DSD7

Definitions

Word: A word is a 32 bit quantity

Half-Word: A half-word is a 16 bit quantity.

General Architecture

Instruction Pipeline

DSD7 has a short three stage overlapped pipeline that allows many instructions to execute in a single clock cycle. Loads and stores stall the pipeline until the memory operation is complete.

Caching

The core has a four-way set-associative instruction cache. At reset the cache is loaded with the contents of the last block of memory ($FFFFFE00 to $FFFFFFFF). The core outputs an address sequence on the bus in order to load the cache during reset.

Register File

The core has a 32 entry, 32 bit general purpose register file. r0 always reads as a zero.

Data / Instruction Granularity

Data and instructions both use a minimum parcel size of 16 bits. Addresses refer to 16 bit quantities.

Bus Interface

The core uses a simple synchronous bus to interface to the reset of the system. A bus transaction is assumed automatically to be completed at the end of a clock cycle unless the ready line (rdy_i) is brought low. Bus transactions are single cycle. The presence of valid data address on the bus is indicated by the vda_o signal. The presence of a valid instruction address is indicated by the vpa_o signal.

Programming Model

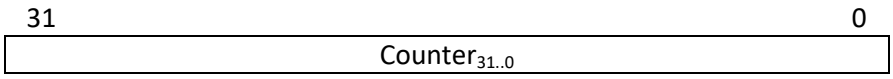
General Purpose Registers

DSD7 has a set of 32, 32 bit registers (r0 to r31) for general purpose use. The r0 register is defined to read as zero. Register r31 is reserved for the stack pointer. The stack pointer although dedicated for stack access may also be used as a general purpose register. Which register is the stack pointer is actually configurable by the config register. The default is r31.

Register	Description / Suggested Usage	Saver
r0	always reads as zero	
r1-r2	return values	caller
r3-r10	temporaries	caller
r11-r17	register variables	callee
r18-r23	function arguments	caller
r24	type number / function argument	caller
r25	class pointer / function argument	caller
r26	thread pointer	callee
r27	global pointer	
r28	exception link register	caller
r29	return address / link register	caller
r30	base / frame pointer	callee
r31	stack pointer	callee

Program Counter

The program counter is a half-word pointer. Instruction parcels are 16 bits wide. Instructions may be made up of multiple parcels. The program counter is 16 bit aligned and 16 bit oriented. Data is also 16 bit oriented referred to as half-word.



Interrupt State Stack

The interrupt state stack stores information required to restore the prior state when an interrupt occurs. This stack stores the program counter, the three general purpose registers (r1, r2, and r29), and status bits (interrupt mask). The stack is only 16 levels deep meaning interrupts can't nest more than 16 levels. This is actually not a limitation as interrupt nesting is rarely used. It is possible to modify the top element of the stack using the IPOP, IPUSH instructions coupled with the itos CSR.

R29 was chosen as state to be automatically stored and restored as it is the link register and therefore would allow calling subroutines one level deep in the interrupt service routine. It's convenient to have state save and restore code implemented as subroutines.

Control and Status Registers

One of the things the author liked about the RISCV ISA is the support for CSR's. There could potentially be up to four sets of CSR's depending on the available core operating levels. Currently only the machine level is supported. The CSR set selected is chosen from the upper two bits of the CSR register number which should be zero for the machine level. Since the register number is a 14 bit field there could be up to 4096 CSR's for each operating level.

$Regno_{11}$	Width	OL	Name	Description	
0x000	32			reserved – reads as zero	
0x001	32	M	HARTID	hardware thread id	
0x002	32	M	TICK	clock cycle counter	
0x003	32		PCR	paging control register	
Exception Processing					
0x004	32	M	VBA	trap vector table base address	
0x006	32	M	CAUSE	exception cause register	
0x007	32	M	BADADDR	bad address register	
0x009	32	M	SCRATCH	scratch register	
0x00C	32	M	SEMA	semaphores	
0x00D	32	M	SP	alternate stack pointer	
0x00E	32	M	SBL	stack bound – lower	
0x00F	32	M	SBU	stack bound - upper	
0x010	32		TCBP	tcb pointer/task register	
0x011	32		CISC	compressed instruction set control	
0x012	32	M	STATUS	status register	
0xFF0	32	M	CONFIG	configuration register	
0xFFE	32	M	CAP	capabilities	
0xFFF	32	M	IMPID	vendor ID and version number	

Hardware Thread Identifier (CSR #001h)

This is an externally supplied identifier that identifies which hardware thread the core represents.

Tick (CSR #002h)

This read-only register contains a count of the number of clock cycles since the core was reset.

PCR (CSR #003h)

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

VBA (CSR #004h)

This register holds the address of the interrupt vector table. On reset the register contains the value $FFFFFFE0. Room should be reserved in the table for future operating modes. Interrupts

will vector to $FFFFFFE0,$FFFFFFE4,$FFFFFFE8, and $FFFFFFEC for operating level zero, one, two, and three respectively.

Cause (CSR #006h)

This register contains a code indicating the cause of an exception. The exception cause register is loaded by the INT instruction.

BADADDR (CSR #007)

This register contains the effective address for a load / store operation that caused a memory management exception. Note that the address of the instruction causing the exception is available in the EPC register (ITOS0).

Scratch (CSR #009h)

This register is available for scratchpad use. It is typically swapped with a GPR during exception processing.

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 SWC instruction was successful. The least significant bit is also cleared automatically when an interrupt (INT) or interrupt return (IRET) instruction is executed. Any one of the remaining bits may also be cleared by an IRET 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,#$41,#E_Busy             // store busy status in ER1 to be returned in r1
                    iret                                // leave system busy status bit set

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

SP (CSR #00Dh)

This register is reserved for access to the alternate stack pointers for different operating levels.

SBL (CSR #00Eh)

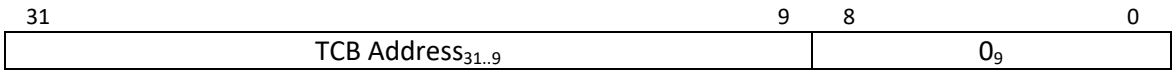
The SBL register contains the address representing the lower bound of the stack. If an address is formed using one of the stack indexing registers (stack pointer x31 or base pointer x30) is lower than the SBL a stack fault occurs. This represents a stack overflow condition.

SBU (CSR #00Fh)

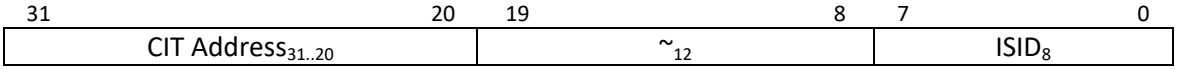
The SBU register contains the address representing the upper bound of the stack. If an address is formed using one of the stack indexing registers (stack pointer x31 or base pointer x30) is higher than the SBU a stack fault occurs. This represents a stack underflow condition.

TCB Pointer (Task) Register (CSR #010h)

This register contains a pointer to the task control block for the active task. The task control block address is 512 character aligned. This register is typically swapped with a GPR in order to save or restore task state in the TCB.



Compressed Instruction Set Control (CSR #011h)



This register controls where in memory the CIT appears (CITA) and which compressed instruction set is active (ISID). The default value of the register - $FFE00000 selects instruction set zero and sets the CITA address range to $FFE00000 to $FFEFFFFF.

Instruction Space Identifier (ISID)

The instruction space identifier is an eight bit register used to determine which set of compressed instructions are to be used by the currently running program. The processor supports multiple sets of compressed instructions. It may be desirable to share the compressed instruction set between several programs as there is limited storage space for compressed instructions. The instruction space identifier forms the upper address bits for the table lookup. The lower address bits of the table are determined by the instruction code.

Compressed Instruction Table Address (CITA)

This register controls where in the memory map the compressed instruction set table appears. By default the value is $FFE00000. Up to 1MB is reserved for this area. There is enough room for 256k compressed instructions. Regular store operations from non-user operating levels may be used to update the table in the chosen address range. However the table may not be read. The core will perform an external write cycle when it updates the table. There should not be another memory at the same location as the compressed instruction table.

ITOS CSR's

The ITOS CSR's act as the top of the interrupt stack. In order to allow nested interrupts the current top of stack must be pushed with the IPUSH instruction before interrupts are enabled in the interrupt subroutine.

ITOS0/EPC CSR #040h

This register contains the return address for the exceptioned instruction.

ITOS1/ER1 CSR #041h

This register contains the value of r1 at the point of exception.

ITOS2/ER2 CSR #042h

This register contains the value of r2 at the point of exception.

ITOS3 CSR #043h

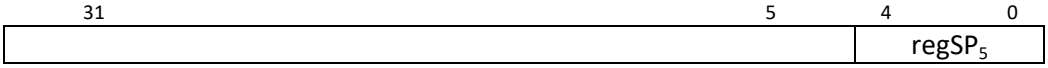
This register contains the value of r29 at the point of exception.

ITOS4 CSR #044h

This register contains the cpu status bits. The least significant bit is the interrupt mask.

CONFIG CSR #FF0h

This register contains information controlling the configuration of the core.



$regSP_5$ This five bit field determines which processor register is used as the stack pointer. On reset it defaults to r31. r0, r1, r2, and r29 should not be used as the stack pointer.

CAP CSR #FFEh

This read-only register contains bits indicating core capabilities. The core may not implement all instructions in hardware in which case they must be emulated with software. There is a single bit for each optional core capability.

Format:

Data Addressability

Data addressability is the same as instruction addressability. All data is addressed as 16 bit half-words. The minimum size parcel of data that can be handled directly is 16 bits. Access for 16 bit data was allowed because instructions may be only 16 bits in size and the author feels it's best to keep the addressability of both code and data the same.

Exceptions

External Interrupts

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

Effect on Machine Status

The operating level is always switched to the machine level on exception. It's up to the machine level code to redirect the exception to a lower operating level when desired. Further exceptions at the machine level are disabled automatically. Machine level 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, r1, r2, r29 and status bits are pushed onto an internal stack when an exception occurs. This stack is only sixteen 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 vector base address (VBA) register. This register is set to $FFFFFF00 on reset. Note that the reset vector is fixed and cannot be relocated.

Vector Address			
0xFF…FFFF00	Exception from user level		
0xFF…FFFF20	Exception from supervisor level		
0xFF…FFFF40	Exception from hypervisor level		
0xFF…FFFF60	Exception from machine level		
0xFF…FFFF80	Non-maskable interrupt		
0xFF…FFFFA0	Reset		

Reset

On a reset the core vectors to address $FFFFFFA0.

Exception Cause Codes

The following table outlines the cause code for a given purpose. These codes are specific to DSD7. 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	
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	

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. The MMU divides memory into 128kB pages (64k half-words). Processor address bits 16 to 24 are used as a nine bit index into a mapping table to find the physical page. The MMU remaps the nine address bits

into a twelve bit value used as address bits 16 to 27 when accessing a physical address. The lower sixteen bits of the address pass through the MMU unchanged. Also passing through the MMU unchanged are address bits 28 to 31. It is assumed that in the system where the MMU would be relevant, that some or all of the high order bits of an address would be left unconnected. The maximum amount of memory that may be mapped in the MMU is 64MiB per map out of a pool of 512MiB. Addresses with the most significant bit set (bit 31) 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 32 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 4 – 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 single cycle.

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 4.

Access Key

The MMU mapping tables are present at I/O address $FFDC4000 to $FFDC41FF. 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 12.

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. Address bits 28 to 31 pass through the MMU unaltered as well.

Mapping Table Layout

	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	
000				WP	PA27	PA26	PA25	PA24	PA23	PA22	PA21	PA20	PA19	PA18	PA17	PA16	
001				WP	PA27	PA26	PA25	PA24	PA23	PA22	PA21	PA20	PA19	PA18	PA17	PA16	
002				WP	PA27	PA26	PA25	PA24	PA23	PA22	PA21	PA20	PA19	PA18	PA17	PA16	
	…																
1FF				WP	PA27	PA26	PA25	PA24	PA23	PA22	PA21	PA20	PA19	PA18	PA17	PA16	

WP = write protect

PAxx = Physical address bit

Paging Control Register Layout

31	30 13	12 8	7 5	4 0
PE	~18	AKey5	~	OKey5

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

AKey = Access Key

OKey = Operate Key

Latency

The address map operation when enabled has a single cycle of latency. The ready line to the core is brought low for a cycle by the MMU. The MMU delays the (wr, vda, and vpa) control signals to memory.

Detailed Instruction Set Description

ADD – Add Register to Register

Description:

Add two registers and place the result in the target register.

Instruction Format:

w0	$04h_6$	$\sim_5$	Rt_5	Rb_5	Ra_5	$0Ch_6$	RR

Operation:

Rt = Ra + Rb

Clock Cycles: 1

Exceptions: none

ADDI – Add Immediate

Description:

Calculate the sum of a register and an immediate value and place the result in the target register.

Instruction Format:

w0	$Immediate_{16}$		Rt_5	Ra_5	$04h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rt_5	Ra_5	$04h_6$	I32
w1	$Immediate_{31..0}$					

Operation:

Rt = Ra + Imm

Clock Cycles: 1

Exceptions: none

AND – And Register to Register

Description:

Bitwise ‘and’ two registers and place the result in the target register.

Instruction Format:

w0	$08h_6$	$\sim_5$	Rt_5	Rb_5	Ra_5	$0Ch_6$	RR

Operation:

Rt = Ra & Rb

Clock Cycles: 1

Exceptions: none

ANDI – Bitwise ‘and’ Immediate

Description:

Perform the bitwise ‘AND’ of a register and an immediate value and place the result in the target register.

Instruction Format:

w0	$Immediate_{16}$		Rt_5	Ra_5	$08h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rt_5	Ra_5	$08h_6$	I32
w1	$Immediate_{31..0}$					

Operation:

Rt = Ra & Imm

Clock Cycles: 1

Bcc – Branch on Compare to Register

Description:

Branch if a comparison condition between a register and another register value is true. The 13 bit displacement is shifted left and sign extended before being added to the program counter. The branch range is then +/- 8k half-words. The comparison is for signed arguments.

Instruction Formats:

$Displacement_{13..1}$	$Cond_3$	Rb_5	Ra_5	12_6

Operation:

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

Clock Cycles: 3 if branch is taken, otherwise 1

Conditions:

$Cond_3$	Mne.	Description
0	BEQ	branch if equal
1	BNE	not equal
2	BAND	branch if a is true and b is true
3	BNAND	
4	BLT	signed less than
5	BGE	signed greater than or equal
6	BLE	signed less than or equal
7	BGT	signed greater than

BccU – Branch on Unsigned Compare to Register

Description:

Branch if a comparison condition between a register and another register value is true. The 13 bit displacement is shifted left and sign extended before being added to the program counter. The branch range is then +/- 8k half-words. The comparison is for unsigned arguments.

Instruction Formats:

$Displacement_{13..1}$	$Cond_3$	Rb_5	Ra_5	12_6

Operation:

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

Clock Cycles: 3 if branch is taken, otherwise 1

Conditions:

$Cond_3$	Mne.	Description
0		reserved
1		reserved
2	BOR	reserved
3	BNOR	reserved
4	BLTU	signed less than
5	BGEU	signed greater than or equal
6	BLEU	signed less than or equal
7	BGTU	signed greater than

BccI – Branch on Compare to Immediate

Description:

Branch if a comparison condition between a register and an immediate value is true. The 13 bit displacement is shifted left and sign extended before being added to the program counter. The branch range is then +/- 8k half-words. The immediate value is sign extended before use. The value may be up to 32 bits in size. The comparison is for signed arguments.

Instruction Formats:

w0	$Displacement_{13..1}$	$Cond_3$	$Imm_{4..0}$	Ra_5	02_6	Bri5
w0	$Displacement_{13..1}$	$Cond_3$	$10h_5$	Ra_5	02_6	Bri32
w1	$Immediate_{31..0}$					

Operation:

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

Clock Cycles: 3 if branch is taken, otherwise 1

Conditions:

$Cond_3$	Mne.	Description
0	BEQI	branch if equal
1	BNEI	not equal
2	BANDI	branch if both true (non-zero)
3	BNANDI	branch if not both true
4	BLTI	signed less than
5	BGEI	signed greater than or equal
6	BLEI	signed less than or equal
7	BGTI	signed greater than

BccUI – Branch on Compare to Unsigned Immediate

Description:

Branch if a comparison condition between a register and an immediate value is true. The 13 bit displacement is shifted left and sign extended before being added to the program counter. The branch range is then +/- 8k half-words. The immediate value is sign extended before use. The value may be up to 32 bits in size. The comparison is for unsigned arguments.

Instruction Formats:

w0	$Displacement_{13..1}$	$Cond_3$	$Imm_{4..0}$	Ra_5	03_6	Bri5
w0	$Displacement_{13..1}$	$Cond_3$	$10h_5$	Ra_5	03_6	Bri32
w1	$Immediate_{31..0}$					

Operation:

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

Clock Cycles: 3 if branch is taken, otherwise 1

Conditions:

$Cond_3$	Mne.	Description
4	BLTUI	unsigned less than
5	BGEUI	unsigned greater than or equal
6	BLEUI	unsigned less than or equal
7	BGTUI	unsigned greater than

CALL – Call Subroutine / Method

Description:

The program counter is loaded with the sum of an immediate value specified in the instruction and the contents of register Ra. The address of the next instruction is pushed onto the stack.

If the Ra field has the value 31 then the program counter is used as the Ra register. This allows program counter relative calls to be performed.

Instruction Format:

47 ... 16	15 ... 11	10 ... 6	5 ... 0	
$Address_{32}$	0_5	Ra_5	$10h_6$	
$Immediate_{16}$	0_5	Ra_5	$14h_6$	
	0_5	Ra_5	$1Ch_6$	

Notes:

If Ra is zero then this instruction is executed in the IFETCH stage of the processor and consequently may execute in a single clock cycle. Otherwise three clock cycles are required.

CLI – Clear Interrupt Mask

Description:

This instruction clears the interrupt mask allowing mask-able interrupts to occur. This instruction should typically be used only after the interrupt state is saved with an IPUSH instruction.

Instruction Format:

0_5	$\sim_5$	$18h_6$

CMPI – Compare Immediate

Description:

Perform a signed comparison of a register and an immediate value and place the relationship result in the target register.

Instruction Format:

w0	$Immediate_{16}$		Rt_5	Ra_5	$05h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rt_5	Ra_5	$05h_6$	I32
w1	$Immediate_{31..0}$					

Operation:

```
If Ra < imm then
        Rt = -1
else if Ra = Imm then
        Rt = 0
else
        Rt = 1
```

Clock Cycles: 1

CMPUI – Compare Unsigned Immediate

Description:

Perform an unsigned comparison of a register and an immediate value and place the relationship result in the target register.

Instruction Format:

w0	$Immediate_{16}$		Rt_5	Ra_5	$05h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rt_5	Ra_5	$05h_6$	I32
w1	$Immediate_{31..0}$					

Operation:

```
If Ra < imm then
        Rt = -1
else if Ra = Imm then
        Rt = 0
else
        Rt = 1
```

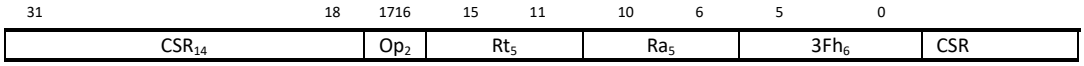
Clock Cycles: 1

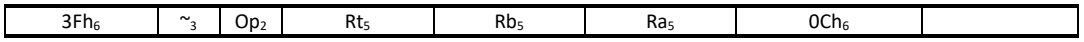
CSR – Control and Status Register Update

Description:

This instruction atomically reads the CSR into a target register then sets it to a value from a register Ra indicated in the instruction. Individual bits in the CSR may be set or cleared by the CSRRS and CSRRC instructions. Which CSR register to access may be specified by an immediate constant in the instruction, or by the contents of register Rb.

Instruction Formats:





Op_2	Mne.	Description
0	CSRRW	Write the entire value of Ra to the CSR
1	CSRRS	Set the bits in the CSR according to the bits set in Ra
2	CSRRC	Clear the bits in the CSR according to the bits set in Ra
3		not used

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

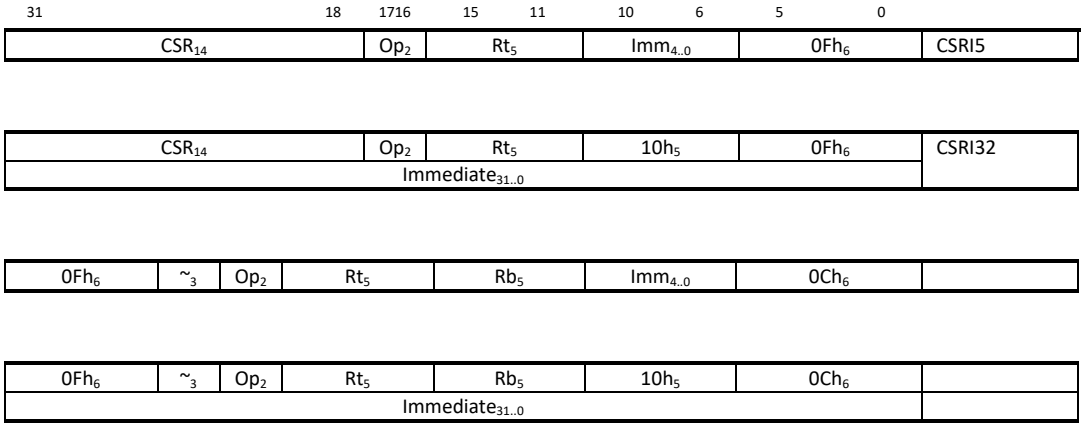
CSR's are determined by the lower 12 bits of the CSR field in the instruction. The upper two bits of the CSR field are reserved, and may be used in the future to resolve the core's operating level.

CSRI – Control and Status Register Update

Description:

This instruction atomically reads the CSR into a target register then sets it to an immediate value supplied by the instruction. 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 register Rb.

Instruction Formats:



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.

INT – Interrupt

Description:

Execute an interrupt. The interrupt executed is identified by a nine bit cause vector. The vector may be used as an index into an exception vector table. The exception return address is the address of the BRK instruction plus the offset specified in the instruction. The exception return address is then either the address of the next instruction or the address of the interrupted instruction depending on the 'O' field in the instruction.

The three general purpose registers (r1, r2, and r29) and the program counter are automatically stored in the top of interrupt stack register.

Further interrupts are automatically masked.

Instruction Format:

O_1	$Cause_9$	$1Bh_6$

Operation:

CSR cause = cause

ITOS <= interrupt mask,r29,r2,r1,program counter

Clock Cycles: 1

Notes:

Nested interrupts may be accomplished by pushing the top of interrupt stack using the IPUSH instruction, then re-enabling interrupts with the CLI instruction. Care must be taken to not allow interrupt nesting more than sixteen levels.

IPOP – Pop from I-Stack

Description:

This instruction pops the top element in the interrupt stack into the itos CSR register. This may be used to modify the return address, r1, r2, r29, or status bits.

Instruction Format:

6_5	$\sim_5$	$18h_6$

IPUSH – Push to I-Stack

Description:

This instruction pushes the contents of the itos CSR register to the internal interrupt stack. This may be used to modify the return address, status bits.

Instruction Format:

5_5	$\sim_5$	$18h_6$

IRET – Return from Interrupt

Description:

Restore registers r1,r2,r29, the program counter, and interrupt mask from the top of interrupt stack register. Clears the semaphore identified by Sm. Semaphore #0 is always cleared.

Instruction Format:

4_5	Sm_5	$18h_6$

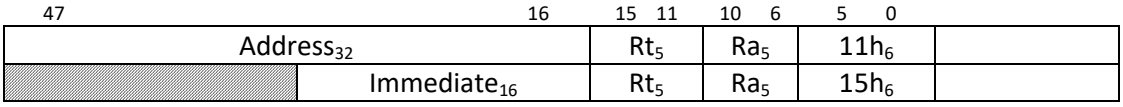
JMP – Jump / Link to Address

Description:

The program counter is loaded with the sum of an immediate value specified in the instruction and the contents of register Ra. The address of the next instruction is stored in the target register.

If the Ra field has the value 31 then the program counter is used as the Ra register. This allows program counter relative calls to be performed.

Instruction Format:



Notes:

If Ra is zero then this instruction is executed in the IFETCH stage of the processor and consequently may execute in a single clock cycle. Otherwise three clock cycles are required.

LDI – Load Immediate

Description:

This is an alternate mnemonic for the ORI instruction where the register Ra is R0. The immediate value is loaded into the target register.

Instruction Format:

w0	$Immediate_{16}$		Rt_5	0_5	$09h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rt_5	0_5	$09h_6$	I32
w1	$Immediate_{31..0}$					

Operation:

Rt = Ra | Imm

Clock Cycles: 1

LH – Load Half

Description:

Loads a half-word of data from memory addressed as the sum of a register (Ra) and an immediate value specified in the instruction. The half-word loaded is sign extended to the width of the register.

Instruction Format:

w0	$Immediate_{16}$		Rt_5	Ra_5	$20h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rt_5	Ra_5	$20h_6$	I32
w1	$Immediate_{31..0}$					

Considerations:

LHU – Load Unsigned Half

Description:

Loads a half-word of data from memory addressed as the sum of a register (Ra) and an immediate value specified in the instruction. The half-word loaded is zero extended to the width of the register.

Instruction Format:

w0	$Immediate_{16}$		Rt_5	Ra_5	$21h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rt_5	Ra_5	$21h_6$	I32
w1	$Immediate_{31..0}$					

Considerations:

LW – Load Word

Description:

Loads a word of data from memory addressed as the sum of a register (Ra) and an immediate value specified in the instruction.

Instruction Format:

w0	$Immediate_{16}$		Rt_5	Ra_5	$22h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rt_5	Ra_5	$22h_6$	I32
w1	$Immediate_{31..0}$					

Considerations:

LWR – Load Word and Reserve Address

Description:

Loads a word of data from memory addressed as the sum of a register (Ra) and an immediate value specified in the instruction. Additionally the address reservation signal is activated.

Instruction Format:

w0	$Immediate_{16}$		Rt_5	Ra_5	$23h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rt_5	Ra_5	$23h_6$	I32
w1	$Immediate_{31..0}$					

Considerations:

MOV – Move Register to Register

Description:

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

Instruction Format:

Rt_5	Ra_5	$1Dh_6$

Clock Cycles: 1

Exceptions: none

NOP – No Operation

Description:

This instruction does not perform any operation, it merely causes the program counter to increment to the next instruction. It may be used to align code.

Instruction Format:

$\sim_{10}$	$1Ah_6$

OR – Or Register to Register

Description:

Bitwise ‘or’ two registers and place the result in the target register.

Instruction Format:

$09h_6$	$\sim_5$	Rt_5	Rb_5	Ra_5	$0Ch_6$

Operation:

Rt = Ra | Rb

Clock Cycles: 1

Exceptions: none

ORI – Bitwise ‘or’ Immediate

Description:

Perform the bitwise ‘OR’ of a register and an immediate value and place the result in the target register.

Instruction Format:

w0	$Immediate_{16}$		Rt_5	Ra_5	$09h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rt_5	Ra_5	$09h_6$	I32
w1	$Immediate_{31..0}$					

Operation:

Rt = Ra | Imm

Clock Cycles: 1

PEA – Push Effective Address

Description:

An address is calculated as the sum of an immediate constant and the value in register Ra. The calculated address is then pushed on the stack. This instruction may also be used to push a constant on the stack.

Instruction Format:

w0	$Immediate_{16}$		0_5	Ra_5	$2Bh_6$	I16
w0	$20h_6$	$\sim_{10}$	0_5	Ra_5	$2Bh_6$	I32
w1	$Immediate_{31..0}$					

Operation:

sp = sp – 2; memory[sp] = Ra + Imm

Clock Cycles:

Considerations:

This seems like a CISC style instruction, but it implements a fairly common operation. Pushing values onto the stack. The author decided to include the instruction after reviewing compiler output which had dozens of the occurrence of: “LD r3,#const; PUSH r3”. Being able to push constants directly onto the stack shortens code.

PUSH – Push Register on Stack

Description:

This instruction pushes the specified register onto the current stack.

Instruction Format:

2_5	$Regno_5$	$19h_6$

$Regno_5$	Register Pushed	
0 to 30	r0 to r30	General purpose Registers
31	sp	Current Stack Pointer

Operation:

SP = SP – 2
memory[SP] = Rn

Assembler Example:

PUSH r1

Considerations:

PUSH is really just a specialized store instruction which uses the stack pointer as an implied register. Because there is only a single register update needed to update the stack pointer the instruction is fairly simple to implement. One of the benefits of a push instruction is a short instruction (16 bits) can be used. PUSH also performs two operations in a single instruction, decrementing the stack pointer, and storing a value to memory. It's good for code density. There is no corresponding POP operation as that's too complex to implement. Unlike a push a POP requires updating two registers at the same time.

A simple compiler will typically push subroutine arguments on the stack before calling the target routine. This can be done with store instructions but is much shorter just to use a PUSH instruction. Typically even in a simple compiler arguments are not popped off the stack. Instead the stack pointer is adjusted directly to effectively remove the arguments. Hence PUSH is used more often than POP.

PUSHI – Push Immediate

Description:

This is an alternate mnemonic for the PEA instruction.

Instruction Format:

w0	$Immediate_{16}$		0_5	0_5	$2Bh_6$	I16
w0	$20h_6$	$\sim_{10}$	0_5	0_5	$2Bh_6$	I32
w1	$Immediate_{31..0}$					

Operation:

SP = SP -2 ; memory[SP] = immediate

Clock Cycles:

RET – Return from Subroutine / Method

Description:

The program counter is popped from the stack. An amount is added to the stack pointer. The amount to add to the stack pointer must include 2 to account for the program counter stored on the stack.

Instruction Formats:

	$00h_5$	Amt_5	$19h_6$	I5
$\sim_{16}$	$00h_5$	$10h_5$	$19h_6$	I32
Amt_{32}				

SEI – Set Interrupt Mask

Description:

This instruction sets the interrupt mask preventing mask-able interrupt from occurring.

Instruction Format:

1_5	$\sim_5$	$18h_6$

SH – Store Half

Description:

Stores a half-word of data to memory addressed as the sum of a register (Ra) and an immediate value specified in the instruction.

Instruction Format:

w0	$Immediate_{16}$		Rs_5	Ra_5	$28h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rs_5	Ra_5	$28h_6$	I32
w1	$Immediate_{31..0}$					

Considerations:

SUB – Subtract Register to Register

Description:

Subtracts two registers and place the result in the target register.

Instruction Format:

$07h_6$	$\sim_5$	Rt_5	Rb_5	Ra_5	$0Ch_6$

Operation:

Rt = Ra - Rb

Clock Cycles: 1

Exceptions: none

SUBI – Subtract Immediate

Description:

Calculate the sum of a register and an immediate value and place the result in the target register. The immediate value is negated by the assembler. This is an alternate mnemonic for the ADDI instruction.

Instruction Format:

w0	$Immediate_{16}$		Rt_5	Ra_5	$04h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rt_5	Ra_5	$04h_6$	I32
w1	$Immediate_{31..0}$					

Operation:

Rt = Ra + -Imm

Clock Cycles: 1

SW – Store Word

Description:

Stores a word of data to memory addressed as the sum of a register (Ra) and an immediate value specified in the instruction.

Instruction Format:

w0	$Immediate_{16}$		Rs_5	Ra_5	$29h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rs_5	Ra_5	$29h_6$	I32
w1	$Immediate_{31..0}$					

Considerations:

SWC – Store Word and Clear Reservation

Description:

Conditionally stores a word of data to memory addressed as the sum of a register (Ra) and an immediate value specified in the instruction. The word is stored to memory only if the address is still reserved by the core. The reservation status is present in bit #0 of the semaphore register.

Instruction Format:

w0	$Immediate_{16}$		Rs_5	Ra_5	$29h_6$	I16
w0	$20h_6$	$\sim_{10}$	Rs_5	Ra_5	$29h_6$	I32
w1	$Immediate_{31..0}$					

Considerations:

XOR – Bitwise ‘exclusive or’ Register

Description:

Perform the bitwise exclusive ‘OR’ of a register and another register and place the result in the target register.

Instruction Format:

w0	$0Ah_6$	$\sim_5$	Rt_5	Rb_5	Ra_5	$0Ch_6$	RR

Operation:

Rt = Ra ^ Rb

Clock Cycles: 1

XORI – Bitwise ‘exclusive or’ Immediate

Description:

Perform the bitwise exclusive ‘OR’ of a register and an immediate value and place the result in the target register.

Instruction Format:

w0	$Immediate_{16}$		Rt_5	Ra_5	$0Ah_6$	I16
w0	$20h_6$	$\sim_{10}$	Rt_5	Ra_5	$0Ah_6$	I32
w1	$Immediate_{31..0}$					

Operation:

Rt = Ra ^ Imm

Clock Cycles: 1

Opcode Maps

Major Opcodes

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x			Bccl	BccUl	ADDI	CMPI	CMPUI		ANDI	ORI	XORI		{r2}	{r3}		CSRI
1x	CALL	JMP	Bcc	BccU	CALL16	JMP16			{sys}	{mem}	NOP	INT	CALL0	MOV		CINSN
2x	LH	LHU	LW	LWR					SH	SW	SWC	PEA				
3x	MULI	MULUI	MULSUI	MULHI	MULUHI	MULSUHI			DIVI	DIVUI	DIVSUI	REMI	REMUI	REMUSI		CSR

{sys} Funct$_4$ Opcodes

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	CLI	SEI			IRET	IPUSH	IPOP									
1x																

{mem} Funct$_5$ Opcodes

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	RET		PUSH	POP	PUSHI											
1x																

{r2} Funct$_7$ Opcodes

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x					ADD	CMP	CMPU	SUB	AND	OR	EOR		NAND	NOR	ENOR	
1x	SHL	SHR	ASR	ROL	ROR		SXB	SXH	SHLI	SHRI	ASRI	ROLI	RORI			
2x	LHX	LHUX	LWX	LWRX					SHX	SWX	SWCX					
3x	MUL	MULU	MULSU	MULH	MULUH	MULSUH			DIV	DIVU	DIVSU	REM	REMU	REMSU		

http://github.com/robfinch/Cores/blob/master/DSD/trunk/rtl/DSD7.v