RACORS

sf16

Family of
16-bit microprocessors

Base ISA Reference Manual

Revision 1.0
05 December 2013

Author: Martin Raubuch

Revision History

Revision	Date	
0.9	13Dec2012	First version, largely derived from the sf32b ISA reference manual
1.0	05Dec2013	Review with typo fixed, property note changed to RACORS GmbH

Table of contents

1 Overview

1.1 Introduction

The sf16 family of 16-bit microprocessors is targeted at embedded control applications that have high performance requirements and are satisfied with a direct addressable data space of 64kBytes. With fixed length 16-bit instruction coding the architectural focus is on high clock rates and small core implementations.

Besides the base (b) ISA defined in this manual the family includes a (d) DSP ISA extension for small 16-bit DSP applications. The (d) DSP ISA extension is defined in a separate manual.

The base ISA is a 16-bit general purpose load/store architecture. Accesses to memory data operands and computations are decoupled by using separate instructions. Memory operands are accessed by 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 sf16b's main features

- Load/store architecture
- Harvard architecture with separate instruction and data address spaces
- 128 kBytes instruction address space and 64 kBytes data address space
- Fixed length 16-bit instruction coding
- Eight 16-bit general purpose registers and eight special registers
- Support for 8-bit and 16-bit signed and unsigned integer data types
- Instructions to support higher precision operands > 16 bits
- Rich set of load/store addressing modes
- Bit manipulation & test instructions: set, clear, toggle & test
- 16*16 multiply instructions with either 16-bit high word or 16-bit low word results
- 16 interrupts with programmable start addresses
- Flexible debug support for application optimized debug concepts
- 16-bit loop counter

1.3 Scope of this manual

This sf16 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
- 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 sf16 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

Reset, Interrupts & Debug Support, defines the reset state, interrupt concept and software debug support concept.

Operand Types, defines bit accurate details of how operands are generated or calculated, defines operand addressing modes

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 eight general purpose registers **R0**, **R1**, **R2**, **R3**, **R4**, **R5**, **R5** or **R7**.
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.
Rb	one of registers **Rn** used as both source and destination operand. In addressing modes with two source operands **Rb** is source operand 1.
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**.
SRn	one of the eight special registers **CC**, **CS**, **LC**, **AU**, **SP**, **TA**, **SA** or **ID**.
SRs	one of the special registers **SRn** used as source operand
SRd	one of the special registers **SRn** used as destination operand
SRLd	one of the low order special registers **CC**, **CS**, **LC** or **AU** used as destination operand
An	one of the four registers **SP**, **TA**, **R6** or **R7** used as indirect memory address in addressing modes with memory source or destination operands.
Ad	one of registers **An** used as destination operand.
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**, **TA**, **SA**.

2.2 Constant Specifications

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

Acronyms for constants with a value range have an optional one or two-character suffix. The first character has the following meaning: **U** (Unsigned) or **S** (Signed) or **A** (Asymmetric). The second character **N** means: Not including zero.

$C7_{U}$	7-bit constant (unsigned) used as source operand of move instructions; legal values are from 0 to 127.
$C7_{UN}$	7-bit constant (unsigned, not including zero) used as source operand of move instructions; legal values are from 1 to 128.
$C7_{SN}$	7-bit constant (signed, not including zero) used as source operand of move and computation instructions; legal values are from -64 to -1 and from 1 to 64.
$C8_{U}$	8-bit constant (unsigned,) used as source operand of computation instructions; legal values are from 0 to 255.
$C8_{UN}$	8-bit constant (unsigned, not including zero) used as source operand of computation instructions; legal values are from 1 to 256.

$C8_A$	8-bit constant (asymmetric) used as source operand of compare instructions; legal values are from -64 to -1 and from 0 to 255.
$C9_S$	9-bit constant (signed) used as source operand of move instructions; legal values are from -256 to 255.
C16	16-bit constant used as source operand with the **addh** instruction, legal values are from 0x0000 to 0xFF00; bits [7:0] are not coded and are always zero.
$DO5_S$	5-bit scaled data address offset (signed) used in an indirect memory addressing mode. Legal values are from -16 to 15 for byte accesses and from -32 to 30 (even values only) for short accesses.
DA8	8-bit scaled direct data address used in the direct memory addressing mode, Legal values are from 0 to 255 for byte accesses and from 0 to 510 (even values only) for short (16-bit) accesses.
SHC4	4-bit shift count used in addressing modes for shift instructions. Legal values are from 0 to 15
BTI4	4-bit bit index used in addressing modes for bit-manipulation instructions, legal values are from 0 to 15
IAH4	4-bit direct instruction address high; used with a special flow instruction to preset the high bits of a 16-bit direct instruction address. Legal values are from 1 to 15.
$IO8_S$	8-bit instruction address offset (signed) used with branch instructions. Legal values are from -128 to 127
IA12	12-bit direct instruction address; used in an addressing mode for jump and jump to subroutine instructions. Legal values are from 0x0000 to 0x0FFF.

2.3 Miscellaneous definitions

opcode	operation code of an instruction; contains sub codes that specify the instruction type and the operands. The sf16 has fixed length 16-bit **opcodes** stored in the instruction memory
eda	effective data address, a 16-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 16-bit word address that points to a 16-bit **opcode** word in the instruction address space.

3 Programming model

3.1 Instruction address space

3.1.1 Size and addressing scheme

The sf16 processors have a 128kBytes instruction address space. Instruction addresses are 16 bits and point to 16-bit opcode words in the instruction memory.

3.1.2 Endianess

The sf16 implements a little endian scheme to map 16-bit opcodes to memory words. In case the instruction interface is wider than 16 bits (e.g. 32-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 sf16 processors have a 64kBytes data address space. Data addresses are 16 bits and point to byte locations in the data memory. The base ISA supports byte (8-bit) and short (16-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 processor all arithmetic is done on 16-bit operands. Byte operands are zero-extended to 16 bits when loaded from memory into one of the general purpose registers. When a signed byte operand is loaded from memory an **`sxbt`** (sign-extend byte) instruction must follow to make sure the register value represents the correct 16-bit 2's complement format of the signed byte value. When register operands are stored to memory they are truncated to the size of the destination operand. When storing a byte value to memory the 8 MSBs of the source register are discarded.

3.2.3 Alignment

The sf16 processors do not handle misaligned memory operands internally. For 16-bit accesses the LSB is ignored. However the full data space address including the LSB 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 sf16 implements a little endian scheme to map 8-bit and 16-bit data operands to memory words.

3.2.5 Summery table

The table below illustrates the mapping of data operands into 16-bit memory words. All operands are aligned to memory words and to their own size.

16-bit Memory words	3		2		1		0	
Memory addresses	n+6		n+4		n+2		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

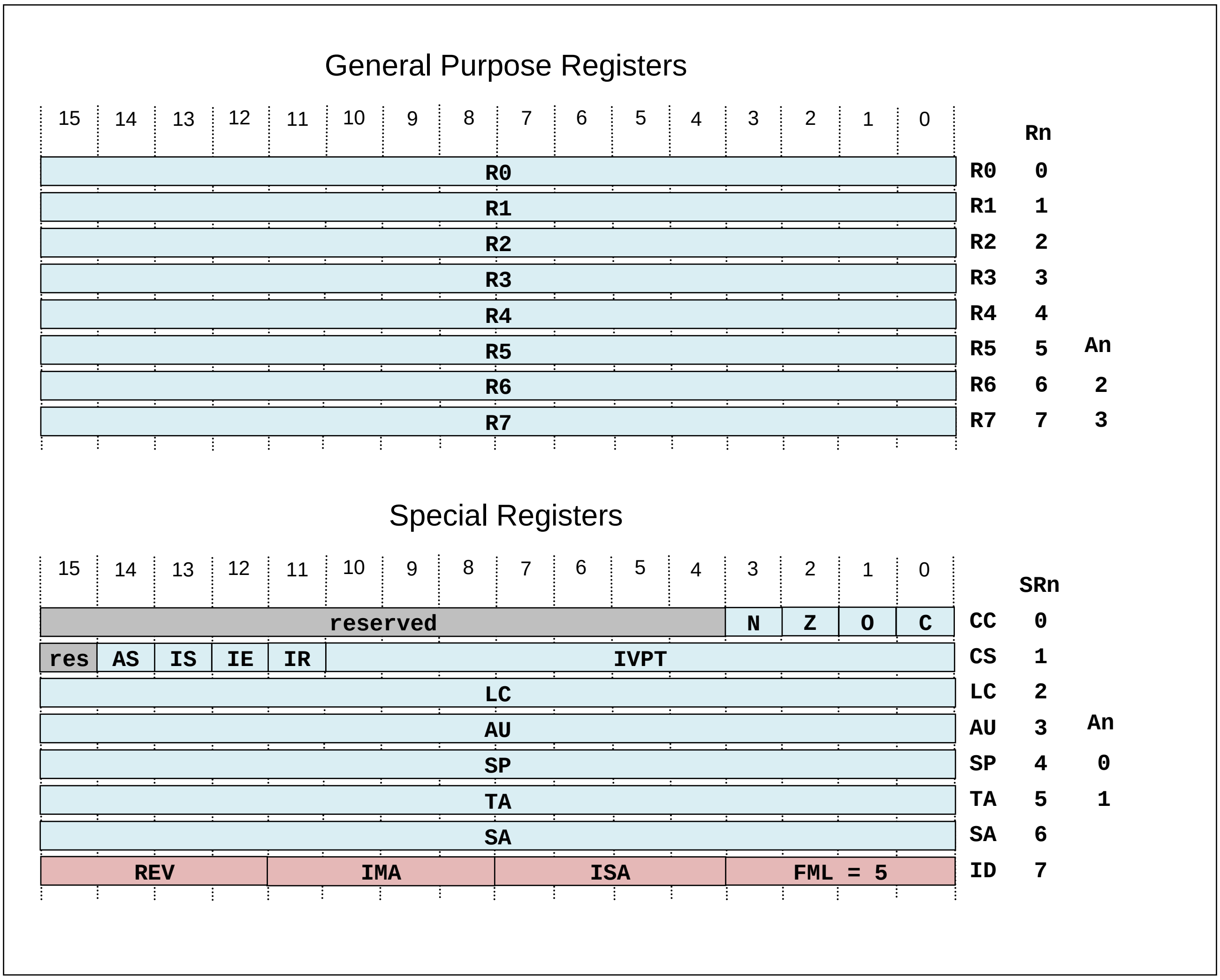
Register values are represented with the LSB at the right most bit position and the MSB at the left most bit position. For an n-bit register the LSB is bit number 0 and the MSB is bit number n-1.

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 sf16 Registers



3.3.3 Register Details

The sf16 has two register spaces referred to as **Rn** (General Purpose Registers) and **SRn** (Special Registers). Individual registers of these spaces are addressed by 3-bit fields in instruction opcodes. A third logical group **An** is defined that contains special registers **SP** and **TA** and general purpose registers **R6** and **R7**. Registers of the **An** group can be used as indirect address and are addressed by 2-bit fields in instruction opcodes.

The general purpose registers **Rn** can be used as source or destination operands of any computation or load/store/move instruction. General purpose registers can also be used as index operand in memory addressing modes.

The special registers **SRn** have dedicated functions and are implicitly used as source and/or destination of certain instructions. Beyond these dedicated functions they cannot be used as source or destination of computation instruction. Dedicated move instructions are available to transfer values from a general purpose register and vice versa. Some special registers can be source or destination of load/store instructions (as part of a register selection), some can be used as indirect address and some can be loaded directly with a constant value.

A special case is the **SA** register which represents two underlying physical registers. When the **AS** flag in register **CS** is clear the actual **SA** physical register (Subroutine Address) is accessed. When the **AS** flag in register **CS** is set the interrupt address register is accessed. This hidden register is used to store the return address when an interrupt is started.

The table below summarizes the sf16 register properties. The paragraphs following the table provide detailed information of register groups and individual registers.

	R0-R7	CC	CS	LC	AU	SP	TA	SA	ID
can be source or destination of computation instr.	yes	no	no	no	no	no	no	no	no
can be used as source of a move instruction	yes	yes	yes	yes	yes	yes	yes	yes	yes
can be used as destination of a move instruction	yes	yes	yes	yes	yes	yes	yes	yes	no
can be used as indirect data address	**R6**, **R7**	no	no	no	no	yes	yes	no	no
can be used as source/dest. of load/store	yes	no	no	no	no	yes	yes	no	no
can be used as indirect memory address index	yes	no	no	no	no	no	no	no	no
can be part of an RGS (Register Selection)	**R2**-**R6**	no	no	no	no	yes	yes	no	no
can be moved directly to the debug port	yes	no	no	no	no	no	no	no	no
can be loaded directly from the debug port	yes	no	no	no	no	no	no	no	no

R0-R7 Eight 16-bit general purpose registers intended for computation operands

CC Condition Code; this 4-bit register contains the condition code flags **C**,**O**,**Z** and **N**. **CC** is an implicit source operand of conditional branch instructions; **CC** is an implicit 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 15 to bit 16 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 15 to bit 16 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 16-bit result of the operation is zero (all 16 bits zero) and is cleared otherwise.

CC.N Negative flag; the **N** flag is set by instructions that update the **CC** register if the 16-bit result of the operation is negative (bit 15 set) and is cleared otherwise.

CS 32-bit Control and Status; This 16-bit register contains a number of control and status flags and also the pointer to the interrupt vector table in the data address space; when **CS** is used as destination register of move instructions only the **IVTP** field is updated with the corresponding bits of the destination operand the **IR**, **IE**, **IS** and **AS** flags remain unchanged

CS.IVTP Interrupt Vectors Table Pointer; this 11-bit field defines the most significant bits of the 16-bit start address of the interrupt vector table in the data address space. The table is aligned on a 32 bytes boundary. The five LSBs of the 16-bit table address are all zeros and are not contained in the **CS** register. When the sf16 starts an interrupt service routine it fetches the start address of the routine from the table pointed to by **IVTP**

CS.IR Interrupt status flag; this flag is set when the sf16 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.AS Address Select; this flag determines if the physical **SA** register (**AS**=0) or the hidden

	interrupt return address register (**AS**=1) is accessed when a move to or from the **SA** special register is executed; load/store instructions to/from the **SA** register number always access the **SA** physical register; dedicated instructions are available to set and clear the **AS** flag
LC	16-bit loop counter; used as loop counter with the **`brlc`** (loop counter branch) instruction to improve code density and performance of inner loops
AU	16-bit address update; used with the **`(An)*`** addressing mode for load/store instructions. With this addressing mode the **AU** register is added to the indirect address register after the memory access
SP	16-bit stack pointer; is part of the **An** register group and can be used as indirect data address; the **`–(An)`** and **`(An)+`** addressing modes which enable push/pop operations and all other memory access addressing modes can be used with any of the four registers of the **An** group so in principal each of the four **An** registers can be used as stack pointer; the unique feature of **SP** is the **`adsp`** (add to stack pointer) instruction; it adds a 7-bit signed constant to the address in **SP** and stores the result in any of the four **An** registers; this can be used to allocate/de-allocate stack space at the beginning and end of sub-routines (**SP** destination) or to move pointers to data objects on the stack to one of the other **An** registers.
TA	16-bit target address; used as instruction address for indirect jump and jump to subroutine instructions; **TA** is also part of the **An** register group and can be used as indirect data address; if used as indirect instruction address **TA** points to a 16-bit opcode word in the 128kBytes instruction address space; if used as indirect data address **TA** points to a byte location in the 64kBytes data address space.
SA	16-bit subroutine return address; points to a 16-bit opcode word in the 128kBytes instruction address space; 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; **SA** can be accessed by **`mtsr`** and **`mfsr`** instructions only when **CS.AS** is zero.
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	Family; this 4-bit code identifies the core family. The code for the sf16 is 5; this code is to distinguish the processor from other architectures e.g. from processors of the eco32, eco16 and sf32 families.
ID.ISA	Instruction Set Architecture; this 4-bit code identifies the processor's **ISA**; the following **ISA** codes are defined for the sf16: 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); the **IMA** code 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

3.3.4 Hidden Registers

The sf16 base ISA has four additional hidden registers ***IRA***, ***CCS***, ***IAH*** and ***IAHS***. The hidden registers are a mandatory part of the programming model but are not contained in the **Rn** or **SRn** groups. For easy distinction from the **Rn** and **SRn** registers hidden register names are printed in italic letters. The following paragraphs are detailed descriptions of the hidden registers.

IRA	16-bit interrupt return address; points to a 16-bit opcode word in the 128kBytes instruction address space; when an interrupt is started the return address (address of the next instruction following the last instruction executed before the interrupt) is stored in ***IRA***; when an **`rtir`** (return from interrupt) instruction is executed the content of ***IRA*** is the effective instruction address where program execution continues; ***IRA*** can be accessed by **`mtsr`** and **`mfsr`** instructions via the **SA** special register number when **CS.AS** = 1; at least after reset ***IRA*** must be set once by an **`mtsr`** instruction before the interrupt state is left by an **`rtir`** instruction to make sure instruction execution continues at a deterministic address (***IRA*** is undefined after reset)

CCS	4-bit Condition Code Shadow register; this register is used to save the state of the **CC** register when an interrupt is started; at the end of interrupt service routines (execution of an **`rtir`** instruction) **CC** is restored from ***CCS***
IAH	4-bit Instruction Address High; this register is used to extend the direct addressable instruction address space from 12 bits (4k instructions) to 16 bits (64k instructions); the **`siah`** instruction sets the ***IAH*** register to the 4-bit constant contained in the opcode; the effective instruction address **eia** of **`jump`** and **`jpsr`** instructions with the **`IA12`** addressing mode is the concatenation of the ***IAH*** register as **eia**[15:12] and the **`IA12`** constant contained in the opcode as **eia**[11:0]; **`jump`** and **`jpsr`** instructions with the **`IA12`** addressing mode clear ***IAH*** to zero after the instruction fetch; ***IAH*** is cleared to zero when the processor is reset; with no preceding **`siah`** instruction **`jump`** and **`jpsr`** instructions with the **`IA12`** addressing mode can only reach the instruction address range from **`0x0000-0x0FFF`** (***IAH***=0); ***IAH*** is saved in ***IAHS*** and then cleared at the beginning of interrupts; the original value is restored from ***IAHS*** at the end of interrupts
IAHS	4-bit Instruction Address High Shadow register; this register is used to save the state of the **IAH** register when an interrupt is started; at the end of interrupt service routines (execution of an **`rtir`** instruction) ***IAH*** is restored from ***IAHS***

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 general purpose 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.

DA8	8-bit absolute, scaled data address
($DO5_S$,An)	Indirect data address with 5-bit signed, scaled offset
(Rx,An)	Indirect data address with scaled index
(An)+	Indirect data address with scaled post-increment
-(An)	Indirect data address with scaled pre-decrement
(An)*	Indirect data address with un-scaled post-update

4.1.2 Registers only addressing modes

Rs	Single register, **Rs** = source operand
Rd	Single register, **Rd** = destination operand
Rs,Rd	Dual registers, **Rs** = source operand, **Rd** = destination operand
Rs0,Rs1	Dual registers, **Rs0** = source operand 0, **Rs1** = source operand 1
SRs,Rd	Dual registers, **SRs** = source operand, **Rd** = destination operand
Rs,SRd	Dual registers, **Rs** = source operand, **SRd** = destination operand
Rs0,Rs1,Rd	Triadic registers, **Rs0** = source operands 0, **Rs1** = source operand 1, **Rd** = destination operand

4.1.3 Registers and constants addressing modes

$C7_U$,CC	Constant and single register, **$C7_U$** = source operand, **CC** = destination operand
$C7_U$,CS	Constant and single register, **$C7_U$** = source operand, **CS** = destination operand
$C7_{UN}$,LC	Constant and single register, **$C7_{UN}$** = source operand, **LC** = destination operand
$C7_{SN}$,AU	Constant and single register, **$C7_{SN}$** = source operand, **AU** = destination operand
$C7_S$,Ad	Constant and single register, **$C7_S$** = source operand 0, **SP** = source operand 1, **Ad** = destination operand
$C8_U$,Rb	Constant and single register, **$C8_U$** = source operand 0, **Rb** = source operand 1 and destination operand
$C8_{UN}$,Rb	Constant and single register, **$C8_{UN}$** = source operand 0, **Rb** = source operand 1 and destination operand
$C8_A$,Rs1	Constant and single register, **$C8_A$** = source operand 0, **Rs1** = source operand 1
$C9_S$,Rd	Constant and single register, **$C9_S$** = source operand, **Rd** = destination operand
C16,Rb	Constant and single register, **C16** = source operand 0, **Rb** = source operand 1 and destination operand
SHC4,Rb	Constant and single register, **SHC4** = source operand 0, **Rb** = source operand 1 and destination operand
BTI4,Rb	Constant and single register, **BTI4** = source operand 0, **Rb** = source operand 1 and destination operand

4.1.4 Instruction memory addressing modes

IA12	12-bit absolute instruction address
$IO8_S$	8-bit signed instruction address offset
IAH4	4-bit MSBs (bits[15:12]) of a 16-bit instruction address

4.1.5 Miscellaneous addressing modes

implied	operands are implicitly defined, there are two instruction categories: the first category (interrupt enable, address select) 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 general purpose 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

ldbt	load byte (8-bit word) from memory and zero-extend to 16 bits
ldsh	load short (16-bit word) from memory
stbt	store byte (8-bit) to memory
stsh	store short (16-bit) to memory

	ldbt	ldsh	stbt	stsh
DA8	X	X	X	X
(DO5$_S$,An)	X	X	X	X
(Rx,An)	X	X	X	X
(An)+	X	X	X	X
-(An)	X	X	X	X
(An)*	X	X	X	X

4.2.2 Move

move	move register to register or constant to register
mfsr	move from special register (to general purpose register)
mtsr	move to special register (from general purpose register)
mfdp	move from debug port (to general purpose register)
mtdp	move to debug port (from general purpose register)

	move	mfsr	mtsr	mfdp	mtdp
Rs,Rd	X				
C9$_S$,Rd	X				
SRs,Rd		X			
Rs,SRd			X		
C7,SRLd			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
adcf	add carry flag to register
addh	add 16-bit constant to register
adsp	add to stack pointer
subf	subtract register from register or constant from register
subc	subtract with carry register from register or constant from register
sbcf	subtract carry flag from register
comp	compare register to register or constant to register
cmpc	compare with carry register to register or constant to register
cpcf	compare carry flag 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

sxbt sign extend byte
sxsh sign extend short

	addt	addc	adcf	addh	adsp	subf	subc	sbcf	comp	cmpc	cpcf	negt	absl	clzr	sxbt	sxsh
Rs0,Rs1,Rd	X	X				X	X									
C8$_{UN}$,Rb	X					X										
C16,Rb				X												
Rs0,Rs1									X	X						
C8$_{A}$,Rs1									X							
Rs,Rd			X					X				X	X	X	X	X
Rs											X					
C7$_{SN}$,Ad					X											
CC update	*	*	*			*	*	*	*	*	*					

4.2.4 Multiplies

mult multiply registers * register, 16*16 -> 32-bit, stores 16-bit low word result
mlhu multiply high unsigned, 16*16 -> 32-bit, stores 16-bit high word result
mlhs multiply high signed, 16*16 -> 32-bit, stores 16-bit high word result

	mult	mlhu	mlhs
Rs0,Rs1,Rd	X	X	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
invt invert (1's complement, invert) from register to register

	andb	iorb	xorb	invt
Rs0,Rs1,Rd	X	X	X	
C8$_{U}$,Rs1,Rd	X	X		
Rs,Rd				X
CC update	*			

4.2.6 Shift

shlz shift left with zero fill, constant or indirect shift count from 0 to 15
shlf shift left with feedback (rotate), constant or indirect shift count from 0 to 15
shru shift right unsigned, constant or indirect shift count from 0 to 15
shrs shift right signed, constant or indirect shift count from 0 to 5

	shlz	shlf	shru	shrs
SHC4,Rb	X	X	X	X
Rs0,Rs1,Rd	X	X	X	X

4.2.7 Bit manipulation

btst bit set, constant or indirect bit index from 0 to 15
btcl bit clear, constant or indirect bit index from 0 to 15
bttg bit toggle, constant or indirect bit index from 0 to 15
btts bit test, constant or indirect bit index from 0 to 15

	btst	btcl	bttg	btts
BIT4,Rs1,Rd	X	X	X	
BTI4,Rs				X
Rs0,Rs1,Rd	X	X	X	
Rs0,Rs1				X
CC update				*

4.2.8 Flow control

jump jump, continue program execution at specified target address
jpsr jump to subroutine
siah set instruction address high
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
stas set address select
clas clear address select

	jump	jpsr	siah	rtsr	rtir	brlc	brxx	stie	clie	scie	rsie	stas	clas
implied	X	X		X	X			X	X	X	X	X	X
IA12	X	X											
$IO8_S$						X	X						
IAH4			X										

4.2.9 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 Reset, Interrupts & Debug-Support

5.1 Reset

5.1.1 Program start address

The processor input signal **IRN**[3:0] and the output signal **IA**[15:0] 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**[15:0] of the first instruction fetch after a reset. All other bits of **IA**[15:0] are zero. In summery the instruction address **IA**[15:0] of the first instruction fetch after a reset is **IA**[15:12] = **IRN**[3:0], **IA**[11:0] = 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.

5.1.2 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

CS.**IE** = 0, interrupts are disabled

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

CS.**AS** = 0, the address select bit is clear (**SA** access to the **SA** physical register)

CC = 0, the condition code flags are all cleared

CCS = 0, Condition Code Shadow (hidden register)

IAH = 0, Instruction Address High (hidden register)

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.

5.2 Interrupts

5.2.1 Overview

The sf16 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 processors 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 32 bytes window of the processor's data address space. The 11-bit field **IVTP** of special register **CS** defines the start address of the table. **IVTP** defines the higher 11 bits of the 16-bit table address. The five least significant bits of the table address are zero. This implies that the interrupt vector table is aligned on a 32 bytes boundary. The table contains 16 entries of 16-bit size. Each entry is a 16-bit instruction address.

When an interrupt is started the instruction address of the next instruction following the last instruction executed before the interrupt is stored in the hidden interrupt return address register ***IRA***. The state of the **CC** register is stored in the hidden condition code shadow register ***CCS***. The hidden Instruction Address High register ***IAH*** is stored in ***IAHS*** and then cleared. When an `rtir` (return from interrupt) instruction is executed the original values of **CC** and ***IAH*** are restored and program execution continues at the address in ***IRA***.

Writing to ***IRA*** can be done using `mtsr Rs,SA` instructions when **CS**.**AS** is set. It 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 reading and writing the register to save, redirect and restore interrupt return addresses in cases of task switches and system calls.

Beside **CC**, ***IAH*** and and the instruction address the sf16 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.

5.2.2 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 saved in hidden register ***IRA***, **CC** and **IAH** are saved in their corresponding shadow registers and then ***IAH*** is cleared to zero. While executing instructions of the interrupt service routine **CS**.**IR** is set. When an **`rtir`** (return from interrupt) instruction is executed at the end of the interrupt service routine **CC** and ***IAH*** are restored from their corresponding shadow registers and execution of the interrupted code sequence continues at the address in ***IRA***.

Clearing of ***IAH*** at the beginning of interrupt routines is necessary to make sure the concept of 12-bit direct instruction addresses using **`jump`**/**`jpsr`** instructions with the **`IA12`** addressing mode directly and 16-bit addresses using a preceding **`siah`** instruction is working correctly also in interrupt routines. The original value of ***IAH*** is restored at the end of the interrupt to make sure an interrupted sequence of **`siah`** and **`jump`**/**`jpsr`** instructions with the **`IA12`** addressing mode is not corrupted.

5.3 Debug Support

5.3.1 Overview

The processors of the sf16 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 16-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 16-bit value from the debug input port to a register **Rn**
- Transfer the program counter value to the debug output port
- Transfer a 16-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 the 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 16-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)

5.3.2 Debug Port

The debug port consists of the following signals:

`DBI[15:0]`	Debug In, 16-bit data input
`DBO[15:0]`	Debug Out, 16-bit data output
`STRQ`	Stop Request, 1-bit control input
`INJI`	Inject Instruction, 1-bit control input
`STPD`	Stopped, 1-bit control output

5.3.3 Debug Instructions

The following dedicated instructions are part of the sf16 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 16-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 16-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.
`stop`	stop, the processor stops fetching new instructions and enters the **stopped** state when all pending instructions are retired.

5.3.4 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.

5.3.4.1 Instruction breakpoints

The instruction that should cause the break point is replaced by 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 by 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 instruction 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.

5.3.4.2 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.

5.3.4.3 Show register content

When the processor is in the **stopped** mode the debug module injects **`mtdp`** instructions to read the content of general purpose registers. To read a special register first an **`mfsr`** instruction is injected to copy the special register to a general purpose register. An **`mtdp`** instruction then transfers the general purpose register content to the debug module.

5.3.4.4 Modify register content

When the processor is in the **stopped** mode the debug module injects **`mfdp`** instructions to change the content of general purpose registers. To modify a special register the value is first written to a general purpose register by injecting an **`mfdp`** instruction. The value is then transferred to the special register by injecting an **`mtsr`** instruction.

5.3.4.5 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.

5.3.4.6 Modify memory content

For memories that the processor can access through 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.

5.3.4.7 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 input port. The debug module then de-asserts the **STRQ** signal. The processor leaves the **stopped** state and starts program execution from the injected address.

5.3.4.8 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 stopped, 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 input port. 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 or two-character suffix with the following meaning:

First character: **U** (Unsigned), **S** (Signed) or **A** (Asymmetric)

Second Character: **N** (Not Including Zero)

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:

- **`uint4`** type: 4-bit unsigned integer
- **`uint16`** type: 16-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 and 2*n for short **`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 16-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. Constant operands are transformed into source operands of instructions.

$\mathtt{C7}_{\mathtt{U}}$ **`7-bit constant (Unsigned)`**

The 7-bit field **$\mathtt{C7}_{\mathtt{U}}$** is extracted from the opcode word and zero-extended to the 16-bit source operand **`src`**. The value range is [0,127].

Used with instruction **`mtsr`** (with **CC** or **CS** destination)

C language description

```
uint16 src;
src = C7U;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$\mathtt{C7}_{\mathtt{U}}$			5	4	3	2	1	0	6							

$C7_{UN}$ 7-bit constant (Unsigned, Not including zero)

The 7-bit field **$C7_{UN}$** is extracted from the opcode word and zero-extended to 16-bits. If the field value is zero the source operand **src** is set to 128. If not the zero-extended field value becomes the source operand **src**. The value range is [1,128].

Used with instruction: **mtsr** (with **LC** destination)

C language description

```
uint16 src;
src = C7UN == 0 ? 128 : C7UN;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$C7_{UN}$			5	4	3	2	1	0	6							

$C7_{SN}$ 7-bit constant (Signed, Not including zero)

The 7-bit field **$C7_{SN}$** is extracted from the opcode word and sign-extended to 16-bits. If the field value is zero the source operand **src** is set to 64. If not the sign-extended field value becomes the source operand **src**. The value range is [-64,1] and [1,64].

Used with instructions: **mtsr** (with **AU** destination)

C language description

```
uint16 src;
src = C7SN == 0 ? 64 : (C7UN & 0x40 ? FF80 | C7UN : C7UN);
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$C7_{SN}$			5	4	3	2	1	0	6							

$C8_{U}$ 8-bit constant (Unsigned)

The 8-bit field **$C8_{U}$** is extracted from the opcode word and zero-extended to the 16-bit source operand **src**. The value range is [0,255].

Used with instructions: **andb, iorb**

C language description

```
uint16 src;
src = C8U;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$C8_{U}$	7	6	5	4	3	2	1	0								

$C8_{UN}$ 8-bit constant (Unsigned, Not including zero)

The 8-bit field **$C8_{UN}$** is extracted from the opcode word and zero-extended to the 16-bit source operand **src**. If the field value is zero the source operand **src** is set to 256. The value range is [1,256].

Used with instructions: **addt, subf**

C language description

```
uint16 src;
src = C8UN == 0 ? 256: C8UN;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$C8_{UN}$	7	6	5	4	3	2	1	0								

$\mathbf{C8_A}$ — **8-bit constant (Asymmetric)**

If the LSB of the opcode word is set the 8-bit field $\mathbf{C8_U}$ is extracted from the opcode word and zero-extended to the 16-bit source operand **src**. If the LSB of the opcode word is clear the 8-bit bit field $\mathbf{C8_U}$ is extracted from the opcode word and the lower 6 bits are converted to the negative 16-bit source operand **src**. The value range is [-64,255].

Used with instruction: **comp**

C language description

```
uint16 src;
src = opcode & 1 ? C8_A : 0xFFc0 | (C8_A & 0x3F);
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$\mathbf{C8_A}$	7	6	5	4	3	2	1	0								

$\mathbf{C9_S}$ — **9-bit constant (Signed)**

The 9-bit field $\mathbf{C9_S}$ is extracted from the opcode word and sign-extended to the 16-bit source operand **src**. The value range is [-256,255].

Used with instruction: **move**

C language description

```
uint16 src;
src = C9_S & 0x100 ? 0xFE00 | C9_S : C9_S;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$\mathbf{C9_S}$	7	6	5	4	3	2	1	0				8				

C16 — **16-bit constant**

The 8-bit field **C16** is extracted from the opcode word and becomes the 16-bit source operand **src**. The value range is [0x0000,0xFF00]. Bits [7:0] of the constant are always zero and are not coded.

Used with instruction: **addh**

C language description

```
uint16 src;
src = C16;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C16	15	14	13	12	11	10	9	8								

SHC4 — **4-bit shift count**

The 4-bit field **SHC4** is extracted from the opcode word and becomes the source operand **src**. The value range is [0,15].

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

C language description

```
uint4 src;
src = SHC4;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHC4			3	2	1	0										

BTI4 — **4-bit bit index**

The 4-bit field **BTI4** is extracted from the opcode word and becomes the source operand **src**. The value range is [0,15]. The bit index is counted from the LSB (**BTI4** = 0) to the MSB (**BTI4** = 15). The bit index operand is used to address individual bits of registers **Rn**.

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

C language description

```
uint4 src;
src = BTI4;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BTI4			3	2	1	0										

6.3 Register operand types

Register operands are contained in one of the eight general purpose registers **Rn** or in one of the eight special registers **SRn**. They can be either source or destination operands. Bit fields in the instruction opcode determine which register of the **Rn**, **SRn** or **An** 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 16-bit source operand **src**. The **Rs** operand type is used with instructions that have a single source operand.

C language description

```
uint16 src;
src = Rs;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs			2	1	0											

Rs0

Rn register used as source 0

The content of register **Rs0** is the 16-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 **src1** – **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

```
uint16 src0;
src0 = Rs0;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs0			2	1	0											

Rs1

Rn register used as source 1

The content of register **Rs1** is the 16-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** – **src0**). If used with shift, bit manipulation or bit-field instructions **Rs1** is the data source operand.

C language description

```
uint16 src1;
src1 = Rs1;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs1						2	1	0								

Rd

Rn register used as destination

The 16-bit destination operand **dst** is stored in register **Rd**.

C language description

```
uint16 dst;
Rd = dst;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd									2	1	0					

Rb

Rn register used as both source 1 and destination

The content of register **Rb** is the 16-bit source operand **`src1`**. The **Rb** operand type is used with instructions that have two source operands. If used with non-commutative instructions like subtract **Rb** is on the left side of the operator (**Rb** – **`src0`**). After the operation the 16-bit destination operand **`dst`** is stored in register **Rb**.

C language description

```
uint16 src1,dst;
src1 = Rb;
Rb = dst;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rb									2	1	0					

SRs

SRn register used as source

The content of register **SRs** is the 16-bit source operand **`src`**. Reserved register bits and bit fields read as zeros. Used with instruction: **`mfsr`**

C language description

```
uint16 src;
src = SRs;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRs			2	1	0											

SRd

SRn register used as destination

The 16-bit destination operand **`dst`** is stored in register **SRd**. The read-only special register **ID** cannot be used as destination register. Used with instruction: **`mtsr`**

C language description

```
uint16 dst;
SRd = dst;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRd			2	1	0											

SRLd

low order special register SRLn used as destination

The 16-bit destination operand **`dst`** is stored in register **SRLd**. The group of low order special registers **SRLn** includes registers **CC**, **CS**, **LC** and **AU**. Used with instruction: **`mtsr`**

C language description

```
uint16 dst;
SRLd = dst;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRLd										1	0					

Ad

address register An used as destination

The 16-bit destination operand **`dst`** is stored in register **Ad**. Used with instruction: **`adsp`**

C language description

```
uint16 dst;
Ad = dst;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ad										1	0					

RGS

`Register Selection`

RGS is a selection of registers of the **Rn** and **SRn** groups. Up to seven registers can be selected by seven flags in the opcode word. Registers **R0**, **R1**, **R2**, **R3**, **R4**, **R5**, **R6**, **TA** and **SA** can be contained in a register selection **RGS**. **R0** and **R1** can only be in an **RGS** of load/store byte instructions. **TA** and **SA** can only be in an **RGS** of load/store short instructions. The register selection **RGS** is either the source operand **`src[n-1:0]`** of a memory store instruction with addressing mode **`-(An)`**or the destination operand **`dst[n-1:0]`**of a memory load instruction with addressing mode **`(An)+`**. The **RGS** source or destination operand is an array of **`n`** 8-bit or 16-bit values where **`n`** is the number of registers selected by **RGS**. In memory the **`n`** values are located at adjacent byte or 16-bit word address locations. 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 **`ldbt, ldsh, stbt, stsh`**

C language description

```
uint16 src16[n], dst16[n];
uint8 src8[n], dst8[n];
if(instruction == stsh)
  src16[n-1:0] = RGS;
if(instruction == ldsh)
  RGS = dst16[n-1:0];
if(instruction == stbt)
  src8[n-1:0] = RGS;
if(instruction == ldbt)
  RGS = dst8[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 0-6 identify **R0** – **R6**
- character T identifies **TA**
- character S identifies **SA**

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`ldbt (An)+,RGS`		6	5	4	3				2	1	0					
`stbt RGS,-(An)`		0	1	2	3				4	5	6					
`ldsh (An)+,RGS`		6	5	4	3				2	T	S					
`stsh RGS,-(An)`		S	T	2	3				4	5	6					

6.4 Memory operand addressing

Memory operands are 8-bit, 16-bit or n* 16-bit memory words used as source operand of load instructions or as destination operand of store instructions. Addressing modes for memory operands determine the 16-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 2-bit field that selects one of registers **SP**, **TA**, **R6** or **R7** as indirect address.

DA8 — 8-bit direct data address

For byte (8-bit) memory operands the effective address **eda** is the 8-bit constant **DA8** extracted from the opcode and zero-extended to 16 bits. Legal values for **eda** are from 0x0000 – 0x00FF. For short (16-bit) memory operands the effective address **eda** is the 8-bit constant **DA8** extracted from the opcode, left-shifted by one bit and zero-extended to 16 bits. Legal values for **eda** are from 0x0000 – 0x01FE (even values only).

Used with instructions **ldbt, ldsh, stbt, stsh**

C language description

```
uint16 src,dst;
void *eda;
if(instruction == (ldbt|stbt))
  eda = DA8;
if(instruction == (ldsh|stsh))
  eda = DA8 << 1;
if(instruction == (ldbt|ldsh))
  dst = *eda;
if(instruction == (stbt|stsh))
  *eda = src;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DA8	4	3	2	1	0	7	6	5								

$(DO5_S, An)$ — Address register indirect with 5-bit signed offset

The 5-bit constant **$DO5_S$** (Signed) is extracted from the opcode and sign-extended to the 16 bit offset **ofs**. The **ofs** value range is [-16,15]. For byte (8-bit) memory operands the effective address **eda** is **ofs** added to the value of the address register **An**. For short (16-bit) memory operands the effective address **eda** is **ofs** left-shifted by one bit and added to the value of the address register **An**.

Used with instructions **ldbt, ldsh, stbt, stsh**

C language description

```
uint16 src,dst,ofs;
void *eda;
ofs = DO5S & 0x10 ? 0xFFE0 | DO5S : DO5S;
if(instruction == (ldbt|stbt))
  eda = An + ofs;
if(instruction == (ldsh|stsh))
  eda = An + (ofs << 1);
if(instruction == (ldbt|ldsh))
  dst = *eda;
if(instruction == (stbt|stsh))
  *eda = src;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$DO5_S$	4	3	2	1	0											
An							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 **ldbt, ldsh, stbt, stsh**

C language description

```
uint16 src,dst;
void *eda;
eda = An + sizeof(memory operand) * Rx;
if(instruction == (ldbt|ldsh))
  dst = *eda;
if(instruction == (stbt|stsh))
  *eda = src;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rx			2	1	0											
An							1	0								

(An)+ Address register indirect with post-increment

This addressing mode is available for both **Rd** destination operands and for **RGS** (register selection) destination operands (**ldbt** and **ldsh** instructions only). The effective address **eda** of the data memory operand is the value of the address register **An**. After the memory access(es) the address register **An** is incremented by the **size** (in bytes) of the operand. Registers of the **RGS** selection are read from memory in the following fixed order (reversed order of **–(An)** addressing mode): **R0**/**SA**, **R1**/**TA**, **R2**, **R3**, **R4**, **R5**, **R6**

Used with instructions **ldbt, stbt, stsh, ldsh**

C language description

```
uint16 dst,rgs[n];       // n = number of registers in RGS
void *eda;
int i;
eda = An;
if((dst==rgs) & (instruction==(ldbt|ldsh)))
  for(i=0;i < n;i++){
    rgs[i] = *eda;
    eda += sizeof(rgs[i]);
  }
else{
  dst = *eda;
  eda += sizeof(dst);
}
An = eda;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
An							1	0								

-(An) Address register indirect with pre-decrement

This addressing mode is available for both **Rs** source operands and for **RGS** (register selection) source operands (**stbt** and **stsh** instructions only). 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 written to memory in the following fixed order (reversed order of **(An)+** addressing mode): **R6**, **R5**, **R4**, **R3**, **R2**, **R1**/**TA**, **R0**/**SA**

Used with instructions **ldbt, ldsh, stbt, stsh**

C language description

```
uint16 src, rgs[n];   // n = number of registers in RGS
void *eda;
int i;
eda = An;
if((src==rgs) & (instruction==(stbt,stsh)))
  for(i=0;i < n;i++){
    eda -= sizeof(rgs[i]);
    *eda = rgs[i];
  }
else{
  eda -= sizeof(dst);
  *eda = src;
}
An = eda;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
An							1	0								

(An)* Address register indirect with post-update

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

Used with instructions **ldbt, ldsh, stbt, stsh**

C language description

```
uint16 dst;
void *eda;
eda = An;
dst = *eda;
An += AU;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
An							1	0								

6.5 Instruction addressing

Instruction addresses point to 16-bit opcode words in the instruction memory. With the exception of some flow instructions the effective instruction address **eia** of the next instruction is the address of the current instruction plus one.

C language description

```
uint16 *eia;
eia[next instruction] = eia[current instruction] + 1;
```

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**.

IA12 — 12-bit absolute instruction address

The 12-bit field **IA12** is extracted from the opcode word. The 16-bit effective instruction address **eia** is the concatenation of the 4-bit hidden register ***IAH*** and the 12-bit constant **IA12**. ***IAH*** becomes **eia**[15:12] and **IA12** becomes **eia**[11:0]. The **IA12** value range is [0,0x0FFF]. After the instruction fetch ***IAH*** is set to zero.

Used with instructions **`jump, jpsr`**

C language description

```
uint16 *eia;
eia = (IAH << 12) | IA12;
IAH = 0;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IA12	11	10	9	8	7	6	5	4	3	2	1	0				

Note

The concept of the **`IA12`** addressing mode together with the 4-bit hidden register ***IAH*** and the **`siah`** instruction enables the full 16-bit instruction address space to be reached with direct addresses. The first 4k instructions from **`0x0000-0x0FFF`** can be reached with a single **`jump/jpsr`** instruction and the **`IA12`** addressing mode. Instruction addresses **`> 0x0FFF`** can be reached with a two-instructions sequence. The first instruction is a **`siah`** instruction and sets the 4-bit hidden register ***IAH*** to bits[15:12] of the target **eia**. The second instruction is a **`jump/jpsr`** with the **`IA12`** addressing mode. For this concept to work it is necessary that the **`IA12`** addressing mode clears the ***IAH*** register to zero after the instruction fetch.

$IO8_S$ — 8-bit instruction offset (Signed)

The 8-bit field **$IO8_S$** is extracted from the opcode word. 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

```
uint16 *eia,*cia;
eia = cia + (IO8S & 0x80 ? 0xFF00 | IO8S : IO8S);
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$IO8_S$	7	6	5	4	3	2	1	0								

IAH4 — 4-bit instruction address high

The 4-bit field **IAH4** is extracted from the opcode word and stored in the 4-bit hidden register ***IAH***.

Used with instruction **`siah`**

C language description

```
uint16 *eia,*cia;
IAH = IAH4;
```

opcode bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IAH4								3	2	1	0					

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**.

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.

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:

- **`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 necessarily 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

These tables list all addressing modes of an 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 $\mathbf{i_{th}}$ 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

ldbt

load byte

Loads the byte (8-bit word) from the effective data address **eda** in the data memory, zero-extends the value to 16 bits and stores it in the 16-bit destination **`dst`**. Some addressing modes update the indirect address register **An** as indicated in the addressing modes table. With the **`(An)+,RGS`** addressing mode the update parameter **n** is the number of registers contained in **RGS** and can take values from 1 to 7.

C language description

```
uint16 dst;
uint8 *eda;
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
direct 8-bit data address	ldbt DA8,Rd	DA8	not appl.	Rd
indirect data address with 5-bit offset	ldbt ($DO5_S$,An),Rd	An+$DO5_S$	no update	Rd
indirect data address with index	ldbt (Rx,An),Rd	An+Rx	no update	Rd
indirect data address with post-increment	ldbt (An)+,Rd	An	+= 1	Rd
indirect data address with pre-decrement	ldbt -(An),Rd	An-1	-= 1	Rd
indirect data address with post-update	ldbt (An)*,Rd	An	+= AU	Rd
indirect data address with post-increment	ldbt (An)+,RGS	An+i	+= n	RGS

ldsh

load short

Loads the short operand (16-bit word) from the effective data address **eda** in the data memory and stores it in the 16-bit destination **`dst`**. Some addressing modes update the indirect address register **An** as indicated in the addressing modes table. With the **`(An)+,RGS`** addressing mode the update parameter **n** is the number of registers contained in **RGS** and can take values from 1 to 7.

C language description

```
uint16 dst,*eda;
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
direct 8-bit data address	ldsh DA8,Rd	DA8	not appl.	Rd
indirect data address with 5-bit offset	ldsh ($DO5_S$,An),Rd	An+$DO5_S$	no update	Rd
indirect data address with index	ldsh (Rx,An),Rd	An+2*Rx	no update	Rd
indirect data address with post-increment	ldsh (An)+,Rd	An	+= 2	Rd
indirect data address with pre-decrement	ldsh -(An),Rd	An-2	-= 2	Rd
indirect data address with post-update	ldsh (An)*,Rd	An	+= AU	Rd
indirect data address with post-increment	ldsh (An)+,RGS	An+2*i	+= 2*n	RGS

mfdp

move from debug port

The 16-bit word driven on the debug input port **`dbgi`** of the processor is stored in the 16-bit destination **`dst`**.

C language description

```
uint16 dbgi,dst;
dst = dbgi;
```

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

mfsr

move from special register

The 16-bit source **src** is stored in the 16-bit destination **dst**. Reserved bits of special register sources read as zeros.

C language description

```
uint16 src,dst;
dst = src;
```

Addressing Modes	assembly format	src	dst
dual registers	mfsr SRs,Rd	SRs	Rd

move

move

The source operand **src** is stored in the 16-bit destination operand **dst**.

C language description

```
uint16 src,dst;
dst = src;
```

Addressing Modes	assembly format	src	dst
dual registers	move Rs,Rd	Rs	Rd
constant and single register	move $C9_S$,Rd	$C9_S$	Rd

mtdp

move to debug port

The 16-bit source operand **src** is transferred to the debug output port **dbgo**.

C language description

```
uint16 dbgo,src;
dbgo = src;
```

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

mtsr

move to special register

The 16-bit source **src** is stored in the 16-bit destination **dst**. With the 7-bit constant source operand option only the four low order special registers **SRLd** can be used as destination and the format of the 7-bit constant depends on the destination register. The addressing modes table therefor has separate entries for the 7-bit constant addressing modes and each possible destination register.

C language description

```
uint16 src,dst;
dst = src;
```

Addressing Modes	assembly format	src	dst
dual registers	mtsr Rs,SRd	Rs	SRd
constant and single register	mtsr $C7_U$,CC	$C7_U$	CC
constant and single register	mtsr $C7_U$,CS	$C7_U$	CS
constant and single register	mtsr $C7_{UN}$,LC	$C7_{UN}$	LC
constant and single register	mtsr $C7_{SN}$,AU	$C7_{SN}$	AU

stbt

store byte

Extracts the least significant byte (8-bit word) from the 16-bit source operand **src** and stores it at the effective data address **eda** in data memory. Some addressing modes update the indirect address register **An** as indicated in the addressing modes table. With the **RGS,-(An)** addressing mode the update parameter **n** is the number of registers contained in **RGS** and can take values from 1 to 7.

C language description

```
uint16 src;
uint8 *eda;
*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
direct 8-bit data address	stbt Rs,DA8	DA8	not appl.	Rs
indirect data address with 5-bit offset	stbt Rs,(DO5$_S$,An)	An+DO5$_S$	no update	Rs
indirect data address with index	stbt Rs,(Rx,An)	An+Rx	no update	Rs
indirect data address with post-increment	stbt Rs,(An)+	An	+= 1	Rs
indirect data address with pre-decrement	stbt Rs,-(An)	An-1	-= 1	Rs
indirect data address with post-update	stbt Rs,(An)*	An	+= AU	Rs
indirect data address with pre-decrement	stbt RGS,-(An)	An-i-1	-= n	RGS

stsh

store short

Stores the 16-bit source operand(s) **src** at the effective data address **eda** in the data memory. Some addressing modes update the indirect address register **An** as indicated in the addressing modes table. With the **RGS,-(An)** addressing mode the update parameter **n** is the number of registers contained in **RGS** and can take values from 1 to 7.

C language description

```
uint16 src,*eda;
*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
direct 8-bit data address	stsh Rs,DA8	DA8	not appl.	Rs
indirect data address with 5-bit offset	stsh Rs,(DO5$_S$,An)	An+DO5$_S$	no update	Rs
indirect data address with index	stsh Rs,(Rx,An)	An+2*Rx	no update	Rs
indirect data address with post-increment	stsh Rs,(An)+	An	+= 2	Rs
indirect data address with pre-decrement	stsh Rs,-(An)	An-2	-= 2	Rs
indirect data address with post-update	stsh Rs,(An)*	An	+= AU	Rs
indirect data address with pre-decrement	stsh RGS,-(An)	An-2*i-2	-= 2*n	RGS

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 sf16 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.

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:

`uint16` type: 16-bit unsigned integer

`sint16` type: 16-bit signed integer

`uint4` type: 4-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 3 of source operand 0 is referenced by **`src0[3].`**

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

These tables list 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

The absolute value of the 16-bit source operand **src** is stored in the 16-bit destination operand **dst**.

C language description

```
uint16 src,dst;
dst = src & 0x8000 ? –src : src;
```

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

adcf

add carry flag

Adds the carry flag **CC.C** to the 16-bit source operand **src** and stores the result in the 16-bit destination operand **dst**. The flags in **CC** are updated. 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

```
uint16 src,dst;
dst = src + CC.C;
CC.C = (src1[15]&src0[15]) | (src1[15]&~dst[15]) | (src0[15]&~dst[15]);
CC.O = (src1[15]&src0[15]&~dst[15]) | (~src1[15]&~src0[15]&dst[15]);
CC.Z = CC.Z & (dst == 0) ? 1 : 0;
CC.N = dst[15];
```

Addressing Modes	assembly format	src	dst
dual registers	adcf Rs,Rd	Rs	Rd

addc

add with carry

Adds the 16-bit source operands **src0**, **src1** and the carry flag **CC.C**. The result is stored in the 16-bit destination operand **dst** and the flags in **CC** are updated. 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

```
uint16 src0,src1,dst;
dst = src1 + src0 + CC.C;
CC.C = (src1[15]&src0[15]) | (src1[15]&~dst[15]) | (src0[15]&~dst[15]);
CC.O = (src1[15]&src0[15]&~dst[15]) | (~src1[15]&~src0[15]&dst[15]);
CC.Z = CC.Z & (dst == 0) ? 1 : 0;
CC.N = dst[15];
```

Addressing Modes	assembly format	src0	src1	dst
triadic registers	addc Rs0,Rs1,Rd	Rs0	Rs1	Rd

addt

add to

Adds the two 16-bit source operands **src0** and **src1**, stores the result in the 16-bit destination operand **dst** and updates the flags in **CC**.

C language description

```
uint16 src0,src1,dst;
dst = src1 + src0;
CC.C = (src1[15]&src0[15]) | (src1[15]&~dst[15]) | (src0[15]&~dst[15]);
CC.O = (src1[15]&src0[15]&~dst[15]) | (~src1[15]&~src0[15]&dst[15]);
CC.Z = dst == 0 ? 1 : 0;
CC.N = dst[15];
```

Addressing Modes	assembly format	src0	src1	dst
triadic registers	addt Rs0,Rs1,Rd	Rs0	Rs1	Rd
constant and single register	addt $C8_{UN}$,Rb	$C8_{UN}$	Rb	Rb

addh

add high

The 16-bit constant **C16** is added to the 16-bit source operand **src1**. The result is stored in the 16-bit destination operand **dst**. Bits [7:0] of constant **C16** are always zero.

C language description

```
uint16 src1,dst;
dst = C16 + src1;
```

Addressing Mode	assembly format	src0	src1	dst
constant and single register	addh C16,Rb	C16	Rb	Rb

Notes

Main purpose of the **addh** instruction is the generation of 16-bit constants in general purpose registers **Rn**. This is done by a **move** $C9_S$**,Rd** instruction followed by a **addh** instruction with the **dst** of the **move** used as both **src1** and **dst** operands. Bits[7:0] of the $C9_S$ of the **move** instruction are the lower 8 bits and the **C16** of the **addh** instruction are the higher 8 bits of the 16-bit constant.

adsp

add to stack pointer

The 7-bit constant $C7_{SN}$ is sign-extended to 16 bits and added to the value of special register **SP**. The result is stored in the 16-bit destination **dst** which is one of the four address registers **An**. $C7_{SN}$ is not including zero. If the field value is zero 64 is added to the value of **SP**.

C language description

```
uint16 con,dst;
con = C7SN == 0 ? 64 : (C7SN & 0x40 ? 0xFF80 | C7SN : C7SN);
dst = SP + con;
```

Addressing Mode	assembly format	src0	src1	dst
constant and single register	adsp C7SN,Ad	$C7_{SN}$	SP	Ad

Notes

The **adsp** instruction is used to allocate and de-allocate stack space in function prologues and epilogues (**SP** destination). It is also used to load an address register with a pointer to a stack location.

clzr

count leading zeros

Counts the number of zero bits in the 16-bit source operand **src** starting with the MSB until the first '1' bit is found. The count is stored in the 16-bit destination operand **dst**. If no '1' bit is found (**src == 0**) the count stored in the destination operand **dst** is 16.

C language description

```
uint16 src,dst;
uint4 bti;
dst = 16;
for(bti=15;bti >= 0;bti--)
  if(src[bti] == 1){
    dst = 15 – bti;
    break;
  }
```

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

cmpc

compare with carry

Subtracts the 16-bit source operand **src0** and the carry flag **CC.C** from the 16-bit source operand **src1** and updates the flags in **CC** according to the result. 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

```
uint16 src0,src1,tmp;
tmp = src1 - src0 – CC.C;
CC.C = (~src1[15]&src0[15]) | (~src1[15]&tmp[15]) | (src0[15]&tmp[15]);
CC.O = (src1[15]&~src0[15]&~tmp[15]) | (~src1[15]&src0[15]&tmp[15]);
CC.Z = CC.Z & (tmp == 0) ? 1 : 0;
CC.N = tmp[15];
```

Addressing Modes	assembly format	src0	src1
dual registers	cmpc Rs0,Rs1	Rs0	Rs1

comp

compare

Subtracts the 16-bit source operand **src0** from the 16-bit source operand **src1** and updates the flags in **CC** according to the result.

C language description

```
uint16 src0,src1,tmp;
tmp = src1 - src0;
CC.C = (~src1[15]&src0[15]) | (~src1[15]&tmp[15]) | (src0[15]&tmp[15]);
CC.O = (src1[15]&~src0[15]&~tmp[15]) | (~src1[15]&src0[15]&tmp[15]);
CC.Z = tmp == 0 ? 1 : 0;
CC.N = tmp[15];
```

Addressing Modes	assembly format	src0	src1
dual registers	comp Rs0,Rs1	Rs0	Rs1
constant and single register	comp $C8_A$,Rs1	$C8_A$	Rs1

cpcf

compare carry flag

Subtracts the carry flag **CC.C** from the 16-bit source operand **src** and updates the flags in **CC** according to the result. 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

```
uint16 src,tmp;
tmp = src – CC.C;
CC.C = ~src[15] & tmp[15];
CC.O = src1[15] & ~tmp[15];
CC.Z = CC.Z & (tmp == 0) ? 1 : 0;
CC.N = tmp[15];
```

Addressing Modes	assembly format	src
dual registers	cpcf Rs	Rs

negt

negate

The 2’s complement of the 16-bit source operand **src** is stored in the 16-bit destination operand **dst**.

C language description

```
uint16 src,dst;
dst = –src;
```

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

sbcf

subtract carry flag

Subtracts the carry flag **CC.C** from the 16-bit source operand **src** stores the result in the 16-bit destination operand **dst** and updates the flags in **CC**. 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

```
uint16 src,dst;
dst = src - CC.C;
CC.C = ~src[15] & dst[15];
CC.O = src[15] & ~dst[15];
CC.Z = CC.Z & (dst == 0) ? 1 : 0;
CC.N = dst[15];
```

Addressing Modes	assembly format	src	dst
dual registers	sbcf Rs	Rs	Rd

subc

subtract with carry

Subtracts the 16-bit source operand **src0** and the carry flag **CC.C** from the 16-bit source operand **src1**. The result is stored in the 16-bit destination operand **dst** and the flags in **CC** are updated. 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

```
uint16 src0,src1,dst;
dst = src1 - src0 - CC.C;
CC.C = (~src1[15]&src0[15]) | (~src1[15]&dst[15]) | (src0[15]&dst[15]);
CC.O = (src1[15]&~src0[15]&~dst[15]) | (~src1[15]&src0[15]&dst[15]);
CC.Z = CC.Z & (dst == 0) ? 1 : 0;
CC.N = dst[15];
```

Addressing Modes	assembly format	src0	src1	dst
triadic registers	subc Rs0,Rs1,Rd	Rs0	Rs1	Rd

subf

subtract from

Subtracts the 16-bit source operand **src0** from the 16-bit source operand **src1**, stores the result in the 16-bit destination operand **dst** and updates the flags in **CC**.

C language description

```
uint32 src0,src1,dst;
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
triadic registers	subf Rs0,Rs1,Rd	Rs0	Rs1	Rd
constant and single register	subf $C8_{UN}$,Rb	$C8_{UN}$	Rb	Rb

sxbt

sign extend byte

Extends the sign of the low-byte of the 16-bit source operand **src** to the high byte and store the result in the 16-bit destination operand **dst**.

C language description

```
uint16 src,dst;
dst = src & 0x80 ? 0xFF00 | src : src & 0xFF;
```

Addressing Modes	assembly format	src	dst
dual registers	sxbt Rs,Rd	Rs	Rd

Notes

Main purpose of the **sxbt** instruction is to convert signed byte operands loaded from memory into a general purpose register **Rn** to a 16-bit 2's complement format for subsequent computations.

sxsh

sign extend short

Extends the sign of the 16-bit source operand **src** to the 16-bit destination operand **dst**.

C language description

```
uint16 src,dst;
dst = src & 0x8000 ? 0xFFFF : 0;
```

Addressing Modes	assembly format	src	dst
dual registers	sxsh Rs,Rd	Rs	Rd

Notes

Main purpose of the **sxsh** instruction is to convert signed short operands loaded from memory into a general purpose register **Rn** to a multi-precision (> 16-bits, e.g. 32-bit) 2's complement format stored in multiple general purpose registers for subsequent multi-precision computations.

8.4 Logic Instructions

andb

logic AND bit wise

Performs a bit wise logic AND operation between the two 16-bit source operands **src0** and **src1**, stores the result in the 16-bit destination operand **dst** and updates the flags in **CC**. 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

```
uint16 src0,src1,dst;
boolean par;
uint4 bti;
dst = src1 & src0;
par = 0;
for(bti=0;bti < 16;bti++)
  par ^= dst[bti];
CC.O = par ^ CC.C;
CC.C = par;
CC.Z = dst == 0 ? 1 : 0;
CC.N = dst[15];
```

Addressing Modes	assembly format	src0	src1	dst
triadic registers	andb Rs0,Rs1,Rd	Rs0	Rs1	Rd
constant and single register	andb $C8_U$,Rb	$C8_U$	Rb	Rb

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 sf16 **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 > 16 bits. For the parity of long bit strings first **CC.C** and **CC.O** are cleared by e.g. a **mtsr 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

Inverts the 16-bit source operand **src** and stores the result in the 16-bit destination operand dst.

C language description

```
uint16 src,dst;
dst = ~src;
```

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

iorb

inclusive OR bit wise

Performs a bit wise inclusive or between the two 16-bit source operands **src0** and **src1** and stores the result in the 16-bit destination operand **dst**.

C language description

```
uint16 src0,src1,dst;
dst = src1 | src0;
```

Addressing Modes	assembly format	src0	src1	dst
triadic registers	iorb Rs0,Rs1,Rd	Rs0	Rs1	Rd
constant and single register	iorb $C8_U$,Rb	$C8_U$	Rb	Rb

xorb

exclusive OR

Performs a bit wise exclusive or between the two 16-bit source operands **src0** and **src1** and stores the result in the 16-bit destination operand **dst**.

C language description

```
uint16 src0,src1,dst;
dst = src1 ^ src0;
```

Addressing Modes	assembly format	src0	src1	dst
triadic registers	xorb Rs0,Rs1,Rd	Rs0	Rs1	Rd

8.5 Shift Instructions

shlf

shift left with feedback

Performs a left shift with feedback (rotate) operation of the 16-bit source operand **src** and stores the result in the 16-bit destination **dst**. The shift count **shc4** can take values from 0 to 15. 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 **shc4** is equal to bits [3:0] of source register **Rs0**. Bits [15:4] of **Rs0** are ignored.

C language description

```
uint16 src,dst;
uint4 shc4;
dst = (src << shc4) | (src >> (16 – shc4));
```

Addressing Modes	assembly format	shc4	src	dst
triadic registers	shlf Rs0,Rs1,Rd	Rs0	Rs1	Rd
constant and single register	shlf SHC4,Rb	SHC4	Rb	Rb

shlz

shift left with zero fill

Performs a left shift with zero fill of the 16-bit source operand **src** and stores the result in the 16-bit destination **dst**. The shift count **shc4** can take values from 0 to 15. In addressing modes with indirect shift count **shc4** is equal to bits [3:0] of source register **Rs0**. Bits [15:4] of **Rs0** are ignored.

C language description

```
uint16 src,dst;
uint4 shc4;
dst = src << shc4;
```

Addressing Modes	assembly format	shc4	src	dst
triadic registers	shlz Rs0,Rs1,Rd	Rs0	Rs1	Rd
constant and single register	shlz SHC4,Rb	SHC4	Rb	Rb

shrs

shift right signed

Performs a signed right shift of the 16-bit source operand **src** and stores the result in the 16-bit destination **dst**. The shift count **shc4** can take values from 0 to 15. Signed shift means that the sign of the source operand **src[15]** is preserved and the destination operand **dst** has the same sign as the source operand **src**. In addressing modes with indirect shift count **shc4** is equal to bits [3:0] of source register **Rs0**. Bits [15:4] of **Rs0** are ignored.

C language description

```
uint16 src,dst;
uint4 shc4;
dst = src >> shc4;
if(src[15])
  dst |= 0xFFFF << (16 – shc4);
```

Addressing Modes	assembly format	shc4	src	dst
triadic registers	shrs Rs0,Rs1,Rd	Rs0	Rs1	Rd
constant and single register	shrs SHC4,Rb	SHC4	Rb	Rb

shru

shift right unsigned

Performs a right shift of the 16-bit source operand **src** and stores the result in the 16-bit destination **dst**. The shift count **shc4** can take values from 0 to 15. In addressing modes with indirect shift count **shc4** is equal to bits [3:0] of source register **Rs0**. Bits [15:4] of **Rs0** are ignored.

C language description

```
uint16 src,dst;
uint4 shc4;
dst = src >> shc4;
```

Addressing Modes	assembly format	shc4	src	dst
triadic registers	shru Rs0,Rs1,Rd	Rs0	Rs1	Rd
constant and single register	shru SHC4,Rb	SHC4	Rb	Rb

8.6 Bit manipulation instructions

btcl

bit clear

Clears the bit of the 16-bit source operand **src** indexed by **bti4** and stores the result in the 16-bit destination **dst**. The bit index **bti4** can take values from 0 to 15. In addressing modes with indirect bit index **bti4** is equal to bits [3:0] of the source register **Rs0**. Bits [15:4] of **Rs0** are ignored.

C language description

```
uint16 src,dst;
uint4 bti4;
dst = src & ~(1 << bti4);
```

Addressing Modes	assembly format	bti4	src	dst
triadic registers	btcl Rs0,Rs1,Rd	Rs0	Rs1	Rd
constant and single register	btcl BTI4,Rb	BTI4	Rb	Rb

btst

bit set

Sets the bit of the 16-bit source operand **src** indexed by **bti4** and stores the result in the 16-bit destination **dst**. The bit index **bti4** can take values from 0 to 15. In addressing modes with indirect bit index **bti4** is equal to bits [3:0] of the source register **Rs0**. Bits [15:4] of **Rs0** are ignored.

C language description

```
uint16 src,dst;
uint4 bti4;
dst = src | (1 << bti4);
```

Addressing Modes	assembly format	bti4	src	dst
triadic registers	btst Rs0,Rs1,Rd	Rs0	Rs1	Rd
constant and single register	btst BTI4,Rb	BTI4	Rb	Rb

bttg

bit toggle

Toggles the bit of the 16-bit source operand **src** indexed by **bti4** and stores the result in the 16-bit destination **dst**. The bit index **bti4** can take values from 0 to 15. In addressing modes with indirect bit index **bti4** is equal to bits [3:0] of the source register **Rs0**. Bits [15:4] of **Rs0** are ignored.

C language description

```
uint16 src,dst;
uint4 bti4;
dst = src ^ (1 << bti4);
```

Addressing Modes	assembly format	bti4	src	dst
triadic registers	bttg Rs0,Rs1,Rd	Rs0	Rs1	Rd
constant and single register	bttg BTI4,Rb	BTI4	Rb	Rb

btts

bit test

Tests the bit of the 16-bit source operand **src** indexed by **bti4** and updates the condition codes in **CC** according to the result. The bit index **bti4** can take values from 0 to 15. In addressing modes with indirect bit index **bti4** is equal to bits [3:0] of source register **Rs0**. Bits [15:4] of **Rs0** are ignored.

C language description

```
uint16 src,tmp;
uint4 bti4;
tmp = src & (1 << bti4);
CC.C = 0;
CC.O = 0;
CC.Z = tmp == 0 ? 1 : 0;
CC.N = tmp[15];
```

Addressing Modes	assembly format	bti4	src
triadic registers	btts Rs0,Rs1	Rs0	Rs1
constant and single register	btts BTI4,Rs	BTI4	Rs

8.7 Multiply Instructions

mlhs

multiply high signed

Performs a signed multiply of the two 16-bit source operands **src0** and **src1**. The 31-bit product is right shifted (sign preserved) by 16 bits, sign-extended to 16 bits and stored in the 16-bit destination **dst**.

C language description

```
uint16 dst;
sint16 scr0,scr1;
dst = (src1 * src0) >> 16;
```

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

mlhu

multiply high unsigned

Performs an unsigned multiply of the two 16-bit source operands **src0** and **src1**. The 32-bit product is right shifted by 16 bits and stored in the 16-bit destination **dst**.

C language description

```
uint16 src0,src1,dst;
dst = (src1 * src0) >> 16;
```

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

mult

multiply

Performs a multiply of the two 16-bit source operands **src0** and **src1** and stores the lower 16 bits of the 32-bit product in the 16-bit destination operand **dst**.

C language description

```
uint16 src0,src1,dst;
dst = src1 * src0;
```

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

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.

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:

`uint16` type: 16-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 11 of source operand 0 is referenced by **`src0[11].`**

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 special 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 **IO8_S**. The 8-bit instruction address offset **IO8_S** is sign-extended to 16 bits and added to **cia**. If **LC** is zero after the decrement program execution continues with the next instruction in sequence.

C language description

```
uint6 tmp,*cia,*eia;
LC -= 1;
if(LC != 0){
  tmp = IO8_S & 0x80 ? 0xFF00 | IO8_S : IO8_S;
  eia = cia + tmp;
}
else
  eia = cia + 1;
```

Addressing Modes	assembly format
8-bit instruction address offset	`brlc IO8_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 **IO8_S**. The 8-bit instruction address offset **IO8_S** is sign-extended to 16 bits and added to **cia**. If the condition **cnd** is false instruction execution continues with the next instruction in sequence.

C language description

```
uint16 tmp,*cia,*eia;
boolean cnd;
if(cnd == true){
  tmp = IO8_S & 0x80 ? 0xFF00 | IO8_S : IO8_S;
  eia = cia + tmp;
}
else
  eia = cia + 1;
```

Addressing modes

All of the 14 conditional branch instructions have the same addressing mode: "8-bit instruction address offset". 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 IO8_S`
branch if carry	`CND = C;`	`brcr IO8_S`
branch if no overflow	`CND = ~O;`	`brno IO8_S`
branch if overflow	`CND = O;`	`brof IO8_S`
branch if non zero	`CND = ~Z;`	`brnz IO8_S`
branch if zero	`CND = Z;`	`brzr IO8_S`
branch if positive	`CND = ~N;`	`brps IO8_S`
branch if negative	`CND = N;`	`brng IO8_S`
Branch if lower or same	`CND = C | Z;`	`brls IO8_S`

branch if higher	`CND = ~C & ~Z;`	`brhi` $IO8_S$
branch if lower	`CND = (N & ~O) | (~N & O);`	`brlo` $IO8_S$
branch if greater of equal	`CND = (N & O) | (~N & ~O);`	`brge` $IO8_S$
branch if lower or equal	`CND = Z | (N & ~O) | (~N & O);`	`brle` $IO8_S$
branch if greater	`CND = ~Z & ((N & O) | (~N & ~O));`	`brgt` $IO8_S$

clas

clear address select

Selects physical register **SA** for access by **mtsr**/**mfsr** instructions with the **SA** special register number by clearing the **AS** bit in register **CS**.

C language description

```
CS.AS = 0;
```

Addressing Modes	assembly format
implied	`clas`

Notes

Load/store instructions with the **SA** special register as destination or source register always access the **SA** physical register and never the ***IRA*** hidden register.

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 hidden register ***IAH*** concatenated with a constant in the opcode or from special register **TA**. In case of the **IA12** addressing mode hidden register ***IAH*** is cleared after the instruction fetch from **eia**.

C language description

```
uint16 *eia;
if(addressing-mode == IA12)
  IAH = 0;
```

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;`
12-bit absolute instruction address	`jump IA12`	`eia = (IAH << 12) | IA12;`

jpsr

jump to subroutine

The address of the next instruction in sequence following the **jpsr** instruction is saved in special register **SA**. This is the current instruction address **cia** plus 1. Program execution continues at the effective instruction address **eia** generated from hidden register ***IAH*** concatenated with a constant in the opcode or from special register **TA**. In case of the **IA12** addressing mode hidden register ***IAH*** is cleared after the instruction fetch from **eia**.

C language description

```
uint16 *cia,*eia;
SA = cia + 1;
if(addressing-mode == IA12)
  IAH = 0;
```

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;`
12-bit absolute instruction address	`jpsr IA12`	`eia = (IAH << 12) | IA12;`

Notes

sf16 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 to the 16-bit value driven on the debug input port **`dbgi`**.

C language description

```
uint16 dbgi,*cia;
cia = dbgi;
```

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** and the hidden instruction address high ***IAH*** register are restored from hidden registers ***CCS*** and ***IAHS*** respectively 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 hidden register ***IRA*** as effective instruction address **`eia`**. If the processor is currently not executing an interrupt the behavior of an **`rtir`** instruction is not defined.

C language description

```
uint16 *cia,*eia;
if(CS.IR){
  eia = IRA;
  CC  = CCS;
  IAH = IAHS;
  CS.IR = 0;
}
```

Addressing Modes	assembly format
implied	`rtir`

rtsr

return from subroutine

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

C language description

```
uint16 *eia;
eia = SA;
```

Addressing Modes	assembly format
implied	`rtsr`

Notes

sf16 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. This is required e.g. for atomic read/modify/write operations on semaphore variables.

siah

set instruction address high

The 4-bit hidden register ***IAH*** is set with the constant **IAH4** contained in the opcode.

C language description

```
IAH = IAH4;
```

Addressing Modes	assembly format
4-bit instruction address high	`siah IAH4`

Notes

The **`siah`** instruction is used to extend the direct addressable instruction address space from 12 bits (4k instructions) to 16 bits (64k instructions). With the **`IA12`** addressing mode alone only the lower 4k instructions from **`0x0000-0x0FFF`** can be reached. For **`jump`**/**`jpsr`** instructions with **`eia`** > **`0x0FFF`** first hidden register ***IAH*** is set to **`eia`**[15:12] by a **`siah`** instruction. The following **`jump`**/**`jpsr`** instruction then contains **`eia`**[11:0] and clears ***IAH*** to zero after the jump.

Although the typical use is a two-instruction sequence of **`siah`** and a **`jump`**/**`jpsr`** with **`IA12`** addressing mode it is not mandatory that a **`siah`** instruction is immediately followed by a **`jump`**/**`jpsr`** **`IA12`** instruction. ***IAH*** will keep the value set by a **`siah`** up to the next **`jump`**/**`jpsr`** **`IA12`** instruction no matter how many instructions are in between. In case an interrupt occurs between a **`siah`** instruction and the next **`jump`**/**`jpsr`** **`IA12`** instruction ***IAH*** is saved in ***IAHS*** and then cleared and is restored from ***IAHS*** at the end of the interrupt

stas

set address select

Selects hidden register ***IRA*** for access by **`mtsr`**/**`mfsr`** instructions with the **SA** special register number by setting the **AS** bit in register **CS**.

C language description

```
CS.AS = 1;
```

Addressing Modes	assembly format
implied	`stas`

Notes

Load/store instructions with the **SA** special register as destination or source register always access the **SA** physical register and never the ***IRA*** hidden register.

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

```
uint16 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 sf16 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															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
absl	Rs,Rd	1	1	Rs			0	0	0	Rd			0	1	1	1	0
adcf	Rs,Rd	1	1	Rs			0	0	1	Rd			0	0	1	1	0
addc	Rs0,Rs1,Rd	0	0	Rs0			Rs1			Rd			0	1	1	1	0
addh	C16,Rb	C16								Rb			0	0	1	1	1
addt	Rs0,Rs1,Rd	0	0	Rs0			Rs1			Rd			0	0	1	1	0
	$C8_{UN}$,Rb	$C8_{UN}$								Rb			0	1	1	1	1
adsp	$C7_{SN}$,Ad	1	0	$C7_{SN}$							Ad		1	1	0	1	0
andb	Rs0,Rs1,Rd	0	0	Rs0			Rs1			Rd			1	0	1	1	0
	$C8_U$,Rb	$C8_U$								Rb			1	0	1	1	1
bral	$IO8_S$	$IO8_S$								1	1	0	1	1	1	0	1
brcr	$IO8_S$	$IO8_S$								0	0	1	1	0	1	0	1
brge	$IO8_S$	$IO8_S$								0	1	1	1	1	1	0	1
brgt	$IO8_S$	$IO8_S$								1	0	1	1	1	1	0	1
brhi	$IO8_S$	$IO8_S$								0	0	1	1	1	1	0	1
brlc	$IO8_S$	$IO8_S$								1	1	1	1	1	1	0	1
brle	$IO8_S$	$IO8_S$								1	0	0	1	1	1	0	1
brlo	$IO8_S$	$IO8_S$								0	1	0	1	1	1	0	1
brls	$IO8_S$	$IO8_S$								0	0	0	1	1	1	0	1
brnc	$IO8_S$	$IO8_S$								0	0	0	1	0	1	0	1
brng	$IO8_S$	$IO8_S$								1	1	1	1	0	1	0	1
brno	$IO8_S$	$IO8_S$								0	1	0	1	0	1	0	1
brnz	$IO8_S$	$IO8_S$								1	0	0	1	0	1	0	1
brof	$IO8_S$	$IO8_S$								0	1	1	1	0	1	0	1
brps	$IO8_S$	$IO8_S$								1	1	0	1	0	1	0	1
brzr	$IO8_S$	$IO8_S$								1	0	1	1	0	1	0	1
btcl	Rs0,Rs1,Rd	0	1	Rs0			Rs1			Rd			1	0	0	1	0
	BTI4,Rb	1	1	BTI4				1	0	Rb			1	0	0	1	0
btst	Rs0,Rs1,Rd	0	1	Rs0			Rs1			Rd			1	1	0	1	0
	BTI4,Rb	1	1	BTI4				1	0	Rb			1	1	0	1	0
bttg	Rs0,Rs1,Rd	0	1	Rs0			Rs1			Rd			1	1	1	1	0
	BTI4,Rb	1	1	BTI4				1	0	Rb			1	1	1	1	0
btts	Rs0,Rs1	0	1	Rs0			Rs1			0	0	0	1	0	1	1	0
	BTI4,Rs	1	1	BTI4				1	0	Rs			1	0	1	1	0
clas	implied	0	0	0	0	0	0	0	1	1	0	1	0	1	1	0	1
clie	implied	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	1
clzr	Rs,Rd	1	1	Rs			0	0	0	Rd			1	0	1	1	0
cmpc	Rs0,Rs1	1	0	Rs0			Rs1			0	0	1	1	0	0	1	0
comp	Rs0,Rs1	1	0	Rs0			Rs1			0	0	0	1	0	0	1	0
	$C8_A$,Rs1	1	0	$C8_A$						Rs1			1	0	1	1	0
		$C8_A$								Rs1			1	1	1	1	1
cpcf	Rs	1	1	Rs			0	0	1	0	0	0	0	1	0	1	0
invt	Rs,Rd	1	1	Rs			0	0	0	Rd			1	0	0	1	0
jump	IA12	IA12												0	0	0	1
	implied	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1
jpsr	IA12	IA12												1	0	0	1
	implied	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1
ldbt	DA8,Rd	DA8								Rd			0	0	0	0	0
	($DO5_S$,An),Rd	$DO5_S$					0	An		Rd			0	0	1	0	0
	(Rx,An),Rd	0	0	Rx			1	An		Rd			0	0	1	0	0
	(An)+,Rd	0	1	0	0	0	1	An		Rd			0	0	1	0	0
	-(An),Rd	0	1	0	0	1	1	An		Rd			0	0	1	0	0
	(An)*,Rd	0	1	0	1	0	1	An		Rd			0	0	1	0	0
	(An)+,RGS	1	6	5	4	3	1	An		2	1	0	0	0	1	0	0
ldsh	DA8,Rd	DA8								Rd			0	1	0	0	0
	($DO5_S$,An),Rd	$DO5_S$					0	An		Rd			0	1	1	0	0
	(Rx,An),Rd	0	0	Rx			1	An		Rd			0	1	1	0	0
	(An)+,Rd	0	1	0	0	0	1	An		Rd			0	1	1	0	0
	-(An),Rd	0	1	0	0	1	1	An		Rd			0	1	1	0	0
	(An)*,Rd	0	1	0	1	0	1	An		Rd			0	1	1	0	0
	(An)+,RGS	1	6	5	4	3	1	An		2	T	S	0	1	1	0	0

mfdp	Rd	1	1	0	0	0	0	1	1	Rd			0	0	0	1	0
mfsr	SRs,Rd	1	1	SRs			0	1	1	Rd			0	1	0	1	0
mlhs	Rs0,Rs1,Rd	1	0	Rs0			Rs1			Rd			0	0	1	1	0
mlhu	Rs0,Rs1,Rd	1	0	Rs0			Rs1			Rd			0	0	0	1	0
move	Rs,Rd	1	1	Rs			0	0	0	Rd			0	0	0	1	0
	$C9_S$,Rd	$C9_S$								Rd			C9	0	0	1	1
mtdp	Rs	1	1	0	0	0	0	1	1	Rs			0	0	1	1	0
mtsr	Rs,SRd	1	1	SRd			0	1	1	Rs			0	1	1	1	0
	C7,SRLd	1	0	C7							SRLd		1	1	1	1	0
mult	Rs0,Rs1,Rd	0	0	Rs0			Rs1			Rd			1	0	0	1	0
negt	Rs,Rd	1	1	Rs			0	0	0	Rd			0	1	0	1	0
rsie	implied	0	0	0	0	0	0	0	1	0	1	0	0	1	1	0	1
rspc	implied	0	0	0	0	0	0	0	0	1	1	1	0	1	1	0	1
rtir	implied	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	1
rtsr	implied	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1
sbcf	Rs,Rd	1	1	Rs			0	0	1	Rd			0	0	0	1	0
shlf	Rs0,Rs1,Rd	0	1	Rs0			Rs1			Rd			0	1	0	1	0
	SHC4,Rb	1	1	SHC4				1	0	Rb			0	1	0	1	0
shlz	Rs0,Rs1,Rd	0	1	Rs0			Rs1			Rd			0	0	0	1	0
	SHC4,Rb	1	1	SHC4				1	0	Rb			0	0	0	1	0
shrs	Rs0,Rs1,Rd	0	1	Rs0			Rs1			Rd			0	1	1	1	0
	SHC4,Rb	1	1	SHC4				1	0	Rb			0	1	1	1	0
shru	Rs0,Rs1,Rd	0	1	Rs0			Rs1			Rd			0	0	1	1	0
	SHC4,Rb	1	1	SHC4				1	0	Rb			0	0	1	1	0
scie	implied	0	0	0	0	0	0	0	1	0	1	1	0	1	1	0	1
siah	IAH4	0	0	0	0	0	0	0	IAH4				0	0	1	0	1
stas	implied	0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	1
stbt	Rs,DA8	DA8								Rs			1	0	0	0	0
	Rs,($DO5_S$,An)	$DO5_S$					0	An		Rs			1	0	1	0	0
	Rs,(Rx,An)	0	0	Rx			1	An		Rs			1	0	1	0	0
	Rs,(An)+	0	1	0	0	0	1	An		Rs			1	0	1	0	0
	Rs,-(An)	0	1	0	0	1	1	An		Rs			1	0	1	0	0
	Rs,(An)*	0	1	0	1	0	1	An		Rs			1	0	1	0	0
	RGS,(An)+	1	0	1	2	3	1	An		4	5	6	1	0	1	0	0
stie	implied	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	1
stop	implied	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1
stsh	Rs,DA8	DA8								Rs			1	1	0	0	0
	Rs,($DO5_S$,An)	$DO5_S$					0	An		Rs			1	1	1	0	0
	Rs,(Rx,An)	0	0	Rx			1	An		Rs			1	1	1	0	0
	Rs,(An)+	0	1	0	0	0	1	An		Rs			1	1	1	0	0
	Rs,-(An)	0	1	0	0	1	1	An		Rs			1	1	1	0	0
	Rs,(An)*	0	1	0	1	0	1	An		Rs			1	1	1	0	0
	RGS,(An)+	1	S	T	2	3	1	An		4	5	6	1	1	1	0	0
subc	Rs0,Rs1,Rd	0	0	Rs0			Rs1			Rd			0	1	0	1	0
subf	Rs0,Rs1,Rd	0	0	Rs0			Rs1			Rd			0	0	0	1	0
	$C8_{UN}$,Rb	$C8_{UN}$								Rb			0	1	0	1	1
svpc	implied	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	1
sxbt	Rs,Rd	1	1	Rs			0	0	0	Rd			1	1	0	1	0
sxsh	Rs,Rd	1	1	Rs			0	0	0	Rd			1	1	1	1	0
xorb	Rs0,Rs1,Rd	0	0	Rs0			Rs1			Rd			1	1	1	1	0

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

Color	Description of table entries
(light blue)	Register select field, selects a register of the programming model
(orange)	Constant field
(yellow)	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 16 columns define how the multi-bit fields from the left column are mapped into 16-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.

Among the register specifications **RGS** is a special case. 7 bits of the opcode marked with single-characters represent the 7 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 **6** represent registers **R0** – **R6**
- 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 and the group of registers that can be part of an **RGS** is different for byte load/store instructions and for short load/store instructions.

Opcode field	opcode bits															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rs,Rs0,Rx,SRs,SRd			2	1	0											
Rs1						2	1	0								
Rs,Rs1,Rd,Rb									2	1	0					
An							1	0								
Ad,SRLd										1	0					
RGS for ldbt		6	5	4	3				2	1	0					
RGS for ldsh		6	5	4	3				2	T	S					
RGS for stbt		0	1	2	3				4	5	6					
RGS for stsh		S	T	2	3				4	5	6					
DA8	4	3	2	1	0	7	6	5								
DO5$_S$	4	3	2	1	0											
C7$_U$,C7$_{UN}$,C7$_{SN}$			5	4	3	2	1	0	6							
C8$_U$,IO8$_S$	7	6	5	4	3	2	1	0								
C8$_A$ positive	7	6	5	4	3	2	1	0								
C8$_A$ negative			5	4	3	2	1	0								
C9$_S$	7	6	5	4	3	2	1	0				8				
SHC4,BTI4			3	2	1	0										
IA12	11	10	9	8	7	6	5	4	3	2	1	0				
IAH4								3	2	1	0					