RACORS

sf32

Family of
32-bit microprocessors

Base ISA Reference Manual

Revision 1.0
20 December 2013

Author: Martin Raubuch

Revision History

Revision	Date	
0.9	23Nov2012	First version, largely based on structure of the eco32b ISA reference manual
1.0	20Dec2013	• Review and typo fixes, property note changed to RACORS GmbH • **`ibos`** and **`ibol`** instructions added (endianess conversion) • sfr32 variant removed

Table of contents

1 Overview

1.1 Introduction

The sf32 family of microprocessors is targeted at computing, embedded control and DSP applications. With fixed length 32-bit instruction coding the family is focused on high clock rates and small core implementations. Code density is also very competitive among architectures with 32-bit fixed length coding but is not a primary focus of the sf32 architecture.

To be able to support a wide range of applications with performance and cost optimized solutions the sf32 family defines a base (b) ISA and various extension ISAs. The base ISA supports general purpose control and computing with minimum overhead to enable very cost efficient implementations. The extension ISAs add advanced DSP capabilities for compute intensive applications. The extension ISAs are defined in separate ISA reference manuals.

The base ISA is a 32-bit general purpose load/store architecture. Accesses to memory data operands and computations are decoupled by using separate instructions. Memory operands are accessed with load/store instructions exclusively. Computation instructions have register or constant source operands and register destination operands. This concept supports the implementation of variants with different pipeline structures and sizes. High level language compilers can schedule instructions in an optimal order for efficient execution with minimal stalls and pipeline bubbles.

1.2 Feature Summery

The following list summarizes the sf32b's main features

- Load/store architecture
- Harvard architecture with separate instruction and data address spaces
- 4 GBytes instruction address space and 4 GBytes data address space
- Fixed length 32-bit instruction coding
- 24 x 32-bit general purpose registers and 7 special registers
- Native support for 8-bit, 16-bit and 32-bit signed and unsigned integer data types
- Rich set of load/store addressing modes
- Bit manipulation & test instructions: set, clear, toggle & test
- 32*32 multiply instructions with either 32-bit high word or 32-bit low word results
- Optional conditional execution of most instructions
- Software controlled branch speculation
- 16 interrupts with programmable start addresses
- Flexible debug support for application optimized debug concepts
- 32-bit loop counter

1.3 Scope of this manual

This sf32 base ISA reference manual contains the following detailed descriptions:

- Instruction set
- Instruction coding
- Size and endianess of instruction and data address spaces
- Registers of the programming model (user registers)
- Register and memory operand types
- Register and memory operand addressing modes
- Operation modes
- Interrupt concept
- Debugging concept

Implementation specific details such as I/O signals, cycle by cycle timing of instructions, operand dependencies and latencies are not part of this ISA reference manual. These details are described in the IMA (Implementation Architecture) reference manual of each implementation.

1.4 Structure of this manual

Below are brief descriptions of the following chapters of this manual:

Definitions, acronym definitions for registers, constants and other sf32 base ISA specific items that are used in the remaining chapters of the document.

Programming model, describes the address spaces and user registers

Instruction set summery, brief descriptions of addressing modes and instructions divided into functional groups

Operation modes, describes the properties of the system and application operation modes, also defines the reset state, interrupt concept and software debug support concept.

Addressing modes, defines bit accurate details of how operands are generated or calculated

Load, store and move instructions, defines bit accurate details of the operations and addressing modes of these instructions

Computation instructions, defines bit accurate details of the operations and addressing modes of these instructions

Flow control instructions, defines bit accurate details of the operations and addressing modes of these instructions

Instruction Coding, tables with instruction coding details in alphabetical order

2 Definitions

2.1 Register Specifications

This section defines the variables and notations used to specify register operands in addressing mode and instruction descriptions.

Rn one of the thirty-two registers **R0**, **R1**, **R2**, **R3**, **R4**, **R5**, **R6**, **R7**, **R8**, **R9**, **RA**, **RB**, **RC**, **RD**, **RE**, **RF**, **RP**, **RQ**, **RU**, **RV**, **RW**, **RX**, **RY**, **RZ**, **LC**, **CC**, **CS**, **EP**, **TA**, **SA**, **IA** or **ID**.

Rs one of registers **Rn** used as source operand, **Rs** is used in addressing modes with a single register source operand

Rs0 one of registers **Rn** used as source operand 0, **Rs0** specifies the first source operand (assembly language operand fields) in addressing modes with two source operands; for non-commutative operations like subtract or compare **Rs0** is the operand on the right side of the operator, e.g. for subtract and compare instructions the operation is **Rs1** - **Rs0**. If used with indirect shift or bit manipulation instructions **Rs0** contains the shift-count, or bit-index operands.

Rs1 one of registers **Rn** used as source operand 1, **Rs1** specifies the second source operand (assembly language operand fields) in addressing modes with two source operands; for non-commutative operations like subtract or compare **Rs1** is the operand on the left side of the operator, e.g. for subtract and compare instructions the operation is **Rs1** - **Rs0**.

Rd one of registers **Rn** used as destination operand.

Ru one of registers **Rn** used as post update operand in indirect data memory addressing modes with indirect post-update. **Ru** is added to the indirect address register **An** after the memory access.

Rx one of registers **Rn** used as index in the indirect data memory addressing mode with scaled index. The effective address of the data memory access is **Rx** shifted left by the size of the operand and added to the content of the indirect address register **An**.

An one of the sixteen address registers **R8**, **R9**, **RA**, **RB**, **RC**, **RD**, **RE**, **RF**, **RP**, **RQ**, **RU**, **RV**, **RW**, **RX**, **RY** or **RZ**; **An** is used as indirect memory address in addressing modes with memory source or destination operands.

RGS specifies a selection of registers for load and store instructions with multiple source or destination operands; the selection can include one or more of the following registers: **R0**, **R1**, **R2**, **R3**, **R4**, **R5**, **R6**, **R7**, **R8**, **R9, RA**, **RB**, **RP**, **RQ**, **RU**, **RV**, **LC**, **TA**, **SA**.

2.2 Constant Specifications

This section defines the acronyms and notations used to specify constant operands in addressing mode and instruction descriptions. For instruction address parameters the size of constant parameters in the opcode is two bits smaller than the parameter. This is because sf32 instruction addresses are byte addresses but with the alignment of opcodes on 32-bit word boundaries the two LSBs of instruction addresses and of instruction address related constant parameters are always zero.

Acronyms for constants with a value range have a one- character suffix with the following meaning:

U (Unsigned) or **S** (Signed)

C12$_{U}$ 12-bit constant (unsigned) used as source operand in an addressing mode for conditional add/subtract instructions; legal values are from 0 to 4095.

C16$_{U}$ 16-bit constant (unsigned) used as source operand, legal values are from 0 to 65535

C32$_{U}$ 32-bit constant (unsigned) used as source operand with the **addh** instruction, legal values are from 0x00000000 to 0xFFFF0000; bits [15:0] are not coded and are always zero.

C16$_{S}$ 16-bit constant (signed) used as source operand, legal values are from -32768 to 32767

C17$_{S}$ 17-bit constant (signed) used as source operand, legal values are from -65536 to 65535

DO12$_{S}$ 12-bit data address offset (signed) used in an indirect memory addressing mode. Legal values are from -2048 to 2047 [bytes].

AU12$_{S}$ 12-bit data address update (signed) used in one of the indirect memory addressing modes. Legal values are from -2048 to 2047 [bytes].

DA16$_S$ 16-bit direct data address (signed) used in the direct memory addressing mode, the 16-bit value from the opcode is sign extended to 32-bit, legal, direct 32-bit data addresses are from 0x00000000 to 0x00007FFF and from 0xFFFF8000 to 0xFFFFFFFF.

SHC5$_U$ 5-bit shift count (unsigned) used in addressing modes for shift instructions. Legal values are from 0 to 31

BTI5$_U$ 5-bit bit index (unsigned) used in addressing modes for bit-manipulation instructions, legal values are from 0 to 31

IO16$_S$ 16-bit instruction address offset (signed) used with branch instructions. The two LSBs are not coded (always zero). Legal values are from -32768 to 32764

IA29$_U$ 29-bit direct instruction address (unsigned) used in an addressing mode for jump and jump to subroutine instructions. The two LSBs are not coded (always zero). Legal values are from 0x00000000 to 0x1FFFFFFC

CND 4-bit field that specifies one of 15 conditions, used to specify the execution condition for instructions with conditional execution

S a single bit constant that can be used to determine if a conditional branch is taken or not in branch speculation situations, legal values are 0 (branch not taken) and 1 (branch taken)

2.3 Miscellaneous definitions

opcode operation code of an instruction; contains sub codes that specify the instruction type and the operands. The sf32 has fixed length 32-bit **opcodes** stored in the instruction memory

eda effective data address, a 32-bit byte address that points to an operand in the data address space, **eda** addresses need not be aligned on the size of the operand.

eia effective instruction address, a 32-bit byte address that points to an **opcode** word in the instruction address space, instructions must be aligned on 32-bit word boundaries, the two LSBs of an effective instruction address are always zero.

som system operation mode, all processor resources are available and all instructions can be executed without restrictions. Bus signals are provided to protect critical memory areas and peripherals.

aom application operation mode, some instructions are illegal and attempts to execute them cause security exceptions.

3 Programming model

3.1 Instruction address space

3.1.1 Size and addressing scheme

The sf32 processors have a 4 GBytes instruction address space. Instruction addresses are 32 bits and point to byte locations in the instruction memory. Instructions must be aligned on 32-bit boundaries. The least significant two bits of instruction addresses are always zero.

3.1.2 Endianess

The sf32 implements a little endian scheme to map 32-bit opcodes to memory words. In case the instruction interface is wider than 32 bits (e.g. 64-bit or wider in super-scalar implementations) the lower address is mapped to the lower bits of the memory word.

3.2 Data address space

3.2.1 Size and addressing scheme

The sf32 processors have a 4 GBytes data address space. Data addresses are 32 bits and point to byte locations in the data memory. The base ISA supports byte (8-bit), short (16-bit) and long (32-bit) memory operands.

3.2.2 Operand types

Operands accessed in the data address space can be unsigned or signed (2’s complement). Inside the processors all arithmetic is done on 32-bit operands. Smaller operands (8-bit, 16-bit) are either sign-extended (signed operands) or zero-extended (unsigned operands) when loaded from memory into one of the general purpose registers. When register operands are stored to memory they are truncated to the size of the destination operand by discarding the 24 MSBs (8-bit destination operand) or the 16 MSBs (16-bit destination operand).

3.2.3 Alignment

The sf32 processors do not handle misaligned memory operands internally. For 32-bit accesses the two LSBs of the memory address are ignored. For 16-bit accesses the LSB is ignored. However the full data space address including the two LSBs is output to the data bus with every access regardless of the operand size. If required by an application misaligned operands can be supported by the memory controller. The processor’s data bus signals provide both the size of the access and the full byte address.

3.2.4 Endianess

The sf32 implements a little endian scheme to map 8-bit, 16-bit and 32-bit data operands to memory words. Endianess conversion instructions are available to efficiently support big endian data objects.

3.2.5 Summery table

The table below illustrates the mapping of data operands into 32-bit memory words. All operands are aligned to memory words and to their own size.

32-bit Memory words	1				0			
Memory addresses	n+4				n			
Long (32-bit) operands	1				0			
Long operands addresses	n+4				n			
Short (16-bit) operands	3		2		1		0	
Short operands addresses	n+6		n+4		n+2		n	
Byte (8-bit) operands	7	6	5	4	3	2	1	0
Byte operands addresses	n+7	n+6	n+5	n+4	n+3	n+2	n+1	n

3.3 Registers

3.3.1 Terminology

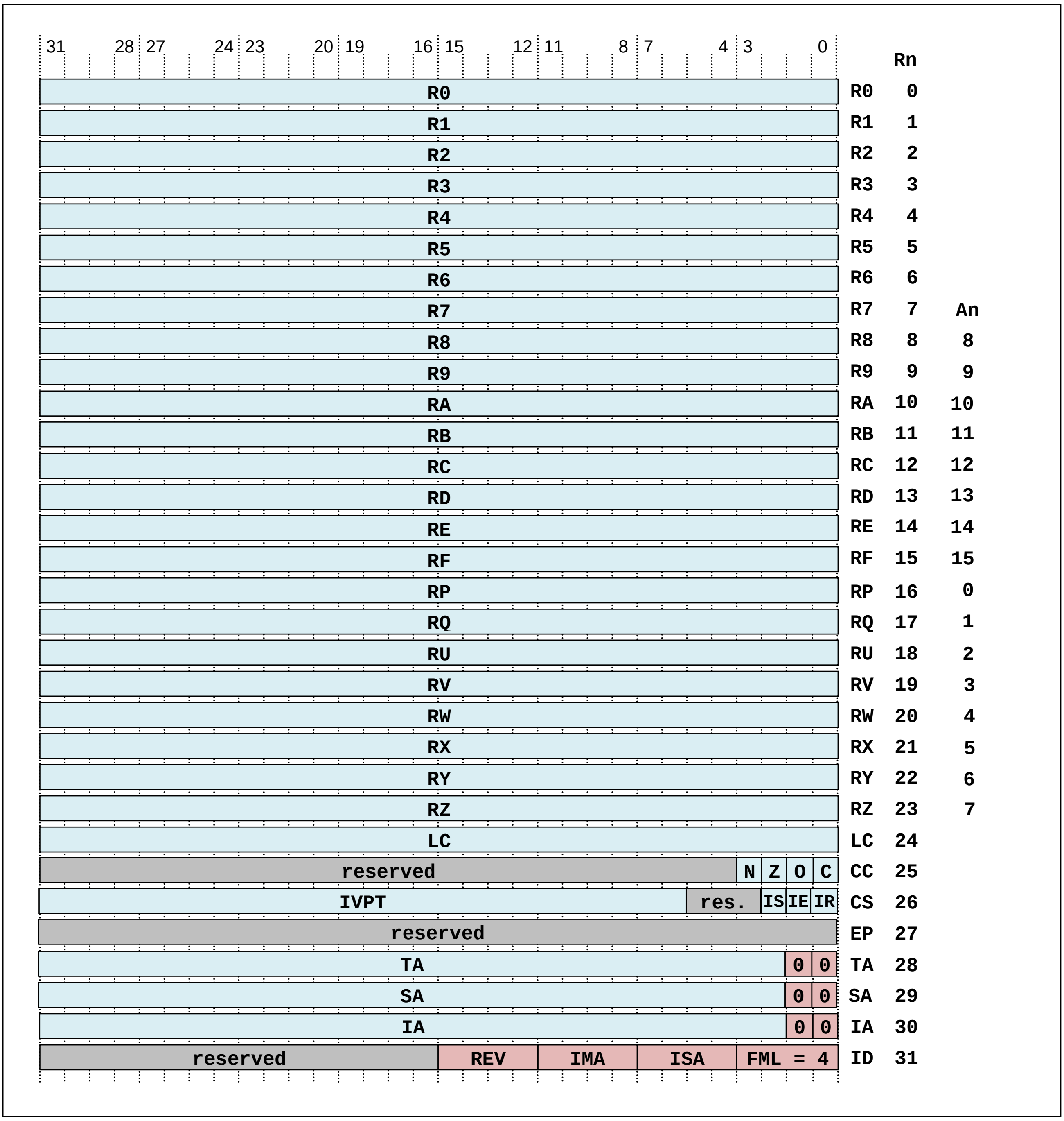
Register values are represented with the LSB at the right most bit position and the MSB at the left most bit position. For an n-bit register the LSB is bit number 0 and the MSB is bit number n-1.

If a register contains multiple named bits or bit-fields then these individual bits or bit-fields are referenced by the register name followed by a '.' character as separator and then followed by the name of the named bit or bit-field as shown below:

<register name>.<bit or bit field name>

For registers that contain a single named bit-field this bit-field has the same name as the register. For example, special register **LC** contains a single 32-bit bit-field with the name **LC**.

3.3.2 Registers



31 … 0		Rn	An
R0	R0	0	
R1	R1	1	
R2	R2	2	
R3	R3	3	
R4	R4	4	
R5	R5	5	
R6	R6	6	
R7	R7	7	
R8	R8	8	8
R9	R9	9	9
RA	RA	10	10
RB	RB	11	11
RC	RC	12	12
RD	RD	13	13
RE	RE	14	14
RF	RF	15	15
RP	RP	16	0
RQ	RQ	17	1
RU	RU	18	2
RV	RV	19	3
RW	RW	20	4
RX	RX	21	5
RY	RY	22	6
RZ	RZ	23	7
LC	LC	24	
reserved (31–4), N (3), Z (2), O (1), C (0)	CC	25	
IVPT (31–6), res. (5–3), IS (2), IE (1), IR (0)	CS	26	
reserved	EP	27	
TA (31–2), 0 (1), 0 (0)	TA	28	
SA (31–2), 0 (1), 0 (0)	SA	29	
IA (31–2), 0 (1), 0 (0)	IA	30	
reserved (31–16), REV (15–12), IMA (11–8), ISA (7–4), FML = 4 (3–0)	ID	31	

3.3.3 Register Details

The sf32 has a single register space containing 24 general purpose registers and 8 special registers. In total these are 32 registers numbered from 0 to 31 that are addressed by 5-bit fields in instruction opcodes and are referred to as **Rn**. A second logical sub-group **An** is defined that contains registers **R8** – **RZ** which can be used as indirect address and is addressed by 4-bit fields in instruction opcodes. The sub-group **An** is contained in **Rn**.

The term “general purpose” is used for registers that can be used as source or destination operands of any computation instruction or load/store/move instruction. General purpose registers can also be used as index or update operand in memory addressing modes.

The term ‘special’ is used for registers **LC**, **CC**, **CS**, **EP**, **TA**, **SA**, **IA** and **ID**. The special registers cannot be used as source or destination of all computation and load/store/move instruction. This may be because of a special function (**CC**, **TA**, **SA**, **IA**), because of potential security violations (**CS**, **IA**), because the register is read-only (**ID**) or because the format and bit width is not suitable for computations (**CC**, **TA**, **SA**, **IA**, **CS**). All special registers except **ID** can be used as destination of move (**`move`**, **`mfdp`**) instructions but not as destination of any computation instructions.

The **EP** register is reserved for parameters of ISA extensions. The base ISA does not use the **EP** register.

The table below summarizes the sf32 register properties. The following paragraphs provide detailed information of register groups and individual registers.

	`R0-R7`	**`R8-RZ`**	**`LC`**	**`CC`**	**`CS`**	**`TA`**	**`SA`**	**`IA`**	**`ID`**
general purpose	yes	yes	no	no	no	no	no	no	no
can be used as destination of move instructions	yes	yes	yes	yes	yes	yes	yes	yes	no
can be used as destination of computation instr.	yes	yes	no	no	no	no	no	no	no
can be used as indirect address	no	yes	no	no	no	no	no	no	no
can be used as source/dest. of load/store	yes	yes	yes	no	no	yes	yes	no	no
can be used as indirect memory address index	yes	yes	yes	no	no	no	no	no	no
can be used as indirect memory address update	yes	yes	yes	no	no	no	no	no	no
can be part of an RGS (Register Selection)	yes	**`RP–RV`**	yes	no	no	yes	yes	no	no
can be modified in application mode	yes	yes	yes	yes	no	yes	yes	no	yes

R0-R7 Eight 32-bit general purpose registers intended for data operands

R8-RZ Sixteen 32-bit general purpose registers intended for data or address operands

LC 32-bit loop counter; used as loop counter with the **`brlc`** (loop counter branch) instruction to improve code density and performance of inner loops;

CC Condition Code. This 4-bit register contains the condition code flags **C**,**O**,**Z** and **N**. **CC** is a source operand of conditional branch instructions and of other instructions with conditional execution. **CC** is a destination operand of some selected computation instruction. The rules of how these instructions update the flags in **CC** are part of the detailed descriptions of these instructions; **CC** cannot be used as source or destination of memory load/store instructions; a hidden shadow register exists to save **CC** when an interrupt is started and to restore the original state of **CC** at the end of an interrupt

CC.C Carry flag. The **C** flag is set by add/subtract/compare arithmetic instructions that update the **CC** register if a carry occurs from bit 31 to bit 32 and is cleared otherwise. Most other instructions that update the **CC** register clear the carry flag. A special case is the **andb** (logic and) instruction. It updates the **CC.C** bit with the parity of the operation result. The flag is set in case of odd parity and is cleared in case of even parity.

CC.O Overflow flag. The **O** flag is set by add/subtract/compare arithmetic instructions that update the **CC** register if an arithmetic overflow occurs from bit 31 to bit 32 and is cleared otherwise. For arithmetic overflow generation the source and destination operands are treated as signed 2’s complement numbers. Most other instructions that update the **CC** register clear the overflow flag. A special case is the **andb** (logic and) instruction. It sets the **CC.O** bit if the result of the operation has odd parity and if the **CC.C** bit is set from a preceding instruction.

CC.Z Zero flag. The **Z** flag is set by instructions that update the **CC** register if the 32-bit result of the operation is zero (all 32 bits zero) and is cleared otherwise.

CC.N Negative flag. The **N** flag is set by instructions that update the **CC** register if the 32-bit result of the operation is negative (bit 31 set) and is cleared otherwise.

CS 32-bit Control and Status; This 32-bit register contains a number of control and status flags and also the pointer to the interrupt vector table in the data address space. Writing to **CS** is possible only in the **som** (System Operation Mode). Attempts to write to **CS** in the **aom** (Application Operation Mode) triggers a security exception; **CS** cannot be used as source or destination of memory load/store instructions; when **CS** is used as destination register of move instructions only the **IVTP** is updated with the corresponding bits of the destination operand the **IR**, **IE** and **IS** flags remain unchanged

CS.IR Interrupt status flag; this flag is set when the sf32 enters an interrupt service routine and is cleared when the processor exits an interrupt service routine.

CS.IE Interrupt Enable; this flag enables or disables interrupts; interrupt requests are acknowledged only if **IE** is set.

CS.IS Interrupt enable Save bit; this flag saves a copy of **CS.IE** when a **`scie`** (save and clear interrupt enable) instruction is executed. Execution of an **`rsie`** (restore interrupt enable) instruction copies **CS.IS** back to **CS.IE**. The **IS** bit together with the **`scie`** and **`rsie`** instructions are used to temporarily disable interrupts and then restore the original interrupt enable state.

CS.IVTP Interrupt Vectors Table Pointer; this 26-bit field defines the most significant bits of the 32-bit start address of the interrupt vector table in the data address space. The table is aligned on a 64 bytes boundary. The six LSBs of the 32-bit table address are all zeros and are not contained in the **CS** register. When the sf32 starts an interrupt service routine it fetches the start address of the routine from the table pointed to by **IVTP**.

TA 32-bit target address; used for indirect jumps and jump to subroutines; points to a 32-bit word (aligned) in the instruction address space; the two LSBs are hard wired to zero

SA 32-bit subroutine return address; points to a 32-bit word (aligned) in the instruction address space; the two LSBs are hard wired to zero; when a **`jpsr`** (jump to subroutine) instruction is executed the return address (address of the next instruction following the **`jpsr`** instruction) is stored in **SA**; when an **`rtsr`** (return from subroutine) instruction is executed **SA** is used as return address

IA 32-bit interrupt return address; points to a 32-bit word (aligned) in the instruction address space; the two LSBs are hard wired to zero; when an interrupt is started the return address (address of the next instruction following the last instruction executed before the interrupt) is stored in **IA**; when an **`rtir`** (return from interrupt) instruction is executed **IA** is used as return address

ID Core ID; This register provides a 16-bit identification code of the processor divided into four separate 4-bit fields; **ID** is a read-only register; writing to **ID** has no effect

ID.FML Core family; This 4-bit code identifies the core family. The code for the sf32 is 4. This code is to distinguish the processor from other RACORS architectures that have a similar **ISA** concept, e.g. from processors of the eco32, eco16 and sf16 families.

ID.ISA Instruction Set Architecture. This 4-bit code identifies the processor’s **ISA**. The following **ISA** codes are defined for the sf32: 1 = base (b), 2 = dsp (d).

ID.IMA Implementation Architecture. This 4-bit code identifies the hardware implementation architecture of the processor. The following codes are defined: 1 = light (l), 2 = performance (p), 3 = superscalar (s), 4 = ultra-light (u). An **IMA** code of 0 is used for the ISS (Instruction Set Simulation) reference model of an ISA, which is not an actual (hardware) implementation.

ID.REV Revision. This is the 4-bit revision code. The first revision is 1. A value of zero is illegal. The revision number is relative to the core type, **IMA** and **ISA**. This means that processors with different **IMA**, **ISA** or core type can have the same **REV** code.

4 Instruction set summery

4.1 *Addressing modes*

This section provides short descriptions of the base ISA addressing modes. The term “register” stands for a register of the **Rn** group.

4.1.1 Data memory addressing modes

These addressing modes are used by load and store instructions to determine the **eda** of the memory source (load) or destination (store) operand(s) and an optional update operation of an indirect address register. Most memory addressing modes have an execution condition **CND**.

DA16$_S$,CND 16-bit absolute data address, 0x00000000 – 0x00007FFF and 0xFFFF8000 – 0xFFFFFFFF, conditional
(DO12$_S$,An),CND Indirect data address with 12-bit signed offset, conditional
(Rx,An),CND Indirect data address with scaled index, conditional
(An,AU12$_S$)*,CND Indirect data address with direct, signed post-update, conditional
(An,Ru)*,CND Indirect data address with indirect post-update, conditional
(An)+ Indirect data address with scaled post-increment
-(An) Indirect data address with scaled pre-decrement

4.1.2 Registers only addressing modes

Rs Single register, **Rs** = source operand
Rd Single register, **Rd** = destination operand
Rs,Rd,CND Dual registers conditional, **Rs** = source operand, **Rd** = destination operand, **CND** = execution condition
Rs0,Rs1,Rd,CND Triadic registers conditional, **Rs0** = source operands 0, **Rs1** = source operand 1, **Rd** = destination operand, **CND** = execution condition

4.1.3 Registers and constants addressing modes

C12$_U$,Rs1,Rd,CND Constant and dual registers conditional, **C12$_U$** = source operand 0, **Rs1** = source operand 1, **Rd** = destination operand, **CND** = execution condition
C16$_U$,Rd,CND Constant and single register conditional, **C16$_U$** = source operand, **Rd** = destination operand, **CND** = execution condition
C16$_S$,Rd;CND Constant and single register conditional, **C16$_S$** = source operand, **Rd** = destination operand, **CND** = execution condition
C17$_S$,Rd;CND Constant and single register conditional, **C17$_S$** = source operand, **Rd** = destination operand, **CND** = execution condition
C17$_S$,Rs1 Constant and single register, **C17$_S$** = source operand 0, **Rs1** = source operand 1
SHC5$_U$,Rs1,Rd,CND Constant and dual registers conditional, **SHC5$_U$** = source operand 0, **Rs1** = source operand 1, **Rd** = destination operand, **CND** = execution condition
BTI5$_U$,Rs1,Rd,CND Constant and dual registers conditional, **BTI5$_U$** = source operand 0, **Rs1** = source operand 1, **Rd** = destination operand, **CND** = execution condition
C16$_U$,Rs1,Rd Constant and dual registers, **C16$_U$** = source operand 0, **Rs1** = source operand 1, **Rd** = destination operand
C32$_U$,Rs1,Rd Constant and dual registers, **C32$_U$** = source operand 0, **Rs1** = source operand 1, **Rd** = destination operand
C16$_S$,Rs1,Rd Constant and dual registers, **C16$_S$** = source operand 0, **Rs1** = source operand 1, **Rd** = destination operand

4.1.4 Instruction memory addressing modes

IA29$_U$ 29-bit absolute instruction address, **eia** = **IA29$_U$**
IO16$_S$ 16-bit signed instruction address offset, **eia** = current instruction address + **IO16$_S$**
IO16$_S$,S 16-bit signed instruction address offset with speculation, **eia** = current instruction address + **IO16$_S$**, the speculation type **S** determines if the branch speculation is for condition true or condition false

4.1.5 Miscellaneous addressing modes

implied operands are implicitly defined, there are two instruction categories: the first category (interrupt enable) uses flags of special register **CS** as source and destination operands; for the second category (`jump`, `jpsr`) **eia** = **TA**.

4.2 Instructions

This section is a summary of the base ISA instructions divided into functional groups. For each group the contained instructions are listed followed by a table with the available addressing modes. Instruction lists have the instruction mnemonic (used in assembly language) on the left side followed by a brief, single line description. In these descriptions the term “register” stands for a register of the **Rn** group.

In the addressing mode tables cells with available addressing modes are marked with an X and cells with non-available combinations of instructions and addressing modes are grayed out. Groups containing instructions that update the condition code flags have an additional row at the bottom of the table. Instructions that update the condition flags in the condition code register **CC** are marked with a ‘*’ in this row.

4.2.1 Load, Store

`ldbz`	load byte (8-bit word) from memory and zero-extend to 32 bits
`ldbs`	load byte (8-bit word) from memory and sign-extend to 32 bits
`ldsz`	load short (16-bit word) from memory and zero-extend to 32 bits
`ldss`	load short (16-bit word) from memory and sign-extend to 32 bits
`ldlg`	load long (32-bit word) from memory
`stbt`	store byte (8-bit) to memory
`stsh`	store short (16-bit) to memory
`stlg`	store long (32-bit) to memory

	ldbz	ldbs	ldsz	ldss	ldlg	stbt	stsh	stlg
DA16$_S$,CND	X	X	X	X	X	X	X	X
(DO12$_S$,An),CND	X	X	X	X	X	X	X	X
(Rx,An),CND	X	X	X	X	X	X	X	X
(An,AU12$_S$)*,CND	X	X	X	X	X	X	X	X
(An,Ru)*,CND	X	X	X	X	X	X	X	X
(An)+	X	X	X	X	X	X	X	X
-(An)	X	X	X	X	X	X	X	X

4.2.2 Move

`move`	move register to register or constant to register
`mfdp`	move from debug port to general purpose register
`mtdp`	move to debug port from general purpose register

	move	mfdp	mtdp
Rs,Rd,CND	X		
C17$_S$,Rd,CND	X		
Rd		X	
Rs			X

4.2.3 Arithmetic, excluding multiplies

`addt`	add register to register or constant to register
`addc`	add with carry register to register or constant to register
`addh`	add 16-bit constant to 16 MSBs of register
`subf`	subtract register from register or constant from register
`subc`	subtract with carry register from register or constant from register
`comp`	compare register to register or constant to register
`cmpc`	compare with carry register to register or constant to register
`negt`	negate (2’s complement) from register to register

absl absolute value (2’s complement if negative, move else) from register to register

clzr count leading zeros from register to register

	addt	addc	addh	subf	subc	comp	cmpc	negt	absl	clzr
Rs0,Rs1,Rd,CND	X	X		X	X					
C16$_U$,Rs1,Rd	X	X		X	X					
C32$_U$,Rs1,Rd			X							
C12$_U$,Rs1,Rd,CND	X	X		X	X					
Rs0,Rs1						X	X			
C17$_S$,Rs1						X	X			
Rs,Rd,CND								X	X	X
CC update	*	*		*	*	*	*			

4.2.4 Multiplies

mult multiply registers * register, 32*32 -> 64-bit, stores 32-bit low word result

mlhu multiply high unsigned, 32*32 -> 64-bit, stores 32-bit high word result

mlhs multiply high signed, 32*32 -> 64-bit, stores 32-bit high word result

mlcu multiply constant unsigned, 16*32 -> 48-bit, stores 32-bit low word result

mlcs multiply constant signed, 16*32 -> 48-bit, stores 32-bit low word result

	mult	mlhu	mlhs	mlcu	mlcs
Rs0,Rs1,Rd,CND	X	X	X		
C16$_U$,Rs1,Rd				X	
C16$_S$,Rs1,Rd					X

4.2.5 Logic

andb and bit wise of two registers or of constant and register

iorb inclusive or bit wise of two registers or of constant and register

xorb exclusive or bit wise of two registers or of constant and register

invt invert (1’s complement, invert) from register to register

	andb	iorb	xorb	invt
Rs0,Rs1,Rd,CND	X	X	X	
C16$_U$,Rs1,Rd	X	X	X	
Rs,Rd,CND				X
CC update	*			

4.2.6 Shift

shlz shift left with zero fill, constant or indirect shift count from 0 to 31

shlf shift left with feedback (rotate), constant or indirect shift count from 0 to 31

shru shift right unsigned, constant or indirect shift count from 0 to 31

shrs shift right signed, constant or indirect shift count from 0 to 31

	shlz	shlf	shru	shrs
SHC5$_U$,Rs1,Rd,CND	X	X	X	X
Rs0,Rs1,Rd,CND	X	X	X	X

4.2.7 Bit manipulation

btst bit set, constant or indirect bit index from 0 to 31

btcl bit clear, constant or indirect bit index from 0 to 31

bttg bit toggle, constant or indirect bit index from 0 to 31

btts bit test, constant or indirect bit index from 0 to 31

	btst	btcl	bttg	btts
BIT5$_U$,Rs1,Rd,CND	X	X	X	
BTI5$_U$,Rs,CND				X
Rs0,Rs1,Rd,CND	X	X	X	
Rs0,Rs1,CND				X
CC update				*

4.2.8 Endianess Conversion

ibos invert byte order short, swaps byte 0 and 1, bits[31:16] unchanged

ibol invert byte order long, inverts byte order from 3,2,1,0 to 0,1,2,3

	ibos	ibol
Rs,Rd,CND	X	X

4.2.9 Flow control

jump jump, continue program execution at specified target address

jpsr jump to subroutine

rtsr return from subroutine

rtir return from interrupt

brlc decrement loop counter and branch if non-zero

brxx branch conditional, 14 conditions, xx is a placeholder for the 2-character condition

stie set interrupt enable

clie clear interrupt enable

scie save and clear interrupt enable

rsie restore interrupt enable

	jump	jpsr	rtsr	rtir	brlc	brxx	stie	clie	scie	rsie
implied	X	X	X	X			X	X	X	X
IA29$_U$/IA22$_U$	X	X								
IO16$_S$					X					
IO16$_S$,S						X				

4.2.10 Miscellaneous

svpc save program counter to debug port

rspc restore program counter from debug port

stop stop, enter debug mode

	svpc	rspc	stop
implied	X	X	X

5 Operation modes

5.1 System and application operation modes

To support secure systems sf32 processors have two operation modes, the **som** (System Operation mode) and the **aom** (Application Operation Mode). In the **som** all processor resources are available and all instructions can be executed without restrictions. In the **aom** instructions with **CS** (Control and Status register) or **IA** (Interrupt return Address) as destination register are illegal. Attempts to execute an illegal instruction cause a security exception (**I0** interrupt).

The **som** mode is entered when the processor starts an interrupt service routine and the **IR** flag in register **CS** is set. Whenever the **IR** flag is cleared the processor is in **aom** mode. After reset the processor starts in the **som** mode with **CS.IR** set.

Both the instruction and the data bus interfaces include output signals that indicate if an access is a **som** access or an **aom** access. In secure systems these signal are used to protect critical system resources from access by application programs. In typical secure systems only operating system routines and certain device driver routines are executed in the **som** mode. Application programs are executed in the **aom** mode.

5.2 Reset

5.2.1 Program start address

The processor input signal **IRN**[3:0] and the output signal **IA**[29:0] determine the program start address in the instruction address space after a reset. The 4-bit interrupt number input signal **IRN**[3:0] is inserted as the four most significant bits of the instruction address **IA**[29:0] of the first instruction fetch after a reset. All other bits of **IA**[29:0] are zero. In summery the instruction address **IA**[29:0] of the first instruction fetch after a reset is **IA**[29:26] = **IRN**[3:0], **IA**[25:0] = 0.

This concept enables start addresses other than zero. While the processor's reset input is asserted external logic drives the **IRN**[3:0] input to the value of the desired start address. In most systems the instruction RAM starts at address zero. Driving **IRN**[3:0] to a non-zero value can be used to divert the program start after reset e.g. to a boot ROM.

5.2.2 Processor state

After a reset the following registers and register fields of the programming model have a defined state:

CS.IR = 1, the processor starts in an interrupt routine and in the **som** operation mode.

CS.IE = 0, maskable interrupts are disabled

CS.IS = 0, the interrupt enable save bit is clear

CC = 0, the condition code flags are all cleared

All other registers and register fields of the programming model are not defined after a reset. Their states and content after a reset is implementation specific. Software should not rely on any specific values.

5.3 Interrupts

5.3.1 Overview

The sf32 processors have 16 interrupts named **I0**, **I1**, **I2** and **I15**. Interrupt requests are acknowledged only if the **IE** bit in register **CS** is set. External logic generates interrupt requests by asserting the processor's interrupt request input signal **IRQ** and driving the number of the requested interrupt on the processor's 4-bit interrupt number input **IRN**[3:0]. The processor acknowledges an interrupt request by asserting the **IACK** output.

Each of the 16 interrupts has an associated start address in the instruction address space. These start addresses are software programmable and are contained in the interrupt vector table which is mapped into a 64 bytes window of the processor's data address space. The 26-bit field **IVTP** of special register **CS** defines the start address of the table. **IVTP** defines the higher 26 bits of the 32-bit table address. The six least significant bits of the table address are zero. This implies that the interrupt vector table is aligned on a 64 bytes boundary. The table contains 16 entries of 32-bit size. Each entry is a 32-bit instruction address. Because sf32 instruction addresses must be aligned on 32-bit boundaries (2 LSBs zero) the two LSBs of interrupt vector table entries are ignored.

When an interrupt is started the instruction address of the next instruction following the last instruction executed before the interrupt is stored in register **IA**. The state of the **CC** register is stored in a hidden register (not visible in the programming model). When an **`rtir`** (return from interrupt) instruction is executed at the end of an interrupt service routine the condition code flags are restored from this hidden register and program execution continues at the instruction address in **IA**.

Writing to register **IA** is required at least after a processor reset to start program execution at a defined address when leaving the interrupt state with an **`rtir`** instruction. Some OS code may require to read and write the **IA** register to save, redirect and restore interrupt return addresses in cases of task switches and system calls.

Beside **CC** and the instruction address the sf32 does not save any registers of the programming model automatically. User program code must save and restore any other registers that are modified by an interrupt service routine.

In the sf32 ISA interrupt **I0** is used for security violation exceptions and should not be triggered by hardware.

5.3.2 Interrupt Flow

An interrupt request is generated when external logic asserts the processor's input signal **IRQ**. The 4-bit interrupt number input signal **IRN**[3:0] determines the number of the requested interrupt from **I0** – **I15**. The request is acknowledged immediately if the processor is not already executing another interrupt service routine (**CS**.**IR** clear). If the processor is already executing an interrupt service routine (**CS**.**IR** set) the request is acknowledged when the processor has returned from this routine and **CS**.**IR** has been cleared.

After the request has been acknowledged the processor reads the start address of the interrupt service routine from the interrupt vector table in the data address space. Before executing the first instruction of the service routine the address of the next instruction of the interrupted code sequence is stored in **IA** and the condition code flags in **CC** are saved in a hidden register. While executing instructions of the interrupt service routine the processor is in the **som** mode and **CS**.**IR** is set. When an **`rtir`** (return from interrupt) instruction is executed at the end of the interrupt service routine the condition code flags and the instruction address are restored and execution of the interrupted code sequence continues.

5.3.3 Security Exceptions

A security exception is generated when the processor is in the **aom** mode and an instruction attempts to modify the **IA** or **CS** register. Instructions that cause a security exception are not executed means their destination operands are not updated.

A security exception is an internally generated **I0** interrupt. Although it is not prohibited to request **I0** interrupts by asserting **IRQ** with **IRN**[3:0] = 0 it is recommended to reserve **I0** interrupts for exceptions.

Reason for protecting **IA** and **CS** in the **AOM** mode is to prevent application software from modifying the interrupt return address or **IVTP** vector table pointer. Redirecting interrupt start addresses or return addresses would enable application software to gain access to protected system resources. In secure systems the interrupt vector table must be stored in a protected memory area that can only be accessed in **som** mode.

5.4 Debug Support

5.4.1 Overview

The processors of the sf32 family have a scalable debug concept. To enable very low cost implementations most of the debug resources are outside the processor core in a separate module. The functionality of this module can be adapted to the requirements of each use case to avoid redundant resources. The processor provides a 32-bit port to connect to the debug module.

To use any debug functions the processor has to be in the **stopped** state. This state is entered by either driving the **STRQ** input signal to the asserted state or by executing a **`stop`** instruction. After all pending instructions are retired the processor indicates it has reached the **stopped** state by asserting the **STPD** output signal. While in the **stopped** state the debug port together with a set of dedicated instructions provide the following low level functions:

- Transfer the content of a register **Rn** to the debug output port
- Transfer a 32-bit value from the debug input port to a register **Rn**
- Transfer the program counter value to the debug output port
- Transfer a 32-bit value from the debug input port to the program counter
- Inject individual instructions via the debug input port and execute them

The debug module must provide the following mandatory and may provide following optional functions:

- Mandatory: communication link to the debug host (PC), e.g. JTAG, UART, USB, Ethernet
- Mandatory: state machine to handle the control signals of the debug port
- Mandatory: a mechanism to transfer 32-bit data words from the processor's debug output port to the debug host and from the debug host to the processor's debug input port
- Mandatory: assert and release the processor's reset input
- Optional: instruction breakpoint register(s)
- Optional: data breakpoint and watch point register(s)
- Optional: access to the processor's instruction memory
- Optional: access to the processor's data memory
- Optional: trace buffer(s)

5.4.2 Debug Port

The debug port consists of the following signals:

DBI[31:0]	Debug In, 32-bit data input
DBO[31:0]	Debug Out, 32-bit data output
STRQ	Stop Request, 1-bit control input
INJI	Inject Instruction, 1-bit control input
STPD	Stopped, 1-bit control output

5.4.3 Debug Instructions

The following dedicated instructions are part of the sf32 debug concept:

mtdp move to debug port, transfers the content of a registers **Rn** to the debug port data output

mfdp move from debug port, transfers the 32-bit value driven on the debug port data input to a register **Rn**

svpc save program counter, transfers the instruction address of the last instruction executed before the **stopped** state was entered to the debug port data output

rspc restore program counter, transfers the 32-bit value driven on the debug port data input to an internal instruction address register. When the processor leaves the **stopped** state program execution continues from this address. As with all sf32 instruction addresses the two LSBs of the 32-bit value are ignored.

stop stop, the processor stops fetching new instructions and enters the **stopped** state when all pending instructions are retired.

5.4.4 Debug Procedures

The following paragraphs describe how the most common debug procedures are implemented and how the functionality is split between the debug module and the processor.

5.4.4.1 Instruction breakpoints

The instruction that should cause the break point is replaced with a **stop** instruction. Executing a **stop** instruction causes the processor to enter the **stopped** mode. There are multiple options of how to replace an instruction of a program with a **stop** instruction.

The simplest option requires that the processor can access the instruction memory via the data bus (instruction memory mapped into the data address space). In this case the debug module can inject an instruction sequence into the processor that writes a **stop** instruction at the desired location of the instruction memory.

In systems where the processor cannot access the instruction memory via the data bus two options exist to generate instruction break points. The first option requires that the debug module has direct access to the processor's instruction memory. In this case the debug module writes **stop** instructions directly into the desired locations of the instruction memory. The second option requires one or more address registers in the debug module and the debug module must be connected to the processor's instruction memory controller. The debug module monitors the processor's instruction bus and compares instruction fetch addresses to the values in the address registers. In case of a match the instruction word read from the instruction memory is replaced on the fly by a **stop** instruction opcode. This option also works for read only instruction memories.

Once an instruction break point has been hit the debug module has to wait until the processor asserts the **STPD** output signal. Then the debug host can access the processor's registers and data memory by injecting instruction sequences via the debug module. To continue normal processor operation the debug module has to assert and then de-assert the **STRQ** signal while the **STPD** output is asserted.

5.4.4.2 Data breakpoints and watch points

Data break points and watch points require a set of registers in the debug module and a connection of the debug module to the processor's data bus. Typical entries have a least a data address register. With optional data value and address/data mask registers a break/watch point becomes more flexible and can also trigger on a data value or address range.

The debug module monitors the processor's data bus and compares data address and data in/out values to the registers of the break/watch point entries. In case of a match a watch point only signals the event to the debug host. In case of a break point hit the debug module brings the processor in the **stopped** mode by asserting the processor's **STRQ** input.

5.4.4.3 Show register content

When the processor is in the **stopped** mode the debug module injects **`mtdp`** instructions to read the content of processor registers.

5.4.4.4 Modify register content

When the processor is in the **stopped** mode the debug module injects **`mfdp`** instructions to change the content of processor registers.

5.4.4.5 Show memory content

For memories that the processor can access through the data bus the desired data word is first read into a general purpose register by injecting a load instruction. Then the general purpose register is read by injecting an **`mtdp`** instruction.

To read from memories that are not mapped into the processor's data address space the debug module requires a direct connection to these memories.

5.4.4.6 Modify memory content

For memories that the processor can access via the data bus the desired data word is first written into a general purpose register by injecting an **`mfdp`** instruction. Then the general purpose register is written into memory by injecting a store instruction.

To write to memories that are not mapped into the processor's data address space the debug module requires a direct connection to these memories.

5.4.4.7 Download and start a program

Data and program code is written into the processor's data and instruction memories using the previously described procedures. To start a program at a certain address in the instruction address space the debug module injects an **`rspc`** instruction and drives the desired address on the debug port input. The debug module then de-asserts the **STRQ** signal. The processor leaves the **stopped** state and starts program execution from the injected address.

5.4.4.8 Saving and restoring the program counter

When the processor has been brought into the **stopped** state to access registers and/or memories by injecting individual instructions via the debug module it is not necessary to save and restore the program counter. The **`rspc`** instruction is used to start programs from a defined location as described previously.

Combinations of **`svpc`** and **`rspc`** instructions are used to execute debug utility routines as part of a system's debug concept. Injecting longer instruction sequences while the processor is in stopped mode, e.g. to copy memory areas can be slow because of the instruction injection process. For each injected instruction the processor's pipeline is flushed and the next instruction can be injected only when the processor has reasserted the **STPD** output. A more efficient method is to store some debug utility routines in a reserved area of the processor's instruction memory space.

To execute a debug utility routine for the debugging of an application program the processor is first brought into the **stopped** state. Then the program counter is saved by injecting a **`svpc`** instruction. The value is the address where the application program has been stopped. It is stored for later use in the debug host or in a register of the debug module. The start address of the debug utility routine is set by injecting an **`rspc`** instruction and driving the start address on the processor's debug port input. The debug module then releases the **`STRQ`** input signal the processor leaves the **stopped** state and executes the debug utility routine. The last instruction of the debug utility routine is a **`stop`** instruction which brings the processor back into the **stopped** state. To continue the application program at the same location it has been stopped an **`rspc`** instruction is injected and the previously saved instruction address is driven on the processor's debug port input. Then the **STRQ** input is released and the processor continues executing the application program.

6 Operand Types

6.1 Legend

This chapter defines the bit accurate generation and calculation of individual operands of instructions. For constant and register data operands the generation of the operand value will be defined. For memory operands and instruction words the generation or calculation of the effective memory address will be defined.

The number and types of the operands of each instruction (also called addressing modes) are not defined here. They are defined in the addressing mode table of each instruction in the instruction details chapters.

The following paragraphs define the formats and notations used in operand type definitions and effective address calculations.

6.1.1 Mnemonic

This is the acronym of the operand type used to specify operands in the addressing mode tables of detailed instruction descriptions.

Mnemonics of constants with a value range have a one-character suffix with the following meaning:
U (Unsigned) or **S** (Signed)

6.1.2 Text Description

Text description of how the operand is generated or calculated. Also lists the instructions for which the operand type is used. Text descriptions reference the variables used in the C language description

6.1.3 C language description

Pseudo C language statements are used as bit true reference of how the operand is generated or calculated. The statements use the following data types and notations:

uint5 type: 5-bit unsigned integer

uint32 type: 32-bit unsigned integer

boolean type: 1-bit Boolean variable, can take the values **true** and **false** or **1** and **0**.

sizeof(memory operand), this operator yields the size of a memory operand in bytes and takes values of 1 for byte (8-bit) operands, 2 for short (16-bit) operands, 4 for long (32-bit) operands, n for byte **RGS** (register selection) operands, 2*n for short **RGS** (register selection) operands and 4*n for long **RGS** (register selection) operands where n is the number of registers in **RGS**.

6.1.4 Opcode

This table defines where the bits of the operand are located in 32-bit opcode words. For each bit that is part of the operand the bit position within the operand type's bit array is specified. Bits that are not part of the operand are empty boxes in white color.

Some operand types have multiple coding options. The opcode tables have separate rows for each coding option.

6.2 Constant operand types

Constant operands are bit fields in instruction opcodes that are directly transformed into source operands of instructions.

C12$_U$ **12-bit constant (Unsigned)**

The 12-bit field **C12$_U$** is extracted from the 32- opcode word and zero-extended to the 32-bit source operand **src**. The value range is [0,4095].

Used with instructions **addt, subf, addc, subc**

C language description

```
uint32 src;
src = C12_U;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C12$_U$					11	10	9	8	7	6	5	4	3	2	1	0																

C16$_U$ 16-bit constant (Unsigned)

The 16-bit field **C16$_U$** is extracted from the opcode word and zero-extended to the 32-bit source operand **src**. The value range is [0,65535].

Used with instructions: **addt, subf, andb, iorb, xorb, mlcu.**

C language description

uint32 src;

src = C16$_U$;

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C16$_U$	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																

C32$_U$ 32-bit constant

The 16-bit field **C32$_U$** is extracted from the opcode word and becomes the 32-bit source operand **src**. The value range is [0x00000000,0xFFFF0000]. Bits [15:0] of the constant are always zero and are not coded.

Used with instruction: **addh**

C language description

uint32 src;

src = C32$_U$;

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C32$_U$	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16																

C16$_S$ 16-bit constant (Signed)

The 16-bit field **C16$_S$** is extracted from the opcode word and sign-extended to the 32-bit source operand **src**. The value range is [-32768,32767].

Used with the instructions: **mlcs**

C language description

uint32 src;

src = C16$_S$ & 0x8000 ? C16$_S$ | 0xFFFF0000 : C16$_S$;

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C16$_S$	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																

C17$_S$ 17-bit constant (Signed)

The 17-bit field **C17$_S$** is extracted from the opcode word and sign-extended to the 32-bit source operand **src**. The value range is [-65536,65535].

Used with the instructions: **move, comp, cmpc**

C language description

uint32 src;

src = C17$_S$ & 0x10000 ? C17$_S$ | 0xFFFE0000 : C17$_S$;

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C17$_S$	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0													16			
					11	10	9	8	7	6	5	4	3	2	1	0							15	14	13	12			16			

SHC5$_U$ 5-bit shift count (Unsigned)

The 5-bit field **SHC5$_U$** is extracted from the 32-bit opcode and becomes the source operand **src**. The value range is [0,31].

Used with instructions: **shlz, shlf, shru, shrs**

C language description

uint5 src;

src = SHC5$_U$;

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHC5$_U$												4	3	2	1	0																

BTI5$_U$

5-bit bit index (Unsigned)

The 5-bit field **BTI5$_U$** is extracted from the 32-bit opcode and becomes the source operand **src**. The value range is [0,31]. The bit index is counted from the LSB (**BTI5$_U$** = 0) to the MSB (**BTI5$_U$** = 31). The bit index operand is used to address individual bits of registers **Rn**.

Used with instructions: **btst, btcl, bttg, btts**

C language description

```
uint5 src;
src = BTI5U;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BTI5$_U$												4	3	2	1	0																

6.3 Register operand types

Register operands are contained in one of the 32 registers **Rn**. They can be either source or destination operands. Bit fields in the instruction opcode determine which register of the **Rn** group or **An** sub-group is used. Reserved register bits and bit-fields read as zeros.

Rs

Rn register used as source

The content of register **Rs** is the 32-bit source operand **src**. The **Rs** operand type is used with instructions that have a single source operand.

C language description

```
uint32 src;
src = Rs;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs												4	3	2	1	0																

Rs0

Rn register used as source 0

The content of register **Rs0** is the 32-bit source operand **src0**. The **Rs0** operand type is used with instructions that have two source operands. If used with non-commutative instructions like subtract or compare **Rs0** is on the right side of the operator (source1 – **Rs0**). If used with shift, bit manipulation or bit-field instructions **Rs0** is the parameter source operand and contains the indirect shift-count or bit-index.

C language description

```
uint32 src0;
src0 = Rs0;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs0												4	3	2	1	0																

Rs1

Rn register used as source 1

The content of register **Rs1** is the 32-bit source operand **src1**. The **Rs1** operand type is used with instructions that have two source operands. If used with non-commutative instructions like subtract or compare **Rs1** is on the left side of the operator (**Rs1** – source0). If used with shift, bit manipulation or bit-field instructions **Rs1** is the data source operand.

C language description

```
uint32 src1;
src1 = Rs1;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs1																						4	3	2	1	0						

Rd

`Rn register used as destination`

The 32-bit destination operand **`dst`** is stored in register **Rd**. If the register number is 26 (**CS** register) only bits [31:6] are updated (**CS**.**IVTP**). Bits [5:3] (hard wired to zero) and bits [2:0] (flags) remain unchanged.

C language description

```
uint32 dst;
Rd = dst;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd																	4	3	2	1	0											

RGS

`Register Selection`

RGS is a selection of registers **Rn**. Up to 19 registers can be selected by 19 flags in the 32-bit opcode. Only registers **R0**-**RB**, **RP**-**RV**, **LC**, **TA** and **SA** can be contained in a register selection **RGS**. The register selection **RGS** is either the source operand **`src[n-1:0]`** of a memory store instruction or the destination operand **`dst[n-1:0]`**of a memory load instruction with addressing mode **`(An)+`** or **`-(An)`**. The **RGS** source or destination operand is an array of **n** 8-bit, 16-bit or 32-bit values where **n** is the number of registers selected by **RGS**. In memory the **n** values are located at adjacent address locations relative to their size which is the same for all **n** values (distance of 1 for bytes, 2 for shorts and 4 for longs). **RGS** register selections are primarily intended for efficient push-to-stack and pop-from-stack operations. Registers are stored to memory and loaded from memory in a fixed order which is reversed between the **`(An)+`** and **`-(An)`** addressing modes. Refer to the **`–(An)`** and **`(An)+`** memory addressing modes in the next section of this chapter for details.

Used with instructions **`ldbz, ldbs, ldsz, ldss, ldlg, stbt, stsh, stlg`**

C language description

```
uint32 src[n], dst[n];
if(instruction == (stbt|stsh|stlg))
  src[n-1:0] = RGS;
if(instruction == (ldbz|ldbs|ldsz|ldss|ldlg))
  RGS = dst[n-1:0];
```

The coding of **RGS** is different for the **`(An)+`** and **`-(An)`** addressing modes. The opcode table below has separate entries for **`(An)+`** and **`-(An)`**. Register flags are identified by single characters with the following notation:

- characters 0-B identify **R0** – **RB**
- characters P-V identify **RP** – **RV**
- character L identifies **LC**
- character T identifies **TA**
- character S identifies **SA**

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`(An)+`	S	T	L	0			1	2	3	4	5	6	7	8	9	A						B	P	Q	U	V						
`-(An)`	V	U	Q	P			B	A	9	8	7	6	5	4	3	2						1	0	L	T	S						

6.4 *Memory operand addressing*

Memory operands are 8-bit, 16-bit, 32-bit or n* 32-bit memory words used as source operand of load instructions or as destination operand of store instructions. Addressing modes for memory operands determine the 32-bit effective data address **eda** of the operand. Some of the indirect memory addressing modes that use an address register **An** to calculate **eda** update the address register **An** as a side effect. For addressing modes where the memory operand size determines the value of an address register **An** increment or decrement or a scale factor the increment values or scale factors are specified in the addressing mode table of the instruction description.

For addressing modes with an indirect address register **An** the opcode contains a 4-bit field that selects one of registers **R8** – **RZ** as indirect address.

DA16$_S$ **16-bit direct data address**

The effective address **eda** of the data memory operand is the 16-bit constant **DA16$_S$** (Signed) extracted from the opcode and sign-extended to 32 bits. Legal values for **eda** are from 0x00000000 – 0x00007FFF and from 0xFFFF8000 to 0xFFFFFFFF.

Used with instructions `ldbz, ldbs, ldsz, ldss, ldlg, stbt, stsh, stlg`

C language description

```
uint32 src,dst;
void *eda;
eda = DA16S & 0x8000 ? DA16S | 0xFFFF0000 : DA16S;
if(instruction == (ldbz|ldbs|ldsz|ldss|ldlg))
  dst = *eda;
if(instruction == (stbt|stsh|stlg))
  *eda = src;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DA16$_S$					11	10	9	8	7	6	5	4	3	2	1	0		15	14	13	12											

(DO12$_S$,An) **Address register indirect with 12-bit signed offset**

The 12-bit constant **DO12$_S$** (Signed) is extracted from the opcode. The effective address **eda** of the data memory operand is the constant **DO12$_S$** sign-extended to 32 bits and added to the value of the address register **An**. The **DO12$_S$** value range is [-2048,2047].

Used with instructions `ldbz, ldbs, ldsz, ldss, ldlg, stbt, stsh, stlg`

C language description

```
uint32 src,dst;
void *eda;
eda = An + (DO12S & 0x800 ? 0xFFFFFF00 | DO12S : DO12S);
if(instruction == (ldbz|ldbs|ldsz|ldss|ldlg))
  dst = *eda;
if(instruction == (stbt|stsh|stlg))
  *eda = src;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DO12$_S$					11	10	9	8	7	6	5	4	3	2	1	0																
An																		3	2	1	0											

(Rx,An) Address register indirect with index

The effective address **eda** of the data memory operand is the index register **Rx** multiplied by the operand size and added to the value of the address register **An**.

Used with instructions **`ldbz, ldbs, ldsz, ldss, ldlg, stbt, stsh, stlg`**

C language description

```
uint32 src,dst;
void *eda;
eda = An + sizeof(memory operand)*Rx;
if(instruction == (ldbz|ldbs|ldsz|ldss|ldlg))
  dst = *eda;
if(instruction == (stbt|stsh|stlg))
  *eda = src;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rx												4	3	2	1	0																
An																		3	2	1	0											

(An,AU12$_S$) Address register indirect with 12-bit direct update

The effective address **eda** of the data memory operand is the value of the address register **An**. After the operand access the 12-bit constant **AU12$_S$** (Signed) is extracted from the opcode, sign-extended to 32 bits and added to the address register **An**. The **AU12$_S$** value range is [-2048,2047].

Used with instructions **`ldbz, ldbs, ldsz, ldss, ldlg, stbt, stsh, stlg`**

C language description

```
uint32 src,dst;
void *eda;
eda = An;
if(instruction == (ldbz|ldbs|ldsz|ldss|ldlg))
  dst = *eda;
if(instruction == (stbt|stsh|stlg))
  *eda = src;
An += AU12S & 0x800 ? 0xFFFFF000 | AU12S : AU12S;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AU12$_S$					11	10	9	8	7	6	5	4	3	2	1	0																
An																		3	2	1	0											

(An,Ru)* Address register indirect with indirect update

The effective address **eda** of the data memory operand is the value of the address register **An**. After the operand access the value of the update register **Ru** is added to the address register **An**.

Used with instructions **`ldbz, ldbs, ldsz, ldss, ldlg, stbt, stsh, stlg`**

C language description

```
uint32 src,dst;
void *eda;
eda = An;
if(instruction == (ldbz|ldbs|ldsz|ldss|ldlg))
  dst = *eda;
if(instruction == (stbt|stsh|stlg))
  *eda = src;
An += Ru;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ru												4	3	2	1	0																
An																		3	2	1	0											

(An)+ Address register indirect with post-increment

This addressing mode is available only for **RGS** (register selection) destination operands. The effective address **eda** of the data memory operand is the value of the address register **An**. After each memory access the address register **An** is incremented by the **size** (in bytes) of the operand. Registers of the **RGS** selection are read-from/written-to memory in the following fixed order (reversed order of **-(An)** addressing mode):

SA, **TA**, **LC**, **R0**, **R1**, **R2**, **R3**, **R4**, **R5**, **R6**, **R7**, **R8**, **R9**, **RA**, **RB**, **RP**, **RQ**, **RU**, **RV**

Used with instructions **ldbz, ldbs, ldsz, ldss, ldlg, stbt, stsh, stlg**

C language description

```
uint32 dst[n];   // n = number of registers in RGS
void *eda;
int i;
eda = An;
for(i=0;i < n;i++){
  if(instruction == (ldbz|ldbs|ldsz|ldss|ldlg))
    dst = *eda;
  if(instruction == (stbt|stsh|stlg))
    *eda = src;
  eda += sizeof(dst[i]);
}
An = eda;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
An																		3	2	1	0											

-(An) Address register indirect with pre-decrement

This addressing mode is available only for **RGS** (register selection) source operands. Before each memory access the address register **An** is decremented by the **size** (in bytes) of the individual operand. The effective address **eda** of the data memory operand is the value of the address register **An** after the decrement. Registers of the **RGS** selection are read-rom/written-to memory in the following fixed order (reversed order of **(An)+** addressing mode):

RV, **RU**, **RQ**, **RP**, **RB**, **RA**, **R9**, **R8**, **R7**, **R6**, **R5**, **R4**, **R3**, **R2**, **R1**, **R0**, **LC**, **TA**, **SA**

Used with instructions **ldbz, ldbs, ldsz, ldss, ldlg, stbt, stsh, stlg**

C language description

```
uint32 src[n];   // n = number of registers in RGS
void *eda;
int i;
eda = An;
for(i=0;i < n;i++){
  eda -= sizeof(src[i]);
  if(instruction == (ldbz|ldbs|ldsz|ldss|ldlg))
    dst = *eda;
  if(instruction == (stbt|stsh|stlg))
    *eda = src;
}
An = eda;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
An																		3	2	1	0											

6.5 Condition operand

The condition **CND** is a 4-bit field in opcodes words of selected addressing modes. Instructions with a **CND** operand are executed only if the condition **CND** is `true`. If the condition **CND** is `false` the instruction performs no operations. There are 14 different conditions which are the same as the conditions tested by the 14 conditional branch instructions. If the 4-bit **CND** field has a value of 15 (all bits set) the instruction is always executed (unconditional) regardless of the state of the flags in register **CC**.

CND

Condition

The Boolean variable **TRUE** is calculated from the condition code flags in special register **CC** using one out of 15 formulas. The formula is determined by the 4-bit operand **CND** in the opcode.

C language description

```
uint4 CND;
boolean C,O,Z,N,TRUE;
C = CC.C;
O = CC.O;
Z = CC.Z;
N = CC.N;
switch(CND){
case  0: TRUE = ~C; break;                       // INC = If No Carry
case  1: TRUE =  C; break;                       // ICR = If Carry
case  2: TRUE = ~O; break;                       // INO = If No Overflow
case  3: TRUE =  O; break;                       // IOF = If Overflow
case  4: TRUE = ~Z; break;                       // INZ = If Non Zero
case  5: TRUE =  Z; break;                       // IZR = If Zero
case  6: TRUE = ~N; break;                       // IPS = If Positive
case  7: TRUE =  N; break;                       // ING = If Negative
case  8: TRUE =  C |  Z; break;                  // ILS = If Lower or Same
case  9: TRUE = ~C & ~Z; break;                  // IHI = If Higher
case 10: TRUE = (N&~O) | (~N& O); break;         // ILO = If Lower
case 11: TRUE = (N& O) | (~N&~O); break;         // IGE = If Greater or Equal
case 12: TRUE =  Z | (N&~O) | (~N& O); break;    // ILE = If Lower or Equal
case 13: TRUE = ~Z & ((C& O) | (~N&~O)); break;  // IGT = If Greater
case 15: TRUE = 1; break;                        // ALW = Always
}
if(TRUE)
  execute instruction;
else
  no operation;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CND	3	2	1	0																												

6.6 Instruction addressing

Instruction addresses point to 32-bit opcode words in the instruction memory. Instruction addresses must be aligned on 32-bit boundaries, the two LSBs are always zero. With the exception of some flow instructions the effective instruction address **eia** of the next instruction is the address of the current instruction plus four.

C language description

```
uint32 *eia;
eia[next instruction] = eia[current instruction] + 4;
```

Some of the flow instructions calculate a new effective instruction address **eia** and instruction execution continues non-sequentially at the new location in the instruction memory. The following paragraphs define how these flow instructions generate the new effective instruction address **eia**.

IA29$_U$ 29-bit absolute instruction address (Unsigned)

The 27-bit field **IA29**$_U$ (Unsigned) is extracted from the opcodes word. The two LSBs of **IA29**$_U$ are not coded and are always zero. **IA29**$_U$ is zero-extended to the 32-bit effective instruction address **eia**. The **IA29**$_U$ value range is [0,0x1FFFFFFC]. An instruction address space of 512 Mbytes can be reached with the **IA29**$_U$ absolute address.

Used with instructions **jump, jpsr**

C language description

```
uint32 *eia;
eia = IA29_U;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IA29$_U$	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2					

IO16$_S$ 16-bit instruction offset (Signed)

The 14-bit field **IO16**$_S$ is extracted from the opcodes word. The two LSBs of **IO16**$_S$ are not coded and are always zero. The new effective instruction address **eia** is the sign extended constant added to the address of the current instruction **cia.**

Used with instructions **brlc, brxx**

C language description

```
uint32 *eia,*cia;
eia = cia + (IO16_S & 0x8000 ? 0xFFFF0000 | IO16_S : IO16_S);
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IO16$_S$									15	14	13	12	11	10	9	8	7	6	5	4	3	2										

S Speculation

The 1-bit flag **S** is extracted from the opcodes word. Processor implementations may use the **S** flag to determine the speculation type (branch taken of branch not taken) of conditional branches in situations where a branch condition is not evaluated yet by the time a branch instruction is decoded. Using the flag can improve the performance (#of effective execution cycles) of conditional branches with a preferred condition evaluation result that is known at compile time. The setting of the **S** flag has no impact on any destination operands. It provides an optional tool for performance improvement of processor implementations.

Used with instructions **brxx**

C language description

```
boolean s;
s = S;
```

opcode bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S																							S									

7 Load, store and move instructions

7.1 Common properties

The load, store and move instructions transfer the source operand to the destination operand without modifying the value of the operand. Except the load/store instructions with **RGS** source or destination operands all load, store and move instructions have a single source operand and a single destination operand. Move instructions have a constant or register source and a register destination. Load instructions have a memory source and a register destination. Store instructions have a register source and a memory destination. None of the load, store and move instructions update the condition code flags in register **CC**.

For load from memory instructions the destination register **`Rd`** cannot be one of the **CC**, **CS**, **IA** or **ID** registers. For store to memory instructions the source register **`Rs`** cannot be one of the **CC** or **CS** registers.

7.2 Legend

The next section lists the load, store and move instructions in alphabetical order and defines the bit accurate operations they perform. The following paragraphs define the formats and notations used in individual instruction definitions.

7.2.1 Mnemonic

A four-character acronym of the instruction used to specify instructions in assembly language source code.

7.2.2 Text Description

Text description of the operations performed. Text descriptions reference the operand variables that are defined and used in the C language description

7.2.3 C language description

These C language statements are the bit true reference of the operations performed by an instruction. The following types and variables are used in the statements:

`uint32`	type: 32-bit unsigned integer
`uint16`	type: 16-bit unsigned integer
`uint8`	type: 8-bit unsigned integer
`boolean`	type: 1-bit Boolean variable, can take the values **`true`** and **`false`** or **`1`** and **`0`**.

The use of unsigned integers does not necessary mean that the underlying operands are unsigned. It means that the computations defined by the C statements are done assuming unsigned operands.

7.2.4 Addressing modes table

This table lists all addressing modes of the instruction. For each addressing mode the assembly language format is specified and the assignment of operands used in the C statements to operand specifiers in the assembly format is given.

For the **`(An)+`** and **`-(An)`** addressing modes with **`RGS`** source or destination operand the **`eda`** (effective data address) column uses variable **`i`** to reference the $\mathtt{i}_{\mathtt{th}}$ element of the **`RGS`** register selection. Variable **`i`** is running from **`0`** to **`n-1`** where **`n`** is the number of registers contained in **`RGS`**.

7.2.5 Notes

Notes are optional and provide hints of how the instruction is used or if other instructions can do similar operations more efficiently.

7.3 Instruction details

ldbs — load byte and sign-extend

If the condition **CND** is true loads the byte (8-bit word) from the effective data address **eda** in the data memory, sign-extends the value to 32 bits and stores it in the 32-bit destination **dst**. If the condition **cnd** is false the instruction performs no operations. For addressing modes with no **CND** parameter the variable **cnd** is always **true**. Some addressing modes update the indirect address register **An** as indicated in the addressing modes table. With the **(An)+,RGS** and **-(An),RGS** addressing modes the update parameter **n** is the number of registers contained in **RGS** and can take values from 1 to 19.

C language description

```
uint32 dst;
uint8 *eda;
boolean cnd;
if(cnd == true)
  dst = *eda & 0x80 ? *eda | 0xFFFFFF00 : *eda;
```

The C language statements for the calculation of the effective data address **eda** and for the **An** update operations are specified in the addressing modes table for each addressing mode.

Addressing Modes	assembly format	eda	An update	dst	cnd
direct 16-bit data address	ldbs DA16$_S$,Rd,CND	DA16$_S$	not appl.	Rd	CND
indirect data address with direct update	ldbs (An,AU12$_S$)*,Rd,CND	An	+= AU12$_S$	Rd	CND
indirect data address with indirect update	ldbs (An,Ru)*,Rd,CND	An	+= Ru	Rd	CND
indirect data address with 12-bit offset	ldbs (DO12$_S$,An),Rd,CND	An+DO12$_S$	no update	Rd	CND
indirect data address with index	ldbs (Rx,An),Rd,CND	An+Rx	no update	Rd	CND
indirect data address with post-increment	ldbs (An)+,RGS	An+i	+= n	RGS	true
indirect data address with pre-decrement	ldbs -(An),RGS	An-i-1	-= n	RGS	true

ldbz — load byte and zero-extend

If the condition **CND** is true loads the byte (8-bit word) from the effective data address **eda** in the data memory, zero-extends the value to 32 bits and stores it in the 32-bit destination **dst**. If the condition **cnd** is false the instruction performs no operations. For addressing modes with no **CND** parameter the variable **cnd** is always **true**. Some addressing modes update the indirect address register **An** as indicated in the addressing modes table. With the **(An)+,RGS** and **-(An),RGS** addressing modes the update parameter **n** is the number of registers contained in **RGS** and can take values from 1 to 19.

C language description

```
uint32 dst;
uint8 *eda;
boolean cnd;
if(cnd == true)
  dst = *eda;
```

The C language statements for the calculation of the effective data address **eda** and for the **An** update operations are specified in the addressing modes table for each addressing mode.

Addressing Modes	assembly format	eda	An update	dst	cnd
direct 16-bit data address	ldbz DA16$_S$,Rd,CND	DA16$_S$	not appl.	Rd	CND
indirect data address with direct update	ldbz (An,AU12$_S$)*,Rd,CND	An	+= AU12$_S$	Rd	CND
indirect data address with indirect update	ldbz (An,Ru)*,Rd,CND	An	+= Ru	Rd	CND
indirect data address with 12-bit offset	ldbz (DO12$_S$,An),Rd,CND	An+DO12$_S$	no update	Rd	CND
indirect data address with index	ldbz (Rx,An),Rd,CND	An+Rx	no update	Rd	CND
indirect data address with post-increment	ldbz (An)+,RGS	An+i	+= n	RGS	true
indirect data address with pre-decrement	ldbz -(An),RGS	An-i-1	-= n	RGS	true

ldlg

load long

If the condition **CND** is true loads the 32-bit word from the effective data address **eda** in the data memory and stores it in the 32-bit destination **dst**. If the condition **cnd** is false the instruction performs no operations. For addressing modes with no **CND** parameter the variable **cnd** is always **true**. Some addressing modes update the indirect address register **An** as indicated in the addressing modes table. With the **(An)+,RGS** and **-(An),RGS** addressing modes the update parameter **n** is the number of registers contained in **RGS** and can take values from 1 to 19.

C language description

```
uint32 dst,*eda;
boolean cnd;
if(cnd == true)
  dst = *eda;
```

The C language statements for the calculation of the effective data address **eda** and for the **An** update operations are specified in the addressing modes table for each addressing mode.

Addressing Modes	assembly format	eda	An update	dst	cnd
direct 16-bit data address	ldlg DA16$_S$,Rd,CND	DA16$_S$	not appl.	Rd	CND
indirect data address with direct update	ldlg (An,AU12$_S$)*,Rd,CND	An	+= AU12$_S$	Rd	CND
indirect data address with indirect update	ldlg (An,Ru)*,Rd,CND	An	+= Ru	Rd	CND
indirect data address with 12-bit offset	ldlg (DO12$_S$,An),Rd,CND	An+DO12$_S$	no update	Rd	CND
indirect data address with index	ldlg (Rx,An),Rd,CND	An+4*Rx	no update	Rd	CND
indirect data address with post-increment	ldlg (An)+,RGS	An+4*i	+= 4*n	RGS	true
indirect data address with pre-decrement	ldlg -(An),RGS	An-4*i-4	-= 4*n	RGS	true

ldss

load short and sign-extend

If the condition **CND** is true loads the short operand (16-bit word) from the effective data address **eda** in the data memory, sign-extends the value to 32 bits and stores it in the 32-bit destination **dst**. If the condition **cnd** is false the instruction performs no operations. For addressing modes with no **CND** parameter the variable **cnd** is always **true**. Some addressing modes update the indirect address register **An** as indicated in the addressing modes table. With the **(An)+,RGS** and **-(An),RGS** addressing modes the update parameter **n** is the number of registers contained in **RGS** and can take values from 1 to 19.

C language description

```
uint32 dst;
uint16 *eda;
boolean cnd;
if(cnd == true)
  dst = *eda & 0x8000 ? *eda | 0xFFFF0000 : *eda;
```

The C language statements for the calculation of the effective data address **eda** and for the **An** update operations are specified in the addressing modes table for each addressing mode.

Addressing Modes	assembly format	eda	An update	dst	cnd
direct 16-bit data address	ldss DA16$_S$,Rd,CND	DA16$_S$	not appl.	Rd	CND
indirect data address with direct update	ldss (An,AU12$_S$)*,Rd,CND	An	+= AU12$_S$	Rd	CND
indirect data address with indirect update	ldss (An,Ru)*,Rd,CND	An	+= Ru	Rd	CND
indirect data address with 12-bit offset	ldss (DO12$_S$,An),Rd,CND	An+DO12$_S$	no update	Rd	CND
indirect data address with index	ldss (Rx,An),Rd,CND	An+2*Rx	no update	Rd	CND
indirect data address with post-increment	ldss (An)+,RGS	An+2*i	+= 2*n	RGS	true
indirect data address with pre-decrement	ldss -(An),RGS	An-2*i-2	-= 2*n	RGS	true

ldsz

load short and zero-extend

If the condition **CND** is true loads the short operand (16-bit word) from the effective data address **eda** in the data memory, zero-extends it to 32 bits and stores it in the 32-bit destination **dst**. If the condition **cnd** is false the instruction performs no operations. For addressing modes with no **CND** parameter the variable **cnd** is always **true**. Some addressing modes update the indirect address register **An** as indicated in the addressing modes table. With the **(An)+,RGS** and **-(An),RGS** addressing modes the update parameter **n** is the number of registers contained in **RGS** and can take values from 1 to 19.

C language description

```
uint32 dst;
uint16 *eda;
boolean cnd;
if(cnd == true)
  dst = *eda;
```

The C language statements for the calculation of the effective data address **eda** and for the **An** update operations are specified in the addressing modes table for each addressing mode.

Addressing Modes	assembly format	eda	An update	dst	cnd
direct 16-bit data address	ldsz DA16$_S$,Rd,CND	DA16$_S$	not appl.	Rd	CND
indirect data address with direct update	ldsz (An,AU12$_S$)*,Rd,CND	An	+= AU12$_S$	Rd	CND
indirect data address with indirect update	ldsz (An,Ru)*,Rd,CND	An	+= Ru	Rd	CND
indirect data address with 12-bit offset	ldsz (DO12$_S$,An),Rd,CND	An+DO12$_S$	no update	Rd	CND
indirect data address with index	ldsz (Rx,An),Rd,CND	An+2*Rx	no update	Rd	CND
indirect data address with post-increment	ldsz (An)+,RGS	An+2*i	+= 2*n	RGS	true
indirect data address with pre-decrement	ldsz -(An),RGS	An-2*i-2	-= 2*n	RGS	true

mfdp

move from debug port

The 32-bit word driven on the debug input port **dbgi** of the processor is stored in the 32-bit destination **dst**.

C language description

```
uint32 dbgi,dst;
dst = dbgi;
```

Addressing Modes	assembly format	src	dst
single register	mfdp Rd	dbgi	Rd

move

move

If the condition **cnd** is true reads the 32-bit source operand **src** and stores it in the 32-bit destination operand **dst**. If the condition **cnd** is false the instruction performs no operation.

C language description

```
uint32 src,dst;
boolean cnd;
if(cnd == true)
  dst = src;
```

Addressing Modes	assembly format	src	dst	cnd
dual registers	move Rs,Rd,CND	Rs	Rd	CND
constant and single register	move C17$_S$,Rd,CND	C17$_S$	Rd	CND

mtdp

move to debug port

The 32-bit source operand **src** is transferred to the debug output port **dbgo**.

C language description

```
uint32 dbgo,src;
dbgo = src;
```

Addressing Modes	assembly format	src	dst
single register	mtdp Rs	Rs	dbgo

stbt

store byte

If the condition **cnd** is true extracts the least significant byte (8-bit word) from the 32-bit source operand **src** and stores it at the effective data address **eda** in data memory. If the condition **cnd** is false the instruction performs no operations. For addressing modes with no **CND** parameter the variable **cnd** is always **true**. Some addressing modes update the indirect address register **An** as indicated in the addressing modes table. With the **(An)+,RGS** and **-(An),RGS** addressing modes the update parameter **n** is the number of registers contained in **RGS** and can take values from 1 to 19.

C language description

```
uint32 src;
uint8 *eda;
if(CND == true)
  *eda = src;
```

The C language statements for the calculation of the effective data address **eda** and the **An** update operations are specified in the addressing modes table for each addressing mode.

Addressing Modes	assembly format	eda	An update	src	cnd
direct 16-bit data address	stbt Rs,DA16$_S$,CND	DA16$_S$	not appl.	Rs	CND
indirect data address with direct update	stbt Rs,(An,AU12$_S$)*,CND	An	+= AU12$_S$	Rs	CND
indirect data address with indirect update	stbt Rs,(An,Ru)*,CND	An	+= Ru	Rs	CND
indirect data address with 12-bit offset	stbt Rs,(DO12$_S$,An),CND	An+DO12$_S$	no update	Rs	CND
indirect data address with index	stbt Rs,(Rx,An),CND	An+Rx	no update	Rs	CND
indirect data address with post-increment	stbt RGS,(An)+	An+i	+= n	RGS	true
indirect data address with pre-decrement	stbt RGS,-(An)	An-i-1	-= n	RGS	true

stlg

store long

If the condition **cnd** is true stores the 32-bit source operand **src** at effective data address **eda** in the data memory. If the condition **cnd** is false the instruction performs no operations. For addressing modes with no **CND** parameter the variable **cnd** is always **true**. Some addressing modes update the indirect address register **An** as indicated in the addressing modes table. With the **(An)+,RGS** and **-(An),RGS** addressing modes the update parameter **n** is the number of registers contained in **RGS** and can take values from 1 to 19.

C language description

```
uint32 src,*eda;
if(CND == true)
  *eda = src;
```

The C language statements for the calculation of the effective data address **eda** and the **An** update operations are specified in the addressing modes table for each addressing mode.

Addressing Modes	assembly format	eda	An update	src	cnd
direct 16-bit data address	stlg Rs,DA16$_S$,CND	DA16$_S$	not appl.	Rs	CND
indirect data address with direct update	stlg Rs,(An,AU12$_S$)*,CND	An	+= AU12$_S$	Rs	CND
indirect data address with indirect update	stlg Rs,(An,Ru)*,CND	An	+= Ru	Rs	CND
indirect data address with 12-bit offset	stlg Rs,(DO12$_S$,An),CND	An+DO12$_S$	no update	Rs	CND
indirect data address with index	stlg Rs,(Rx,An),CND	An+4*Rx	no update	Rs	CND
indirect data address with post-increment	stlg RGS,(An)+	An+4*i	+= 4*n	RGS	true
indirect data address with pre-decrement	stlg RGS,-(An)	An-4*i-4	-= 4*n	RGS	true

stsh

store short

If the condition **cnd** is true extracts the lower 16 bits from the 32-bit source operand **src** and stores it at the effective data address **eda** in the data memory. If the condition **cnd** is false the instruction performs no operations. For addressing modes with no **CND** parameter the variable **cnd** is always **true**. Some addressing modes update the indirect address register **An** as indicated in the addressing modes table. With the **(An)+,RGS** and **-(An),RGS** addressing modes the update parameter **n** is the number of registers contained in **RGS** and can take values from 1 to 19.

C language description

```
uint32 src;
uint16 *eda;
if(CND == true)
  *eda = src;
```

The C language statements for the calculation of the effective data address **eda** and the **An** update operations are specified in the addressing modes table for each addressing mode.

Addressing Modes	assembly format	eda	An update	src	cnd
direct 16-bit data address	stsh Rs,DA16$_S$,CND	DA16$_S$	not appl.	Rs	CND
indirect data address with direct update	stsh Rs,(An,AU12$_S$)*,CND	An	+= AU12$_S$	Rs	CND
indirect data address with indirect update	stsh Rs,(An,Ru)*,CND	An	+= Ru	Rs	CND
indirect data address with 12-bit offset	stsh Rs,(DO12$_S$,An),CND	An+DO12$_S$	no update	Rs	CND
indirect data address with index	stsh Rs,(Rx,An),CND	An+2*Rx	no update	Rs	CND
indirect data address with post-increment	stsh RGS,(An)+	An+2*i	+= 2*n	RGS	true
indirect data address with pre-decrement	stsh RGS,-(An)	An-2*i-2	-= 2*n	RGS	true

8 Computation instructions

8.1 Common properties

Computation instructions perform mathematical operations on the data values of software programs. One or more source operands are transformed to a destination operand by an arithmetic, logic, shift, bit manipulation, or multiply operation.

8.2 Legend

The next sections define the bit accurate operations of the sf32 computation instructions grouped into categories and in alphabetical order for each category. The following paragraphs define the formats and notations used in individual instruction definitions.

8.2.1 Mnemonic

A four-character acronym of the instruction used to specify instructions in assembly language source code.

8.2.2 Text Description

Text description of the operations performed. Text descriptions reference the operand variables that are defined and used in the C language description

8.2.3 C language description

These C language statements are the bit true reference of the operations performed by an instruction. The following types and variables are used in the statements:

`uint32`	type: 32-bit unsigned integer
`sint32`	type: 32-bit signed integer
`uint16`	type: 16-bit unsigned integer
`uint5`	type: 5-bit unsigned integer
`boolean`	type: 1-bit Boolean variable, can take the values **`true`** and **`false`** or 1 and 0.

In addition to these variables the condition code flags in special register **CC** are used directly as destination operands. If the C language description of an instruction contains no statements that assign new values to the condition code flags then the instruction does not update the **CC** register.

Individual bits of non-array variables are referenced by the variable name followed by the bit number in square brackets. E.g. bit 23 of source operand 0 is referenced by **`src0[23].`**

The use of unsigned integers does not necessary mean that the underlying operands are unsigned. It means that the computations defined by the C statements are done assuming unsigned operands.

8.2.4 Addressing modes table

This table lists all addressing modes of the instruction. For each addressing mode the assembly language format is specified and the assignment of operands used in the C statements to operand specifiers in the assembly format is given.

8.2.5 Notes

Notes are optional and provide hints of how the instruction is used or if other instructions can do similar operations more efficiently.

8.3 Arithmetic Instructions

absl

absolute value

If the condition **cnd** is **true** the absolute value of the 32-bit source operand **src** is stored in the 32-bit destination operand **dst**. If the condition **cnd** is **false** the instruction performs no operations.

C language description

```
uint32 src,dst;
boolean cnd;
if(cnd == true)
  dst = src & 0x80000000 ? -src : src;
```

Addressing Modes	assembly format	src	dst	cnd
dual registers	absl Rs,Rd,CND	Rs	Rd	CND

addc

add with carry

If the condition **cnd** is **true** adds the 32-bit source operands **src0**, **src1** and the carry flag **CC.C**. The result is stored in the 32-bit destination operand **dst** and the flags in **CC** are updated. If the condition **cnd** is **false** the instruction performs no operations. The zero flag **CC.Z** is set only if **dst** is zero and if **CC.Z** was set before the operation. If one of these two conditions is not met **CC.Z** is cleared.

C language description

```
uint32 src0,src1,dst;
boolean cnd;
if(cnd == true){
  dst = src1 + src0 + CC.C;
  CC.C = (src1[31]&src0[31]) | (src1[31]&~dst[31]) | (src0[31]&~dst[31]);
  CC.O = (src1[31]&src0[31]&~dst[31]) | (~src1[31]&~src0[31]&dst[31]);
  CC.Z = CC.Z & (dst == 0) ? 1 : 0;
  CC.N = dst[31];
}
```

Addressing Modes	assembly format	src0	src1	dst	cnd
triadic registers	addc Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	addc C12$_U$,Rs1,Rd,CND	C12$_U$	Rs1	Rd	CND

addt

add to

If the condition **cnd** is **true** adds the two 32-bit source operands **src0** and **src1**, stores the result in the 32-bit destination operand **dst** and updates the flags in **CC**. If the condition **cnd** is **false** the instruction performs no operations. For addressing modes with no **CND** parameter the variable **cnd** is always **true**.

C language description

```
uint32 src0,src1,dst;
boolean cnd;
if(cnd == true){
  dst = src1 + src0;
  CC.C = (src1[31]&src0[31]) | (src1[31]&~dst[31]) | (src0[31]&~dst[31]);
  CC.O = (src1[31]&src0[31]&~dst[31]) | (~src1[31]&~src0[31]&dst[31]);
  CC.Z = dst == 0 ? 1 : 0;
  CC.N = dst[31];
}
```

Addressing Modes	assembly format	src0	src1	dst	cnd
triadic registers	addt Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	addt C16$_U$,Rs1,Rd	C16$_U$	Rs1	Rd	true
constant and dual registers	addt C12$_U$,Rs1,Rd,CND	C12$_U$	Rs1	Rd	CND

addh

add high

The 32-bit constant **$C32_U$** is added to the 32-bit source operand **src1**. The result is stored in the 32-bit destination operand **dst**. Bits [15:0] of constant **$C32_U$** are always zero.

C language description

```
uint32 C32_U,src1,dst;
dst = C32_U + src1 + src0;
```

Addressing Mode	assembly format	src0	src1	dst
constant and dual registers	addh $C32_U$,Rs1,Rd	$C32_U$	Rs1	Rd

Notes

Main purpose of the **addh** instruction is the generation of 32-bit constants. This is done by a **move $C17_S$,Rd** instruction followed by a **addh** instruction with the **dst** of the **move** used as both **src1** and **dst** operands. Bits[15:0] of the **$C17_S$** of the **move** instruction are the lower 16 bits and the **$C32_U$** of the **addh** instruction are the higher 16 bits of the 32-bit constant.

clzr

count leading zeros

If the condition **cnd** is **true** counts the number of zero bits in the 32-bit source operand **src** starting with the MSB until the first ‘1’ bit is found. The count is stored in the 32-bit destination operand **dst**. If no ‘1’ bit is found (**src == 0**) the count stored in the destination operand **dst** is 32. If the condition **cnd** is **false** the instruction performs no operations.

C language description

```
uint32 src,dst;
boolean cnd;
uint5 bti;
if(cnd == true){
  dst = 32;
  for(bti=31;bti >= 0;bti--)
    if(src[bti] == 1){
      dst = 31 – bti;
      break;
    }
}
```

Addressing Modes	assembly format	src	dst	cnd
dual registers	clzr Rs,Rd,CND	Rs	Rd	CND

cmpc

compare with carry

Subtracts the 32-bit source operand **src0** and the carry flag **CC.C** from the 32-bit source operand **src1** and updates the flags in **CC** according to the result. **$C17_S$** is sign-extended to 32 bits before being used as **src0**. The zero flag **CC.Z** is set only if **dst** is zero and if **CC.Z** was set before the operation. If one of these two conditions is not met **CC.Z** is cleared.

C language description

```
uint32 src0,src1,tmp;
tmp = src1 - src0 – CC.C;
CC.C = (~src1[31]&src0[31]) | (~src1[31]&tmp[31]) | (src0[31]&tmp[31]);
CC.O = (src1[31]&~src0[31]&~tmp[31]) | (~src1[31]&src0[31]&tmp[31]);
CC.Z = CC.Z & (tmp == 0) ? 1 : 0;
CC.N = tmp[31];
```

Addressing Modes	assembly format	src0	src1
dual registers	cmpc Rs0,Rs1	Rs0	Rs1
constant and single register	cmpc $C17_S$,Rs1	$C17_S$	Rs1

comp

compare

Subtracts the 32-bit source operand **src0** from the 32-bit source operand **src1** and updates the flags in **CC** according to the result. **C17**$_S$ is sign-extended to 32 bits before being used as **src0**.

C language description

```
uint32 src0,src1,tmp;
tmp = src1 - src0;
CC.C = (~src1[31]&src0[31]) | (~src1[31]&tmp[31]) | (src0[31]&tmp[31]);
CC.O = (src1[31]&~src0[31]&~tmp[31]) | (~src1[31]&src0[31]&tmp[31]);
CC.Z = tmp == 0 ? 1 : 0;
CC.N = tmp[31];
```

Addressing Modes	assembly format	src0	src1
dual registers	comp Rs0,Rs1	Rs0	Rs1
constant and single register	comp C17$_S$,Rs1	C17$_S$	Rs1

negt

negate

If the condition **cnd** is **true** the 2's complement of the 32-bit source operand **src** is stored in the 32-bit destination operand **dst**. If the condition **cnd** is **false** the instruction performs no operations

C language description

```
uint32 src,dst;
boolean cnd;
if(cnd == true)
  dst = –src;
```

Addressing Modes	assembly format	src	dst	cnd
dual registers	negt Rs,Rd,CND	Rs	Rd	CND

subc

subtract with carry

If the condition **cnd** is **true** subtracts the 32-bit source operand **src0** and the carry flag **CC.C** from the 32-bit source operand **src1**. The result is stored in the 32-bit destination operand **dst** and the flags in **CC** are updated. If the condition **cnd** is **false** the instruction performs no operations. The zero flag **CC.Z** is set only if **dst** is zero and if **CC.Z** was set before the operation. If one of these two conditions is not met **CC.Z** is cleared.

C language description

```
uint32 src0,src1,dst;
boolean cnd;
if(cnd == true){
  dst = src1 - src0 - CC.C;
  CC.C = (~src1[31]&src0[31]) | (~src1[31]&dst[31]) | (src0[31]&dst[31]);
  CC.O = (src1[31]&~src0[31]&~dst[31]) | (~src1[31]&src0[31]&dst[31]);
  CC.Z = CC.Z & (dst == 0) ? 1 : 0;
  CC.N = dst[31];
}
```

Addressing Modes	assembly format	src0	src1	dst	cnd
triadic registers	subc Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	subc C12$_U$,Rs1,Rd,CND	C12$_U$	Rs1	Rd	CND

subf

subtract from

If the condition **cnd** is **true** subtracts the 32-bit source operand **src0** from the 32-bit source operand **src1**, stores the result in the 32-bit destination operand **dst** and updates the flags in **CC**. If the condition **cnd** is **false** the instruction performs no operations. For addressing modes with no **CND** parameter the variable **cnd** is always **true**.

C language description

```
  uint32 src0,src1,dst;
  boolean cnd;
  if(cnd == true){
    dst = src1 - src0;
    CC.C = (~src1[31]&src0[31]) | (~src1[31]&dst[31]) | (src0[31]&dst[31]);
    CC.O = (src1[31]&~src0[31]&~dst[31]) | (~src1[31]&src0[31]&dst[31]);
    CC.Z = dst == 0 ? 1 : 0;
    CC.N = dst[31];
  }
```

Addressing Modes	assembly format	src0	src1	dst	cnd
triadic registers	subf Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	subf $C16_U$,Rs1,Rd	$C16_U$	Rs1	Rd	true
constant and dual registers	subf $C12_U$,Rs1,Rd,CND	$C12_U$	Rs1	Rd	CND

8.4 Logic Instructions

andb

logic AND bit wise

If the condition **cnd** is **true** performs a bit wise logic AND operation between the two 32-bit source operands **src0** and **src1**, stores the result in the 32-bit destination operand **dst** and updates the flags in **CC**. If the condition **cnd** is **false** the instruction performs no operations. For addressing modes with no **CND** parameter the variable **cnd** is always **true**. The order of C statements is important regarding the update of **CC.O**. **CC.O** uses the old value of **CC.C** as source operand before **CC.C** is updated by the **andb** instruction.

C language description

```
  uint32 src0,src1,dst;
  boolean cnd,par;
  uint5 bti;
  if(cnd == true){
    dst = src1 & src0;
    par = 0;
    for(bti=0;bti < 32;bti++)
      par ^= dst[bti];
    CC.O = par ^ CC.C;
    CC.C = par;
    CC.Z = dst == 0 ? 1 : 0;
    CC.N = dst[31];
  }
```

Addressing Modes	assembly format	src0	src1	dst	cnd
triadic registers	andb Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	andb $C16_U$,Rs1,Rd	$C16_U$	Rs1	Rd	true

Notes

The **andb** instruction is the only logic instruction that updates **CC**. This is because ‘and’ operations are frequently used to test bits or bit fields against zero.

A special feature of the sf32 **andb** instruction is the parity generation in **CC.C** and **CC.O**. It is useful for CRC calculations and other security and data integrity related algorithms. **CC.C** contains the parity of the destination operand of the current **andb** instruction. **CC.O** is used for the parity of longer bit strings > 32 bits. For the parity of long bit strings first **CC.C** and **CC.O** are cleared by e.g. a **move 0,CC** instruction. Then a sequence of **andb** instructions is executed, as many as are necessary to cover the entire long string. After the last **andb** instruction **CC.O** is the parity of the entire long string.

invt

invert

If the condition **cnd** is **true** inverts the 32-bit source operand **src** and stores the result in the 32-bit destination operand **dst**. If the condition **cnd** is **false** the instruction performs no operations.

C language description

```
uint32 src,dst;
boolean cnd;
if(cnd == true)
  dst = ~src;
```

Addressing Modes	assembly format	src	dst	cnd
dual registers	invt Rs,Rd,CND	Rs	Rd	CND

iorb

inclusive OR bit wise

If the condition **cnd** is **true** performs a bit wise inclusive or between the two 32-bit source operands **src0** and **src1** and stores the result in the 32-bit destination operand **dst**. If the condition **cnd** is **false** the instruction performs no operations. For addressing modes with no **CND** parameter the variable **cnd** is always **true**.

C language description

```
uint32 src0,src1,dst;
boolean cnd;
if(cnd == true)
  dst = src1 | src0;
```

Addressing Modes	assembly format	src0	src1	dst	cnd
triadic registers	iorb Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	iorb $C16_U$,Rs1,Rd	$C16_U$	Rs1	Rd	true

xorb

exclusive OR

If the condition **cnd** is **true** performs a bit wise exclusive or between the two 32-bit source operands **src0** and **src1** and stores the result in the 32-bit destination operand **dst**. If the condition **cnd** is **false** the instruction performs no operations. For addressing modes with no **CND** parameter the variable **cnd** is always **true**.

C language description

```
uint32 src0,src1,dst;
boolean cnd;
if(cnd == true)
  dst = src1 ^ src0;
```

Addressing Modes	assembly format	src0	src1	dst	cnd
triadic registers	xorb Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	xorb $C16_U$,Rs1,Rd	$C16_U$	Rs1	Rd	true

8.5 Shift Instructions

shlf

shift left with feedback

If the condition **cnd** is **true** performs a left shift with feedback (rotate) operation of the 32-bit source operand **src** and stores the result in the 32-bit destination **dst**. The shift count **shc5** can take values from 0 to 31. If the condition **cnd** is **false** the instruction performs no operations. The shift with feedback operation is a left shift that shifts in the bits shifted out at the MSB of the operand back in at the LSB of the operand. In addressing modes with indirect shift count **shc5** is equal to bits [4:0] of source register **Rs0**. Bits [31:5] of **Rs0** are ignored.

C language description

```
uint32 src,dst;
uint5 shc5;
boolean cnd;
if(cnd == true)
  dst = (src << shc5) | (src >> (32 – shc5));
```

Addressing Modes	assembly format	shc5	src	dst	cnd
triadic registers	shlf Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	shlf SHC5$_U$,Rs1,Rd,CND	SHC5$_U$	Rs1	Rd	CND

shlz

shift left with zero fill

If the condition **cnd** is **true** performs a left shift with zero fill of the 32-bit source operand **src** and stores the result in the 32-bit destination **dst**. The shift count **shc5** can take values from 0 to 31. If the condition **cnd** is **false** the instruction performs no operations. In addressing modes with indirect shift count **shc5** is equal to bits [4:0] of source register **Rs0**. Bits [31:5] of **Rs0** are ignored.

C language description

```
uint32 src,dst;
uint5 shc5;
boolean cnd;
if(cnd == true)
  dst = src << shc5;
```

Addressing Modes	assembly format	shc5	src	dst	cnd
triadic registers	shlz Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	shlz SHC5$_U$,Rs1,Rd,CND	SHC5$_U$	Rs1	Rd	CND

shrs

shift right signed

If the condition **cnd** is **true** performs a signed right shift of the 32-bit source operand **src** and stores the result in the 32-bit destination **dst**. The shift count **shc5** can take values from 0 to 31. If the condition **cnd** is **false** the instruction performs no operations. Signed shift means that the sign of the source operand **src[31]** is preserved and the destination operand **dst** has the same sign as the source operand **src**. In addressing modes with indirect shift count **shc5** is equal to bits [4:0] of source register **Rs0**. Bits [31:5] of **Rs0** are ignored.

C language description

```
uint32 src,dst;
uint5 shc5;
boolean cnd;
if(cnd == true){
  dst = src >> shc5;
  if(src[31])
    dst |= 0xFFFFFFFF << (32 – shc5);
}
```

Addressing Modes	assembly format	shc5	src	dst	cnd
triadic registers	shrs Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	shrs SHC5$_U$,Rs1,Rd,CND	SHC5$_U$	Rs1	Rd	CND

shru

shift right unsigned

If the condition **cnd** is **true** performs a right shift of the 32-bit source operand **src** and stores the result in the 32-bit destination **dst**. The shift count **shc5** can take values from 0 to 31. If the condition **cnd** is **false** the instruction performs no operations. In addressing modes with indirect shift count **shc5** is equal to bits [4:0] of source register **Rs0**. Bits [31:5] of **Rs0** are ignored.

C language description

```
uint32 src,dst;
uint5 shc5;
boolean cnd;
if(cnd == true)
  dst = src >> shc5;
```

Addressing Modes	assembly format	shc5	src	dst	cnd
triadic registers	shru Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	shru SHC5$_U$,Rs1,Rd,CND	SHC5$_U$	Rs1	Rd	CND

8.6 Bit manipulation instructions

btcl

bit clear

If the condition **cnd** is **true** clears the bit of the 32-bit source operand **src** indexed by **bti5** and stores the result in the 32-bit destination **dst**. If the condition **cnd** is **false** the instruction performs no operations. The bit index **bti5** can take values from 0 to 31. In addressing modes with indirect bit index **bti5** is equal to bits [4:0] of the source register **Rs0**. Bits [31:5] of **Rs0** are ignored.

C language description

```
uint32 src,dst;
uint5 bti5;
boolean cnd;
if(cnd == true)
  dst = src & ~(1 << bti5);
```

Addressing Modes	assembly format	bti5	src	dst	cnd
triadic registers	btcl Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	btcl BTI5$_U$,Rs1,Rd,CND	BTI5$_U$	Rs1	Rd	CND

btst

bit set

If the condition **cnd** is **true** sets the bit of the 32-bit source operand **src** indexed by **bti5** and stores the result in the 32-bit destination **dst**. If the condition **cnd** is **false** the instruction performs no operations. The bit index **bti5** can take values from 0 to 31. In addressing modes with indirect bit index **bti5** is equal to bits [4:0] of the source register **Rs0**. Bits [31:5] of **Rs0** are ignored.

C language description

```
uint32 src,dst;
uint5 bti5;
boolean cnd;
if(cnd == true)
  dst = src | (1 << bti5);
```

Addressing Modes	assembly format	bti5	src	dst	cnd
triadic registers	btst Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	btst BTI5$_U$,Rs1,Rd,CND	BTI5$_U$	Rs1	Rd	CND

bttg

bit toggle

If the condition **cnd** is **true** toggles the bit of the 32-bit source operand **src** indexed by **bti5** and stores the result in the 32-bit destination **dst**. If the condition **cnd** is **false** the instruction performs no operations. The bit index **bti5** can take values from 0 to 31. In addressing modes with indirect bit index **bti5** is equal to bits [4:0] of the source register **Rs0**. Bits [31:5] of **Rs0** are ignored.

C language description

```
uint32 src,dst;
uint5 bti5;
boolean cnd;
if(cnd == true)
  dst = src ^ (1 << bti5);
```

Addressing Modes	assembly format	bti5	src	dst	cnd
triadic registers	bttg Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND
constant and dual registers	bttg BTI5$_U$,Rs1,Rd,CND	BTI5$_U$	Rs1	Rd	CND

btts

bit test

If the condition **cnd** is **true** tests the bit of the 32-bit source operand **src** indexed by **bti5** and updates the condition codes in **CC** according to the result. The bit index **bti5** can take values from 0 to 31. If the condition **cnd** is **false** the instruction performs no operations. In addressing modes with indirect bit index **bti5** is equal to bits [4:0] of source register **Rs0**. Bits [31:5] of **Rs0** are ignored.

C language description

```
uint32 src,tmp;
uint5 bti5;
boolean cnd;
if(cnd == true){
  tmp = src & (1 << bti);
  CC.C = 0;
  CC.O = 0;
  CC.Z = tmp == 0 ? 1 : 0;
  CC.N = tmp[31];
}
```

Addressing Modes	assembly format	bti5	src	cnd
triadic registers	btts Rs0,Rs1,CND	Rs0	Rs1	CND
constant and dual registers	btts BTI5$_U$,Rs1,CND	BTI5$_U$	Rs1	CND

8.7 Multiply Instructions

mlcs

multiply constant signed

Performs a signed multiply of the 16-bit constant **C16$_S$** and the 32-bit source operand **src**. The lower 32 bits of the 47-bit product are stored in the 32-bit destination **dst**.

C language description

```
uint32 dst;
sint32 src;
dst = C16_S * src;
```

Addressing Modes	assembly format	src0	src1	dst
constant and dual registers	mlcs C16$_S$,Rs1,Rd	C16$_S$	Rs1	Rd

mlcu

multiply constant unsigned

Performs an unsigned multiply of the 16-bit constant **$C16_U$** and the 32-bit source operand **src**. The lower 32 bits of the 48-bit product are stored in the 32-bit destination **dst**.

C language description

```
uint32 dst,src;
dst = C16U * src;
```

Addressing Modes	assembly format	src0	src1	dst
constant and dual registers	mlcu $C16_U$,Rs1,Rd	$C16_U$	Rs1	Rd

mlhs

multiply high signed

If the condition **cnd** is **true** performs a signed multiply of the two 32-bit source operands **src0** and **src1**. The 63-bit product is right shifted (sign preserved) by 32 bits, sign-extended to 32 bits and stored in the 32-bit destination **dst**. If the condition **cnd** is **false** the instruction performs no operations.

C language description

```
uint32 dst;
sint32 scr0,scr1;
boolean cnd;
if(cnd == true)
  dst = (src1 * src0) >> 32;
```

Addressing Modes	assembly format	src0	src1	dst	cnd
triadic registers	mlhs Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND

mlhu

multiply high unsigned

If the condition **cnd** is **true** performs an unsigned multiply of the two 32-bit source operands **src0** and **src1**. The 64-bit product is right shifted by 32 bits and stored in the 32-bit destination dst. If the condition **cnd** is **false** the instruction performs no operations.

C language description

```
uint32 src0,src1,dst;
boolean cnd;
if(cnd == true)
  dst = (src1 * src0) >> 32;
```

Addressing Modes	assembly format	src0	src1	dst	cnd
triadic registers	mlhu Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND

mult

multiply

If the condition **cnd** is **true** performs a multiply of the two 32-bit source operands **src0** and **src1** and stores the lower 32 bits of the 64-bit product in the 32-bit destination operand **dst**. If the condition **cnd** is **false** the instruction performs no operations.

C language description

```
uint32 src0,src1,dst;
boolean cnd;
if(cnd == true)
  dst = src1 * src0;
```

Addressing Modes	assembly format	src0	src1	dst	cnd
triadic registers	mult Rs0,Rs1,Rd,CND	Rs0	Rs1	Rd	CND

8.8 Endianess Conversion Instructions

ibol

invert byte order long

If the condition **cnd** is **true** inverts the byte order of the 32-bit source operand **src** and stores the result in the 32-bit destination operand **dst**. If the condition **cnd** is **false** the instruction performs no operations.

C language description

```
uint32 src,dst;
boolean cnd;
if(cnd == true)
  dst = ((src&0xFF)<<24) | ((src&0xFF00)<<8) | ((src>>8)&0xFF00) | (src>>24);
```

Addressing Modes	assembly format	src	dst	cnd
dual registers	ibol Rs,Rd,CND	Rs	Rd	CND

ibos

invert byte order short

If the condition **cnd** is **true** swaps bytes 0 and 1 of the 32-bit source operand **src** and stores the result in the 32-bit destination operand **dst**. The higher 16 bits of the source operand are passed to the destination without change. If the condition **cnd** is **false** the instruction performs no operations.

C language description

```
uint32 src,dst;
boolean cnd;
if(cnd == true)
  dst = (src&0xFFFF0000) | ((src&0xFF)<<8) | ((src>>8)&0xFF);
```

Addressing Modes	assembly format	src	dst	cnd
dual registers	ibos Rs,Rd,CND	Rs	Rd	CND

9 Flow control instructions

9.1 Common properties

The instructions of this category control the program flow. They don't perform data operations and do not update general purpose registers.

9.2 Legend

The next section lists the flow control instructions in alphabetical order and defines the bit accurate operations they perform. The following paragraphs define the formats and notations used in individual instruction definitions.

9.2.1 Mnemonic

A four-character acronym of the instruction used to specify instructions in assembly language source code.

9.2.2 Text Description

Text description of the operations performed. Text descriptions reference the operand variables that are defined and used in the C language description

9.2.3 C language description

These C language statements are the bit true reference of the operations performed by an instruction. The following types and variables are used in the statements:

`uint32` type: 32-bit unsigned integer

`Boolean` type: 1-bit Boolean variable, can take the values `true` and `false` or 1 and 0.

Individual bits of variables are referenced by the variable name followed by the bit number in square brackets. E.g. bit 23 of source operand 0 is referenced by **`src0[23].`**

The use of unsigned integers does not necessary mean that the underlying operands are unsigned. It means that the computations defined by the C statements are done assuming unsigned operands.

9.2.4 Addressing modes table

This table lists all addressing modes of the instruction. For each addressing mode the assembly language format is specified.

9.2.5 Notes

Notes are optional and provide hints of how the instruction is used or if other instructions can do similar operations more efficiently.

9.3 Instruction details

brlc — decrement loop counter and branch if non zero

Decrements register **LC** (loop counter). If **LC** is unequal zero after the decrement program execution continues at the effective instruction address **`eia`** calculated from the current instruction address **`cia`** and constant **IO16$_S$**. The 16-bit instruction address offset **IO16$_S$** is sign-extended to 32 bits and added to **`cia`**. The two LSBs of **IO16$_S$** are always zero and are not contained in the instruction’s opcode. If **LC** is zero after the decrement program execution continues with the next instruction in sequence.

C language description

```
uint32 tmp,*cia,*eia;
LC -= 1;
if(LC != 0){
  tmp = IO16S & 0x8000 ? IO16S | 0xFFFF0000 : IO16S;
  eia = cia + tmp;
}
else
  eia = cia + 4;
```

Addressing Modes	assembly format
16-bit instruction address offset	brlc IO16$_S$

brxx — branch if condition ‘xx’ is true

This is a group of 14 conditional branch instructions. Individual instructions have different mnemonics (see addressing modes table), **xx** is a placeholder for the two characters that express the condition.

If the condition **`cnd`** is true program execution continues at the effective instruction address **`eia`** calculated from the current instruction address **`cia`** and constant **IO16$_S$**. The 16-bit instruction address offset **IO16$_S$** is sign-extended to 32 bits and added to **`cia`**. The two LSBs of **IO16$_S$** are always zero and are not contained in the instruction’s opcode. If the condition **`cnd`** is false instruction execution continues with the next instruction in sequence.

The **`bsxx`** addressing mode includes the speculation flag **`S`**. Processor implementations with branch speculation functionality can use this flag to decide whether to speculatively take a branch or not in cases where the condition **`cnd`** is not evaluated yet by the time the conditional branch instruction is decoded. In case of wrong speculation these implementations must revert back to the correct branch option. The **`S`** flag is a feature to improve the performance of conditional branch instruction execution. Processor implementations may or may not use the flag. The setting of the **`S`** flag has no impact on any operand results.

C language description

```
uint32 tmp,*cia,*eia;
boolean cnd;
if(cnd == true){
  tmp = IO16S & 0x8000 ? IO16S | 0xFFFF0000 : IO16S;
  eia = cia + tmp;
}
else
  eia = cia + 4;
```

Addressing modes

All of the 14 conditional branch instructions have the same addressing mode: “16-bit instruction address offset with speculation”. In the table below the addressing mode column is omitted. Instead the table includes a column that specifies the conditions **`cnd`** as C language statements. The following variables are used in the statements:

```
boolean C,O,Z,N;
C = CC.C;
O = CC.O;
Z = CC.Z;
N = CC.N;
```

Instruction	Condition	assembly format
branch if no carry	`CND = ~C;`	brnc IO16$_S$,S
branch if carry	`CND = C;`	brcr IO16$_S$,S
branch if no overflow	`CND = ~O;`	brno IO16$_S$,S
branch if overflow	`CND = O;`	brof IO16$_S$,S
branch if non zero	`CND = ~Z;`	brnz IO16$_S$,S
branch if zero	`CND = Z;`	brzr IO16$_S$,S
branch if positive	`CND = ~N;`	brps IO16$_S$,S
branch if negative	`CND = N;`	brng IO16$_S$,S
Branch if lower or same	`CND = C | Z;`	brls IO16$_S$,S
branch if higher	`CND = ~C & ~Z;`	brhi IO16$_S$,S
branch if lower	`CND = (N & ~O) | (~N & O);`	brlo IO16$_S$,S
branch if greater of equal	`CND = (N & O) | (~N & ~O);`	brge IO16$_S$,S
branch if lower or equal	`CND = Z | (N & ~O) | (~N & O);`	brle IO16$_S$,S
branch if greater	`CND = ~Z & ((N & O) | (~N & ~O));`	brgt IO16$_S$,S

clie

clear interrupt enable

Disables interrupts by clearing the interrupt enable bit **IE** in register **CS**.

C language description

```
CS.IE = 0;
```

Addressing Modes	assembly format
implied	clie

jump

jump

Program execution continues at the effective instruction address **eia** generated from a constant in the opcode or from register **TA**. The two LSBs of the 32-bit effective instruction address **eia** are always zero.

C language description

```
uint32 *eia;
```

The C language statements for the calculation of **eia** are specified in the addressing modes table for each addressing mode.

Addressing Modes	assembly format	eia
implied	jump	eia = TA;
29-bit absolute instruction address	jump IA29$_U$	eia = IA29$_U$;

jpsr

jump to subroutine

The address of the next instruction in sequence following the **jpsr** instruction is saved in register **SA**. This is the current instruction address **cia** plus 4. Program execution continues at the effective instruction address **eia** generated from a constant in the opcode or from register **TA**. The two LSBs of the 32-bit effective instruction address **eia** are always zero.

C language description

```
uint32 *cia,*eia;
SA = cia + 4;
```

The C language statements for the calculation of **eia** are specified in the addressing modes table for each addressing mode.

Addressing Modes	assembly format	eia
implied	jpsr	eia = TA;
29-bit absolute instruction address	jpsr IA29$_U$	eia = IA29$_U$;

Notes

sf32 processors do not automatically save and restore the return addresses of sub-routines on a stack. For nested sub-routines software must save and restore special register **SA** using store and load instructions. In the lowest nesting level where no further sub-routines are called saving and restoring of **SA** is not necessary.

rsie

restore interrupt enable

Copies the interrupt enable save bit **IS** in **CS** to the **IE** bit in **CS**.

C language description

```
CS.IE = CS.IS;
```

Addressing Modes	assembly format
implied	rsie

Notes

The **rsie** instruction is used to restore the original interrupt enable state after it has been saved with a **scie** instruction.

rspc

restore program counter

The current instruction address **cia** is set with the 32-bit value driven on the debug input port **dbgi**. The two LSBs of **cia** are forced to zero; the corresponding two LSBs from the debug input port are ignored.

C language description

```
uint32 dbgi,*cia;
cia = dbgi & 0xFFFFFFFC;
```

Addressing Modes	assembly format
implied	rspc

Notes

The **svpc** instruction is used by software debugging systems to save the current instruction address when the processor is in the stopped state. The debugger can then execute debugger utility routines in normal operation mode. To continue execution of the program under debug an **rspc** instruction is injected while the processor is in the stopped state to restore the original instruction address.

rtir

return from interrupt

The condition codes **CC** are restored from a hidden register where they had been saved when the interrupt was started. The interrupt flag in **CS.IR** is cleared. Program execution continues at the address in register **IA** as effective instruction address **eia**. If the processor is currently not executing an interrupt the behavior of **rtir** instructions is not defined.

C language description

```
uint32 *cia,*eia;
if(CS.IR){
  eia = IA;
  CC = hidden condition codes register;
  CS.IR = 0;
}
```

Addressing Modes	assembly format
implied	rtir

rtsr

return from subroutine

Program execution continues at the address in register **SA** as effective instruction address **eia**.

C language description

```
uint32 *eia;
eia = SA;
```

Addressing Modes	assembly format
implied	rtsr

Notes

sf32 processors do not automatically save and restore the return addresses of sub-routines on a stack. For nested sub-routines software must save and restore register **SA** using store and load instructions. In the lowest nesting level where no further sub-routines are called saving and restoring of **SA** is not necessary.

scie

save and clear interrupt enable

Copies the interrupt enable bit **IE** in **CS** to the **IS** bit in **CS** and then clears **IE**. Disables interrupts.

C language description

```
CS.IS = CS.IE;
CS.IE = 0;
```

Addressing Modes	assembly format
implied	scie

Notes

The **scie** instruction is used to temporarily disable interrupts and then restore the original interrupt enable state with an **rsie** instruction.

stie

set interrupt enable

Enables interrupts by setting the interrupt enable bit **IE** in register **CS**.

C language description

```
CS.IE = 1;
```

Addressing Modes	assembly format
implied	stie

stop

stop

Instruction fetching stops and the processor waits until execution of previously fetched instructions is finished. Then the debug state is entered. To resume program execution external debug hardware must signal the end of the debug state.

C language description

```
Not applicable
```

Addressing Modes	assembly format
implied	stop

Notes

The **stop** instruction is used by software debugging systems to set instruction break points. Debugger software replaces instructions at desired break point positions with **stop** instructions. Debugger controlled single stepping through programs is also done using **stop** instructions.

svpc

save program counter

The current instruction address **cia** is transferred to the debug output port **dbgo**.

C language description

```
uint32 dbgo,*cia;
dbgo = cia;
```

Addressing Modes	assembly format
implied	svpc

Notes

The **svpc** instruction is used by software debugging systems to save the current instruction address when the processor is in the stopped state. The debugger can then execute debugger utility routines in normal operation mode. To continue execution of the program under debug an **rspc** instruction is injected while the processor is in the stopped state to restore the original instruction address.

Instruction Coding

The following table contains the opcodes of all sf32 base ISA instructions. The instructions are listed in alphabetical order. For instructions with multiple addressing modes all addressing modes are listed sequentially in the table. Following the opcode table are two more tables. The first table explains the color coding of the opcode table. The second table defines the bit assignments of bit fields in the opcode table.

	Addressing Modes	opcode bits																															
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
absl	Rs,Rd,CND	CND				0	1	0	0	0	0	0	Rs					Rd					0	0	0	0	0	0	1	1	0	0	0
addc	Rs0,Rs1,Rd,CND	CND				0	0	1	0	0	0	0	Rs0					Rd					Rs1					0	1	1	0	0	0
	C12$_U$,Rs1,Rd,CND	CND				C12$_U$												Rd					Rs1					0	1	1	0	0	1
addh	C32$_U$,Rs1,Rd	C32$_U$																Rd					Rs1					0	1	1	0	1	0
addt	Rs0,Rs1,Rd,CND	CND				0	0	1	0	0	0	0	Rs0					Rd					Rs1					0	0	1	0	0	0
	C12$_U$,Rs1,Rd,CND	CND				C12$_U$												Rd					Rs1					0	0	1	0	0	1
	C16$_U$,Rs1,Rd	C16$_U$																Rd					Rs1					0	0	1	0	1	0
andb	Rs0,Rs1,Rd,CND	CND				0	0	1	0	0	0	0	Rs0					Rd					Rs1					1	0	1	0	0	0
	C16$_U$,Rs1,Rd	C16$_U$																Rd					Rs1					1	0	1	0	1	0
brcr	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	0	0	0	1	0	1	1	0
brge	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	1	0	1	1	0	1	1	0
brgt	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	1	1	0	1	0	1	1	0
brhi	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	1	0	0	1	0	1	1	0
brlc	IO16$_S$	0	0	0	0	0	0	0	0	IO16$_S$														0	1	1	1	1	0	0	1	1	0
brle	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	1	1	0	0	0	1	1	0
brlo	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	1	0	1	0	0	1	1	0
brls	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	1	0	0	0	0	1	1	0
brnc	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	0	0	0	0	0	1	1	0
brng	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	0	1	1	1	0	1	1	0
brno	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	0	0	1	0	0	1	1	0
brnz	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	0	1	0	0	0	1	1	0
brof	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	0	0	1	1	0	1	1	0
brps	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	0	1	1	0	0	1	1	0
brzr	IO16$_S$,S	0	0	0	0	0	0	0	0	IO16$_S$														S	1	0	1	0	1	0	1	1	0
btcl	Rs0,Rs1,Rd,CND	CND				1	0	1	0	0	0	0	Rs0					Rd					Rs1					1	0	0	0	0	0
	BTI5$_U$,Rs1,Rd,CND	CND				1	0	0	0	0	0	0	BTI5$_U$					Rd					Rs1					1	0	0	0	0	0
btst	Rs0,Rs1,Rd,CND	CND				1	0	1	0	0	0	0	Rs0					Rd					Rs1					1	1	0	0	0	0
	BTI5$_U$,Rs1,Rd,CND	CND				1	0	0	0	0	0	0	BTI5$_U$					Rd					Rs1					1	1	0	0	0	0
bttg	Rs0,Rs1,Rd,CND	CND				1	0	1	0	0	0	0	Rs0					Rd					Rs1					1	1	1	0	0	0
	BTI5$_U$,Rs1,Rd,CND	CND				1	0	0	0	0	0	0	BTI5$_U$					Rd					Rs1					1	1	1	0	0	0
btts	Rs0,Rs1,CND	CND				1	0	1	0	0	0	0	Rs0					0	0	0	0	0	Rs1					1	0	1	0	0	0
	BTI5$_U$,Rs1,CND	CND				1	0	0	0	0	0	0	BTI5$_U$					0	0	0	0	0	Rs1					1	0	1	0	0	0
clie	implied	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0
clzr	Rs,Rd,CND	CND				0	1	0	0	0	0	0	Rs					Rd					0	0	0	0	0	1	0	1	0	0	0
cmpc	C16$_S$,Rs1	0	0	0	0	1	1	1	0	0	0	0	Rs0					0	0	0	0	0	Rs1					0	1	0	0	0	0
	C17$_S$,Rs1	C17$_S$																0	0	0	0	0	Rs1					1	1	C	0	1	1
comp	Rs0,Rs1	0	0	0	0	1	1	1	0	0	0	0	Rs0					0	0	0	0	0	Rs1					0	0	0	0	0	0
	C17$_S$,Rs1	C17$_S$																0	0	0	0	0	Rs1					1	0	C	0	1	1
ibol	Rs,Rd,CND	CND				0	1	0	0	0	0	0	Rs					Rd					0	0	0	0	0	1	1	1	0	0	0
ibos	Rs,Rd,CND	CND				0	1	0	0	0	0	0	Rs					Rd					0	0	0	0	0	1	1	0	0	0	0
invt	Rs,Rd,CND	CND				0	1	0	0	0	0	0	Rs					Rd					0	0	0	0	0	1	0	0	0	0	0
iorb	Rs0,Rs1,Rd,CND	CND				0	0	1	0	0	0	0	Rs0					Rd					Rs1					1	1	0	0	0	0
	C16$_U$,Rs1,Rd	C16$_U$																Rd					Rs1					1	1	0	0	1	0
jump	IA29$_U$	IA29$_U$																											0	1	1	1	0
	implied	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
jpsr	IA29$_U$	IA29$_U$																											1	1	1	1	0
	implied	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0
ldbs	DA16$_S$,Rd,CND	CND				DA16$_S$												0	DA16$_S$				Rd					0	0	1	1	0	0
	(DO12$_S$,An),Rd,CND	CND				DO12$_S$												1	An				Rd					0	0	1	1	0	0
	(An,AU12$_S$)*,Rd,CND	CND				AU12$_S$												0	An				Rd					0	0	1	1	0	1
	(An,Ru)*,Rd,CND	CND				0	0	0	0	0	0	0	Ru					1	An				Rd					0	0	1	1	0	1
	(Rx,An),Rd,CND	CND				0	1	0	0	0	0	0	Rx					1	An				Rd					0	0	1	1	0	1
	(An)+,RGS	S	T	L	0	1	0	1	2	3	4	5	6	7	8	9	A	1	An				B	P	Q	U	V	0	0	1	1	0	1
	-(An),RGS	V	U	Q	P	1	1	B	A	9	8	7	6	5	4	3	2	1	An				1	0	L	T	S	0	0	1	1	0	1
ldbz	DA16$_S$,Rd,CND	CND				DA16$_S$												0	DA16$_S$				Rd					0	0	0	1	0	0
	(DO12$_S$,An),Rd,CND	CND				DO12$_S$												1	An				Rd					0	0	0	1	0	0
	(An,AU12$_S$)*,Rd,CND	CND				AU12$_S$												0	An				Rd					0	0	0	1	0	1
	(An,Ru)*,Rd,CND	CND				0	0	0	0	0	0	0	Ru					1	An				Rd					0	0	0	1	0	1
	(Rx,An),Rd,CND	CND				0	1	0	0	0	0	0	Rx					1	An				Rd					0	0	0	1	0	1
	(An)+,RGS	S	T	L	0	1	0	1	2	3	4	5	6	7	8	9	A	1	An				B	P	Q	U	V	0	0	0	1	0	1

	-(An),RGS	V	U	Q	P	1	1	B	A	9	8	7	6	5	4	3	2	1	An				1	0	L	T	S	0	0	0	1	0	1
ldlg	DA16$_S$,Rd,CND	CND				DA16$_S$												0	DA16$_S$				Rd					1	0	0	1	0	0
	(DO12$_S$,An),Rd,CND	CND				DO12$_S$												1	An				Rd					1	0	0	1	0	0
	(An,AU12$_S$)*,Rd,CND	CND				AU12$_S$												0	An				Rd					1	0	0	1	0	1
	(An,Ru)*,Rd,CND	CND				0	0	0	0	0	0	0	Ru					1	An				Rd					1	0	0	1	0	1
	(Rx,An),Rd,CND	CND				0	1	0	0	0	0	0	Rx					1	An				Rd					1	0	0	1	0	1
	(An)+,RGS	S	T	L	0	1	0	1	2	3	4	5	6	7	8	9	A	1	An				B	P	Q	U	V	1	0	0	1	0	1
	-(An),RGS	V	U	Q	P	1	1	B	A	9	8	7	6	5	4	3	2	1	An				1	0	L	T	S	1	0	0	1	0	1
ldss	DA16$_S$,Rd,CND	CND				DA16$_S$												0	DA16$_S$				Rd					0	1	1	1	0	0
	(DO12$_S$,An),Rd,CND	CND				DO12$_S$												1	An				Rd					0	1	1	1	0	0
	(An,AU12$_S$)*,Rd,CND	CND				AU12$_S$												0	An				Rd					0	1	1	1	0	1
	(An,Ru)*,Rd,CND	CND				0	0	0	0	0	0	0	Ru					1	An				Rd					0	1	1	1	0	1
	(Rx,An),Rd,CND	CND				0	1	0	0	0	0	0	Rx					1	An				Rd					0	1	1	1	0	1
	(An)+,RGS	S	T	L	0	1	0	1	2	3	4	5	6	7	8	9	A	1	An				B	P	Q	U	V	0	1	1	1	0	1
	-(An),RGS	V	U	Q	P	1	1	B	A	9	8	7	6	5	4	3	2	1	An				1	0	L	T	S	0	1	1	1	0	1
ldsz	DA16$_S$,Rd,CND	CND				DA16$_S$												0	DA16$_S$				Rd					0	1	0	1	0	0
	(DO12$_S$,An),Rd,CND	CND				DO12$_S$												1	An				Rd					0	1	0	1	0	0
	(An,AU12$_S$)*,Rd,CND	CND				AU12$_S$												0	An				Rd					0	1	0	1	0	1
	(An,Ru)*,Rd,CND	CND				0	0	0	0	0	0	0	Ru					1	An				Rd					0	1	0	1	0	1
	(Rx,An),Rd,CND	CND				0	1	0	0	0	0	0	Rx					1	An				Rd					0	1	0	1	0	1
	(An)+,RGS	S	T	L	0	1	0	1	2	3	4	5	6	7	8	9	A	1	An				B	P	Q	U	V	0	1	0	1	0	1
	-(An),RGS	V	U	Q	P	1	1	B	A	9	8	7	6	5	4	3	2	1	An				1	0	L	T	S	0	1	0	1	0	1
mfdp	Rd	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	Rd					0	0	0	0	0	0	0	1	0	0	0
mlcs	C16$_S$,Rs1,Rd	C16$_S$																Rd					Rs1					0	0	1	0	1	1
mlcu	C16$_U$,Rs1,Rd	C16$_U$																Rd					Rs1					0	0	0	0	1	1
mlhs	Rs0,Rs1,Rd,CND	CND				0	0	0	0	0	0	0	Rs0					Rd					Rs1					0	1	1	0	0	0
mlhu	Rs0,Rs1,Rd,CND	CND				0	0	0	0	0	0	0	Rs0					Rd					Rs1					0	1	0	0	0	0
move	Rs,Rd,CND	CND				0	1	0	0	0	0	0	Rs					Rd					0	0	0	0	0	0	0	0	0	0	0
	C17$_S$,Rd,CND	CND				C17$_S$												Rd					0	C17$_S$				1	0	C	0	0	1
mtdp	Rs	0	0	0	0	1	1	0	0	0	0	0	Rs					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
mult	Rs0,Rs1,Rd,CND	CND				0	0	0	0	0	0	0	Rs0					Rd					Rs1					0	0	0	0	0	0
negt	Rs,Rd,CND	CND				0	1	0	0	0	0	0	Rs					Rd					0	0	0	0	0	0	1	0	0	0	0
rsie	implied	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	0
rspc	implied	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	1	0
rtir	implied	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0
rtsr	implied	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0
scie	implied	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	1	0
shlf	SHC5$_U$,Rs1,Rd,CND	CND				1	0	0	0	0	0	0	SHC5$_U$					Rd					Rs1					0	1	0	0	0	0
	Rs0,Rs1,Rd,CND	CND				1	0	1	0	0	0	0	Rs0					Rd					Rs1					0	1	0	0	0	0
shlz	SHC5$_U$,Rs1,Rd,CND	CND				1	0	0	0	0	0	0	SHC5$_U$					Rd					Rs1					0	0	0	0	0	0
	Rs0,Rs1,Rd,CND	CND				1	0	1	0	0	0	0	Rs0					Rd					Rs1					0	0	0	0	0	0
shrs	SHC5$_U$,Rs1,Rd,CND	CND				1	0	0	0	0	0	0	SHC5$_U$					Rd					Rs1					0	1	1	0	0	0
	Rs0,Rs1,Rd,CND	CND				1	0	1	0	0	0	0	Rs0					Rd					Rs1					0	1	1	0	0	0
shru	SHC5$_U$,Rs1,Rd,CND	CND				1	0	0	0	0	0	0	SHC5$_U$					Rd					Rs1					0	0	1	0	0	0
	Rs0,Rs1,Rd,CND	CND				1	0	1	0	0	0	0	Rs0					Rd					Rs1					0	0	1	0	0	0
stbt	Rs,DA16$_S$,CND	CND				DA16$_S$												0	DA16$_S$				Rs					1	0	1	1	0	0
	Rs,(DO12$_S$,An),CND	CND				DO12$_S$												1	An				Rs					1	0	1	1	0	0
	Rs,(An,AU12$_S$)*,CND	CND				AU12$_S$												0	An				Rs					1	0	1	1	0	1
	Rs,(An,Ru)*,CND	CND				0	0	0	0	0	0	0	Ru					1	An				Rs					1	0	1	1	0	1
	Rs,(Rx,An),CND	CND				0	1	0	0	0	0	0	Rx					1	An				Rs					1	0	1	1	0	1
	RGS,(An)+	S	T	L	0	1	0	1	2	3	4	5	6	7	8	9	A	1	An				B	P	Q	U	V	1	0	1	1	0	1
	RGS,-(An)	V	U	Q	P	1	1	B	A	9	8	7	6	5	4	3	2	1	An				1	0	L	T	S	1	0	1	1	0	1
stie	implied	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0
stlg	Rs,DA16$_S$,CND	CND				DA16$_S$												0	DA16$_S$				Rs					1	1	0	1	0	0
	Rs,(DO12$_S$,An),CND	CND				DO12$_S$												1	An				Rs					1	1	0	1	0	0
	Rs,(An,AU12$_S$)*,CND	CND				AU12$_S$												0	An				Rs					1	1	0	1	0	1
	Rs,(An,Ru)*,CND	CND				0	0	0	0	0	0	0	Ru					1	An				Rs					1	1	0	1	0	1
	Rs,(Rx,An),CND	CND				0	1	0	0	0	0	0	Rx					1	An				Rs					1	1	0	1	0	1
	RGS,(An)+	S	T	L	0	1	0	1	2	3	4	5	6	7	8	9	A	1	An				B	P	Q	U	V	1	1	0	1	0	1
	RGS,-(An)	V	U	Q	P	1	1	B	A	9	8	7	6	5	4	3	2	1	An				1	0	L	T	S	1	1	0	1	0	1
stop	implied	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0
stsh	Rs,DA16$_S$,CND	CND				DA16$_S$												0	DA16$_S$				Rs					1	1	1	1	0	0
	Rs,(DO12$_S$,An),CND	CND				DO12$_S$												1	An				Rs					1	1	1	1	0	0
	Rs,(An,AU12$_S$)*,CND	CND				AU12$_S$												0	An				Rs					1	1	1	1	0	1
	Rs,(An,Ru)*,CND	CND				0	0	0	0	0	0	0	Ru					1	An				Rs					1	1	1	1	0	1
	Rs,(Rx,An),CND	CND				0	1	0	0	0	0	0	Rx					1	An				Rs					1	1	1	1	0	1
	RGS,(An)+	S	T	L	0	1	0	1	2	3	4	5	6	7	8	9	A	1	An				B	P	Q	U	V	1	1	1	1	0	1
	RGS,-(An)	V	U	Q	P	1	1	B	A	9	8	7	6	5	4	3	2	1	An				1	0	L	T	S	1	1	1	1	0	1
subc	Rs0,Rs1,Rd,CND	CND				0	0	1	0	0	0	0	Rs0					Rd					Rs1					0	1	0	0	0	0
	C12$_U$,Rs1,Rd,CND	CND				C12$_U$												Rd					Rs1					0	1	0	0	0	1

subf	Rs0,Rs1,Rd,CND	CND				0	0	1	0	0	0	0	Rs0					Rd					Rs1					0	0	0	0	0	0
	C12$_U$,Rs1,Rd,CND	CND				C12$_U$												Rd					Rs1					0	0	0	0	0	1
	C16$_U$,Rs1,Rd	C16$_U$																Rd					Rs1					0	0	0	0	1	0
svpc	implied	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0
xorb	Rs0,Rs1,Rd,CND	CND				0	0	1	0	0	0	0	Rs0					Rd					Rs1					1	1	1	0	0	0
	C16$_U$,Rs1,Rd	C16$_U$																Rd					Rs1					1	1	1	0	1	0

The next table explains the color coding used in the opcode table above.

Color	Description of table entries
🟦	Register select field, selects a register of the programming model
🟧	Constant field
🟨	Fixed coded bits used to distinguish between instruction groups and individual instructions within groups

The next table defines the bit assignments of register select and constant fields in the opcode table. The left column contains the names of one or more register select or constant fields. If there are more fields separated by semicolons then all of these fields have the same format. The right 32 columns define how the multi-bit fields from the left column are mapped into 32-bit opcodes. For all left column fields except **RGS** the numbers given in the opcode columns define the bit positions and bit ordering of the multi-bit field(s) specified in the left column.

RGS is a special case. 19 bits of the opcode marked with single-characters represent the 19 possible registers of a register selection. Bits that are set are part of the register selection bits that are cleared are not part of the register selection. The single character markings relate to registers in the following way:

- Bits marked **0** to **B** represent registers **R0** – **RB**
- Bits marked **P** to **V** represent registers **RP** – **RV**
- The bit marked **L** represents register **LC**
- The bit marked **T** represents register **TA**
- The bit marked **S** represents register **SA**

Note that the **RGS** coding is different (reversed) for the **(An)+** and **-(An)** addressing modes

Some fields like e.g. the **C16$_U$** and **C16$_S$** have multiple coding options. The opcode table defines which coding type is used with each instruction.

Opcode fields that specify instruction addresses or offsets of instruction addresses are two bit shorter than the width specified in the left table column. This is because instruction addresses are specified in bytes but point to 32-bit word boundaries in the instruction memory. The two LSBs therefore are always zero and are not coded.

Opcode field	opcode bits																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs,Rs0,Rx,Ru												4	3	2	1	0																
Rd																	4	3	2	1	0											
An																		3	2	1	0											
Rs,Rs1,Rd																						4	3	2	1	0						
RGS for (An)+	S	T	L	0			1	2	3	4	5	6	7	8	9	A						B	P	Q	U	V						
RGS for –(An)	V	U	Q	P			B	A	9	8	7	6	5	4	3	2						1	2	L	T	S						
DA16$_S$					11	10	9	8	7	6	5	4	3	2	1	0		15	14	13	12											
DO12$_S$,AU12$_S$,C12$_U$					11	10	9	8	7	6	5	4	3	2	1	0																
C16$_U$, C16$_S$	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																
					11	10	9	8	7	6	5	4	3	2	1	0							15	14	13	12						
C17$_S$	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0													16			
					11	10	9	8	7	6	5	4	3	2	1	0							15	14	13	12			16			
CND[3:0]	3	2	1	0																												
SHC5$_U$,BTI5$_U$												4	3	2	1	0																
IO16$_S$									15	14	13	12	11	10	9	8	7	6	5	4	3	2										
S																							S									
IA29$_U$	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2					