DSD7

## 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 data stack pointer. The stack pointer although dedicated for stack access may also be used as a general purpose register.

|  |  |  |
| --- | --- | --- |
| 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-r25 | function arguments | caller |
| r26 | type number / function argument | caller |
| r27 | class pointer / function argument | caller |
| r28 | thread pointer | callee |
| r29 | global pointer |  |
| 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. The program counter is automatically stored on the return address stack of the running task when a CALL instruction is executed. The program counter will be popped off the return address stack by the RET instruction.

The program counter is split into two sections, the lower 24 bits which increment as instructions are fetched, and the upper eight bits which act as a program bank.

|  |  |
| --- | --- |
| 31 24 | 23 0 |
| Bank31..24 | Counter23..0 |

## 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, task register, 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 ihold CSR.

# 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.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Regno11 | 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 |  |
|  | | | | |  |
| 0x00D | 32 | M | SP | alternate stack pointer |  |
| 0x010 | 32 |  | TCBP | tcb pointer register |  |
| 0x011 | 32 |  | CISC | compressed instruction set control |  |
| 0x012 | 32 | M | STATUS | status register |  |
| 0x017 | 32 |  | BANK | Program Bank Registers |  |
| 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.

## Scratch (CSR #009h)

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

## TCB Pointer Register (CSR #010h)

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

|  |  |
| --- | --- |
| 31 9 | 8 0 |
| TCB Address31..9 | 09 |

## Compressed Instruction Set Control (CSR #011h)

|  |  |  |
| --- | --- | --- |
| 31 20 | 19 8 | 7 0 |
| CIT Address31..20 | ~12 | ISID8 |

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.

## Bank CSR #017h

This register contains three bank values selectable by the CALL and JMP instructions. The bank value is loaded into the bank portion of the program counter register. The lower eight bits of this register are read-only.

|  |  |  |  |
| --- | --- | --- | --- |
| 31 24 | 23 16 | 15 8 | 7 0 |
| Bank3 | Bank2 | Bank1 | Current Bank |

## IHOLDL CSR #040h

This register contains the return address for the exceptioned instruction.

## IHOLDH CSR #041h

This register contains the task to return to and cpu status bits.

|  |  |
| --- | --- |
| 31 8 | 7 0 |
| Status Bits | Task Number8 |

Format:

# Definitions

Word: A word is a 32 bit quantity

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

# Data 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.

# Detailed Instruction Set Description

## 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 | Displacement13..1 | Cond3 | Imm4..0 | Ra5 | 026 | Bri5 |
| w0 | Displacement13..1 | Cond3 | 10h5 | Ra5 | 026 | Bri32 |
| w1 | Immediate31..0 | | | | |

Operation:

if (Ra cond #imm)

pc <= pc + displacement

Clock Cycles: 3 if branch is taken, otherwise 2

Conditions:

|  |  |  |
| --- | --- | --- |
| Cond3 | 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

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 | Displacement13..1 | Cond3 | Imm4..0 | Ra5 | 036 | Bri5 |
| w0 | Displacement13..1 | Cond3 | 10h5 | Ra5 | 036 | Bri32 |
| w1 | Immediate31..0 | | | | |

Operation:

if (Ra cond #imm)

pc <= pc + displacement

Clock Cycles: 3 if branch is taken, otherwise 2

Conditions:

|  |  |  |
| --- | --- | --- |
| Cond3 | 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 |

## BRK – Break

Description:

Execute a breakpoint 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. This instruction may be followed by an instruction word of information. The exception return address is the address of the BRK instruction plus the offset specified in the instruction.

Instruction Format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| w0 | ~13 | | Offs4 | Vector9 | 00h6 | |  |
| w0 | 20h6 | ~7 | Offs4 | Vector9 | | 00h6 | I32 |
| w1 | Data31..0 | | | | | |

Operation:

cause = vector

Clock Cycles: 1

CALL – Call Subroutine / Invoke Method

Description:

This instruction transfers the control of a program to a different point. The program counter is loaded with the value specified by the instruction. The return address is pushed to the stack. The program bank is loaded from the selected bank register.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| w0 | Sel1..0 | Address24..1 | | | 11h6 | J26 |
| h0 |  | | 54 | Ra5 | 5Ah7 | MEM |

Bank Selection:

|  |  |
| --- | --- |
| Sel1..0 | Selection |
| 0 | Current program bank |
| 1 | Bank register #1 |
| 2 | Bank register #2 |
| 3 | Bank register #3 |

## 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.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 18 | 1716 | 15 11 | 10 6 | 5 0 |  |
| CSR14 | Op2 | Rt5 | Imm4..0 | 0Fh6 | CSRI5 |
| CSR14 | Op2 | Rt5 | 10h5 | 0Fh6 | CSRI32 |
| Immediate31..0 | | | | |

|  |  |  |
| --- | --- | --- |
| Op2 | Mne. | Description |
| 0 | CSRRWI | Write the entire value of register Ra to the CSR |
| 1 | CSRRSI | Set the bits in the CSR according to the bits set in Ra |
| 2 | CSRRCI | 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.

## IPOP – Pop from I-Stack

Description:

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

## IPUSH – Push to I-Stack

Description:

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

## 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 | Immediate16 | | Rt5 | | Ra5 | 20h6 | | I16 |
| w0 | 20h6 | ~10 | Rt5 | Ra5 | | | 20h6 | I32 |
| w1 | Immediate31..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 | Immediate16 | | Rt5 | | Ra5 | 21h6 | | I16 |
| w0 | 20h6 | ~10 | Rt5 | Ra5 | | | 21h6 | I32 |
| w1 | Immediate31..0 | | | | | | |

Considerations:

## 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:

|  |  |
| --- | --- |
| ~10 | 1Ah6 |

## PUSH – Push Register on Stack

Description:

This instruction pushes the specified register onto the current stack.

Instruction Format:

|  |  |  |
| --- | --- | --- |
| 25 | Regno5 | 19h6 |

|  |  |  |
| --- | --- | --- |
| Regno5 | Register Pushed |  |
| 0 to 30 | r0 to r30 | General purpose Registers |
| 31 | sp | Current Stack Pointer |

Operation:

SP = SP – 4

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.

## RET – Return from Method / Subroutine

Description:

This instruction pops the program counter from the stack and adds a five bit zero extended offset shifted left once. An amount may also be added to the stack pointer to remove parameters from the stack. The amount specified should be at least two to account for popping the return address. Typically the offset will be zero, however it may be non-zero in order to return past the calling point a number of half-words which may be used for static parameters.

Instruction Format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | | Offs5 | ~5 | 1Bh6 | |  |
| w0 | Immediate16 | | Offs5 | ~5 | 01h6 | | I16 |
| w0 | 20h6 | ~10 | Offs5 | ~5 | | 01h6 | I32 |
| w1 | Immediate31..0 | | | | | |

Operation:

PC = memory[SP] + offset \* 2

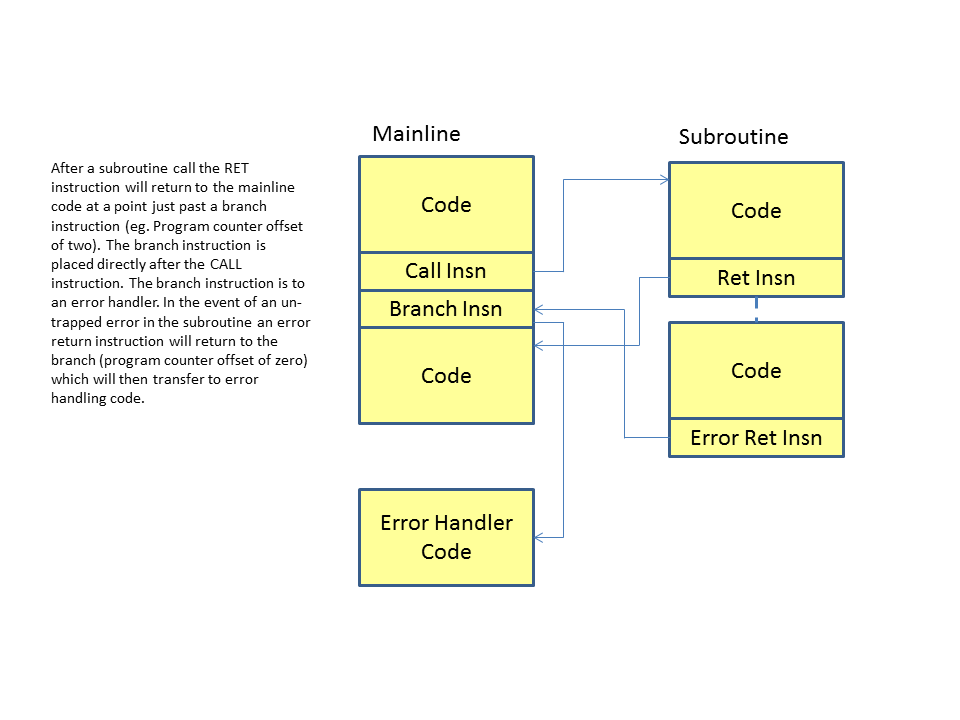
SP = SP + imm

Clock Cycles: 3

Considerations:

When adding an amount to the stack pointer space for the return address must be accounted for in the immediate constant. This requirement is in lieu of needing a triple input adder to add the constant plus two for the return address. The short form of the instruction does not have this problem as the additional amount added to the stack pointer would be zero.

Being able to specify a return past the calling point is useful as well in some circumstances. For instance static parameters may be passed directly in the code. Another case where it may be useful to return past the calling point is in the case of error handling. Two return points could be used, one point directly after the call point could be a jump to an error handler. The second point a word or two further in the instruction stream would be the continuance of normal processing.



# Opcode Maps

## Major Opcodes

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | BRK | RET | BccI | BccUI | ADDI | CMPI | CMPUI |  | ANDI | ORI | XORI |  | {r2} | {r3} |  | CSRI |
| 1x | JMP | CALL | Bcc | BccU |  |  |  |  | {sys} | {mem} | NOP | RET2 |  |  |  | CINSN |
| 2x | LH | LHU | LW | LWR |  |  |  |  | SH | SW | SWC |  |  |  |  |  |
| 3x | MULI | MULUI | MULSUI | MULHI | MULUHI | MULSUHI |  |  | DIVI | DIVUI | DIVSUI | REMI | REMUI | REMUSI |  | CSR |

## {mem} Funct4 Opcodes

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x |  |  | PUSH | POP | JMPR | CALLR |  |  |  |  |  |  |  |  |  |  |

## {r2} Funct7 Opcodes

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