value of a double precision floating point number contained in a general purpose register. The sign bit of the number is cleared. The precision of the number is not affected and the number is not rounded.

Instruction Format:

39 38	37 35	34 32	31 26	25 20	19 14	13 8	7 0
$\sim_2$	$Prec_2$	Rm_3	Rt_6	15_6	$\sim_6$	Ra_6	$F1h_8$

Clock Cycles: 3

Execution Units: All Floating Point

Operation:

Rt = Ra

FADD – Floating point addition

Description:

Add two floating point numbers in registers Ra and Rb and place the result into target register Rt.

Instruction Format:

39 38	37 35	34 32	31 26	25 20	19 14	13 8	7 0
$\sim_2$	$Prec_3$	Rm_3	Rt_6	04_6	Rb_6	Ra_6	$F1h_8$

Clock Cycles: 10

Execution Units: All Floating Point

FCMP - Float Compare

Description:

The register compare instruction compares two registers as floating point values and sets the flags in the target integer register as a result.

Instruction Format:

39 38	37 35	34 32	31 26	25 20	19 14	13 8	7 0
$\sim_2$	$Prec_3$	Rm_3	Rt_6	06_6	Rb_6	Ra_6	$F1h_8$

Clock Cycles: 3

Execution Units: All ALU's

Operation:

```
if Ra < Rb
        Rt[1]= true
else
        Rt[1] = false
if mag Ra < mag Rb
        Rt[2] = true
else
        Rt[2] = false
if Ra = Rb
        Rt[0] = true
else
        Rt[0] = false
if Ra <= Rb
        Rt[3] = true
else
        Rt[3] = false
if unordered
        Rt[4] = true
else
        Rt[4] = false
```

FCX – Clear Floating Point Exceptions

Description:

This instruction clears floating point exceptions. The Exceptions to clear are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format:

39 38	37 35	34 32	31 26	25 20	19 14	13 8	7 0
$\sim_2$	$Prec_2$	Rm_3	$\sim_6$	21_6	$Immed_6$	Ra_6	$F1h_8$

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	global invalid operation clears the following: - division of infinities - zero divided by zero - subtraction of infinities - infinity times zero - NaN comparison - division by zero
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	summary exception

FDIV – Floating point division

Description:

Divide two floating point numbers in registers Ra and Rb and place the result into target register Rt.

Instruction Format:

39 38	37 35	34 32	31 26	25 20	19 14	13 8	7 0
$\sim_2$	$Prec_3$	Rm_3	Rt_6	09_6	Rb_6	Ra_6	$F1h_8$

Clock Cycles: 140

Execution Units: All Floating Point

FDX – Disable Floating Point Exceptions

Description:

This instruction disables floating point exceptions. The Exceptions disabled are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format:

39 38	37 35	34 32	31 26	25 20	19 14	13 8	7 0
$\sim_2$	$Prec_2$	Rm_3	$\sim_6$	22_6	$Immed_6$	Ra_6	$F1h_8$

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Disabled
0	invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

FEX – Enable Floating Point Exceptions

Description:

This instruction enables floating point exceptions. The Exceptions enabled are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format:

39 38	37 35	34 32	31 26	25 20	19 14	13 8	7 0
$\sim_2$	$Prec_2$	Rm_3	$\sim_6$	23_6	$Immed_6$	Ra_6	$F1h_8$

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

FMUL – Floating point multiplication

Description:

Multiply two floating point numbers in registers Ra and Rb and place the result into target register Rt.

Instruction Format:

39 38	37 35	34 32	31 26	25 20	19 14	13 8	7 0
$\sim_2$	$Prec_3$	Rm_3	Rt_6	08_6	Rb_6	Ra_6	$F1h_8$

Clock Cycles: 5

Execution Units: All Floating Point

FSUB – Floating point subtraction

Description:

Subtract two floating point numbers in registers Ra and Rb and place the result into target register Rt.

Instruction Format:

39 38	37 35	34 32	31 26	25 20	19 14	13 8	7 0
$\sim_2$	$Prec_3$	Rm_3	Rt_6	05_6	Rb_6	Ra_6	$F1h_8$

Clock Cycles: 10

Execution Units: All Floating Point

FTX – Trigger Floating Point Exceptions

Description:

This instruction triggers floating point exceptions. The Exceptions to trigger are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format:

39 38	37 35	34 32	31 26	25 20	19 14	13 6	5 0
$\sim_2$	$Prec_3$	Rm_3	$\sim_6$	20_6	Imm_6	Ra_6	$F1h_8$

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	global invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

LFx – Load Float

Description:

Loads a word of data from memory addressed as the sum of a register (Ra) and an immediate value specified in the instruction. The address must be word aligned for proper operation.

Instruction Format:

Opcode Maps

Major Opcodes $IR_{[7:0]}$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	MFLT	{R1}	{R2}	{R3}	ADD #	SUB #	CMP #		AND #	OR #	XOR #	CHK #		{SYS}	_16ADD #	CSR
1x	MUL #	MULU #	MULSU #	{BITFIELD}	ADDO #	SUBO #	CMPU #		DIV #	DIVU #	DIVSU #	REM #	REMU #	REMSU #	_8ADD #	_10ADD #
2x	MULH #	MULUH #	MULSUH #		SADD #	SSUB #	LEA	LEAX	DIVR #						_4ADD #	
3x	MOV						FBEQ	FBNE	FBLT	FBGE	FBLE	FBGT	FBOR	FBUN	_2ADD #	
4x	JMP	JMP ()	JMP.T ()				BEQ	BNE	BLT	BGE	BLE	BGT	BLTU	BGEU	BLEU	BGTU
5x	CALL	CALL ()	CALL.T ()	TGT	BBC #	BBS #	BEQ #	BNE #	BLT #	BGE #	BLE #	BGT #	BLTU #	BGEU #	BLEU #	BGTU #
6x	CSZ #	CSNZ #	CSN #	CSNN #	CSP #	CSNP #	CSOD #	CSEV #	ZSZ #	ZSNZ #	ZSN #	ZSNN #	ZNP #	ZSNP #	ZSOD #	ZSEV #
7x	PUSH	POP					SEQ #	SNE #	SLT #	SGE #	SLE #	SGT #	SLTU #	SGEU #	SLEU #	SGTU #
8x	LDB	LDBU	LDW	LDWU	LDP	LDPU	LDD	LDT	LDTU						LDDAR	
9x	STB	STW	STP	STD	STT	STDCR	INC						PEA			
Ax	LDBX	LDBUX	LDWX	LDWUX	LDPX	LDPUX	LDDX	LDTX	LDTUX	LV	LVWS	LVX	LCL	CAS	LDDARX	
Bx	STBX	STWX	STPX	STDX	STTX	STDCRX	INCX			SV	SVWS	SVX	PEAX		CACHE	
Cx	IMMEDIATE EXTENSION															
Dx																
Ex		BRK	REX	WAI	RTI	MEMSB	MEMDB	SYNC	CLI	SEI	NOP	PUSHF	POPF			RET
Fx		{F1}	{F2}						MARK1	MARK2	MARK3	MARK4				MFLT

Major Functs $IR_{[39:32]}$ $IR_{[7:0]}$=02

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x					ADD	SUB	CMP		AND	OR	XOR	CHK		{SYS}	_16ADD	
1x	MUL	MULU	MULSU		ADDU	SUBU	CMPU		DIV	DIVU	DIVSU	REM	REMU	REMSU	_8ADD	
2x	MULH	MULUH	MULSUH	LEAX	SADD	SSUB			DIVR						_4ADD	
3x	SHL	SHRU	ASL	ASR	ROL	ROR									_2ADD	ABS
4x	SHL #	SHRU #	ASL #	ASR #	ROL #	ROR #			NAND	NOR	XNOR	ANDN	ORN		MIN	MAX
5x																MUX
6x	CSZ	CSNZ	CSN	CSNN	CSP	CSNP	CSOD	CSEV	ZSZ	ZSNZ	ZSN	ZSNN	ZNP	ZSNP	ZSOD	ZSEV

7x							SEQ	SNE	SLT	SGE	SLE	SGT	SLTU	SGEU	SLEU	SGTU
8x																RET
9x																
Ax																
Bx																
Cx																
Dx																
Ex																
Fx																

Float Ops $IR_{[39:32]}$ $IR_{[7:0]}$=F1h

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x					FADD	FSUB	CMP		FMUL	FDIV		FCHK				
1x						FABS										
2x	FTX	FCX	FDX	FEX												
3x																

Bitfield Ops $IR_{[39:36]}$ $IR_{[7:0]}$=13h

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BFSET	BFCLR	BFCHG	BFINS	BFINSI	BFEXT	BFEXTU									

WISHBONE Compatibility Datasheet

The DSD9 core may be directly interfaced to a WISHBONE compatible bus.

WISHBONE Datasheet WISHBONE SoC Architecture Specification, Revision B.3		
Description:	Specifications:	
General Description:	Central processing unit (CPU core)	
Supported Cycles:	MASTER, READ / WRITE MASTER, BLOCK READ / WRITE	
Data port, size: Data port, granularity: Data port, maximum operand size: Data transfer ordering: Data transfer sequencing	128 bit 8 bit 128 bit Little Endian any (undefined)	
Clock frequency constraints:		
Supported signal list and cross reference to equivalent WISHBONE signals	Signal Name: ack_i adr_o(31:0) clk_i dat_i(128:0) dat_o(128:0) cyc_o stb_o wr_o sel_o(15:0)	WISHBONE Equiv. ACK_I ADR_O() CLK_I DAT_I() DAT_O() CYC_O STB_O WE_O SEL_O
Special Requirements:		