2 - STANDARD INSTRUCTION SET

2.1 - Addressing Modes

2.1.1 - Short adressing modes

The ST10 family of devices use several powerful addressing modes for access to word, byte and bit data. This section describes short, long and indirect address modes, constants and branch target addressing modes. Short addressing modes use an implicit base offset address to specify the 24-bit physical address. Short addressing modes give access to the GPR, SFR or bit-addressable memory spacePhysicalAddress = BaseAddress + Δ x ShortAddress.

Note: Δ = 1 for byte GPRs, Δ = 2 for word GPRs (see Table 1).

Rw, Rb

Specifies direct access to any GPR in the currently active context (register bank). Both ’Rw’ and ’Rb’ require four bits in the instruction format. The base address of the current register bank is determined by the content of register CP. ’Rw’ specifies a 4-bit word GPR address relative to the base address (CP), while ’Rb’ specifies a 4 bit byte GPR address relative to the base address (CP).

reg

Specifies direct access to any (E)SFR or GPR in the currently active context (register bank). ’reg’ requires eight bits in the instruction format. Short ’reg’ addresses from 00h to EFh always specify (E)SFRs. In this case, the factor ’Δ’ equals 2 and the base address is 00’F000h for the standard SFR area, or 00’FE00h for the extended ESFR area. ‘reg’ accesses to the ESFR area require a preceding EXT*R instruction to switch the base address. Depending on the opcode of an instruction, either the total word (for word operations), or the low byte (for byte operations) of an SFR can be addressed via 'reg'. Note that the high byte of an SFR cannot be accessed by the 'reg' addressing mode. Short 'reg' addresses from F0h to FFh always specify GPRs. In this case, only the lower four bits of 'reg' are significant for physical address generation, therefore it can be regarded as identical to the address generation described for the 'Rb' and 'Rw' addressing modes.

bitoff

Specifies direct access to any word in the bit-addressable memory space. 'bitoff' requires eight bits in the instruction format. Depending on the specified 'bitoff' range, different base addresses are used to generate physical addresses: Short 'bitoff' addresses from 00h to 7Fh use 00’FD00h as a base address, therefore they specify the 128 highest internal RAM word locations (00’FD00h to 00’FDFEh).Short 'bitoff' addresses from 80h to EFh use 00’FF00h as a base address to specify the highest internal SFR word locations (00’FF00h to 00’FFDEh) or use 00’F100h as a base address to specify the highest internal ESFR word locations (00’F100h to 00’F1DEh). ‘bitoff’ accesses to the ESFR area require a preceding EXT*R instruction to switch the base address. For short 'bitoff' addresses from F0h to FFh, only the lowest four bits and the contents of the CP register are used to generate the physical address of the selected word GPR.

bitaddr

Any bit address is specified by a word address within the bit-addressable memory space (see 'bitoff'), and by a bit position ('bitpos') within that word. Thus, 'bitaddr' requires twelve bits in the instruction format.

Table 1 : Short addressing mode summary

Mnemo	Physical Address		Short Address Range		Scope of Access	
Rw	(CP)	+ 2*Rw	Rw	= 0...15	GPRs	(Word) 16 values
Rb	(CP)	+ 1*Rb	Rb	= 0...15	GPRs	(Byte) 16 values
reg	00’FE00h 00’F000h (CP) (CP)	+ 2*reg + 2*reg + 2*(reg^0Fh) + 1*(reg^0Fh)	reg reg reg reg	= 00h...EFh = 00h...EFh = F0h...FFh = F0h...FFh	SFRs ESFRs GPRs GPRs	(Word, Low byte) (Word, Low byte) (Word) 16 values (Bytes) 16 values
bitoff	00’FD00h 00’FF00h (CP)	+ 2*bitoff + 2*(bitoff^FFh) + 2*(bitoff^0Fh)	bitoff bitoff bitoff	= 00h...7Fh = 80h...EFh = F0h...FFh	RAM SFR GPR	Bit word offset 128 values Bit word offset 128 values Bit word offset 16 values
bitaddr	Word offset as with bitoff Immediate bit position		bitoff bitpos	= 00h...FFh = 0...15	Any single bit	

2.1.2 - Long addressing mode

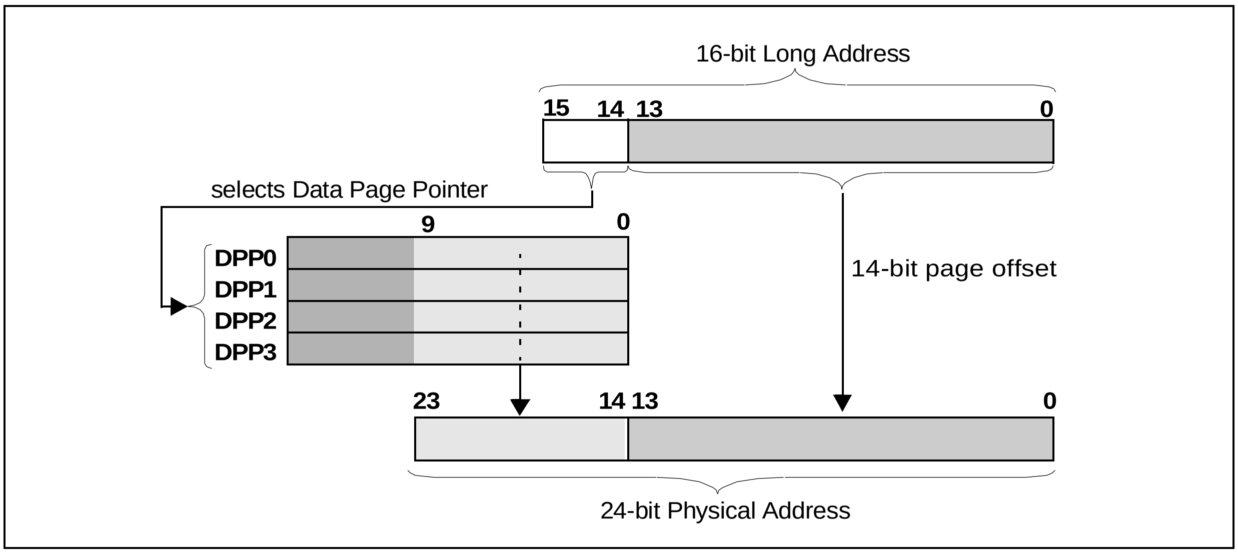
Long addressing mode uses one of the four DPP registers to specify a physical 18-bit or 24-bit address. Any word or byte data within the entire address space can be accessed in this mode. All devices support an override mechanism for the DPP addressing scheme (see section 2.1.3 - DPP override mechanism).

Long addresses (16-bit) are treated in two parts. Bits 13...0 specify a 14-bit data page offset, and bits 15...14 specify the Data Page Pointer (1 of 4). The DPP is used to generate the physical 24-bit address (see Figure 1).

All ST10 devices support an address space of up to 16MByte, so only the lower ten bits of the selected DPP register content are concatenated with the 14-bit data page offset to build the physical address.

Note: Word accesses on odd byte addresses are not executed, but rather trigger a hardware trap. After reset, the DPP registers are initialized so that all long addresses are directly mapped onto the identical physical addresses, within segment 0.

Figure 1 : Interpretation of a 16-bit long address



The long addressing mode is referred to by the mnemonic “mem”.

Table 2 : Summary of long address modes

Mnemo	Physical Address		Long Address Range	Scope of Access
mem	(DPP0)	|| mem^3FFFh	0000h...3FFFh	Any Word or Byte
	(DPP1)	|| mem^3FFFh	4000h...7FFFh	
	(DPP2)	|| mem^3FFFh	8000h...BFFFh	
	(DPP3)	|| mem^3FFFh	C000h...FFFFh	
mem	pag	|| mem^3FFFh	0000h...FFFFh (14-bit)	Any Word or Byte
mem	seg	|| mem	0000h...FFFFh (16-bit)	Any Word or Byte

2.1.3 - DPP override mechanism

The DPP override mechanism temporarily bypasses the DPP addressing scheme. The EXTP(R) and EXTS(R) instructions override this addressing mechanism. Instruction EXTP(R) replaces the content of the respective DPP register, while instruction EXTS(R) concatenates the complete 16-bit long address with the specified segment base address. The overriding page or segment may be specified directly as a constant (#pag, #seg) or by a word GPR (Rw) (see Figure 2).

2.1.4 - Indirect addressing modes

Indirect addressing modes can be considered as a combination of short and long addressing modes. In this mode, long 16-bit addresses are specified indirectly by the contents of a word GPR, which is specified directly by a short 4-bit address ('Rw'=0 to 15). Some indirect addressing modes add a constant value to the GPR contents before the long 16-bit address is calculated. Other indirect addressing modes allow decrementing or incrementing of the indirect address pointers (GPR content) by 2 or 1 (referring to words or bytes).

In each case, one of the four DPP registers is used to specify the physical 18-bit or 24-bit addresses. Any word or byte data within the entire memory space can be addressed indirectly. Note that EXTP(R) and EXTS(R) instructions override the DPP mechanism.

Instructions using the lowest four word GPRs (R3...R0) as indirect address pointers are specified by short 2-bit addresses.

Word accesses on odd byte addresses are not executed, but rather trigger a hardware trap.

After reset, the DPP registers are initialized in a way that all indirect long addresses are directly mapped onto the identical physical addresses.

Physical addresses are generated from indirect address pointers by the following algorithm:

1. Calculate the physical address of the word GPR which is used as indirect address pointer, by using the specified short address ('Rw') and the current register bank base address (CP).

GPRAddress = (CP) + 2 x ShortAddress

2. Pre-decremented indirect address pointers ('-Rw') are decremented by a data-type-dependent value (Δ = 1 for byte operations, Δ = 2 for word operations), before the long 16-bit address is generated:

(GPRAddress) = (GPRAddress) - Δ [optional step!]

3. Calculate the long 16-bit (Rw + #data16 if selected) address by adding a constant value (if selected) to the content of the indirect address pointer:

Long Address = (GPR Address) + Constant

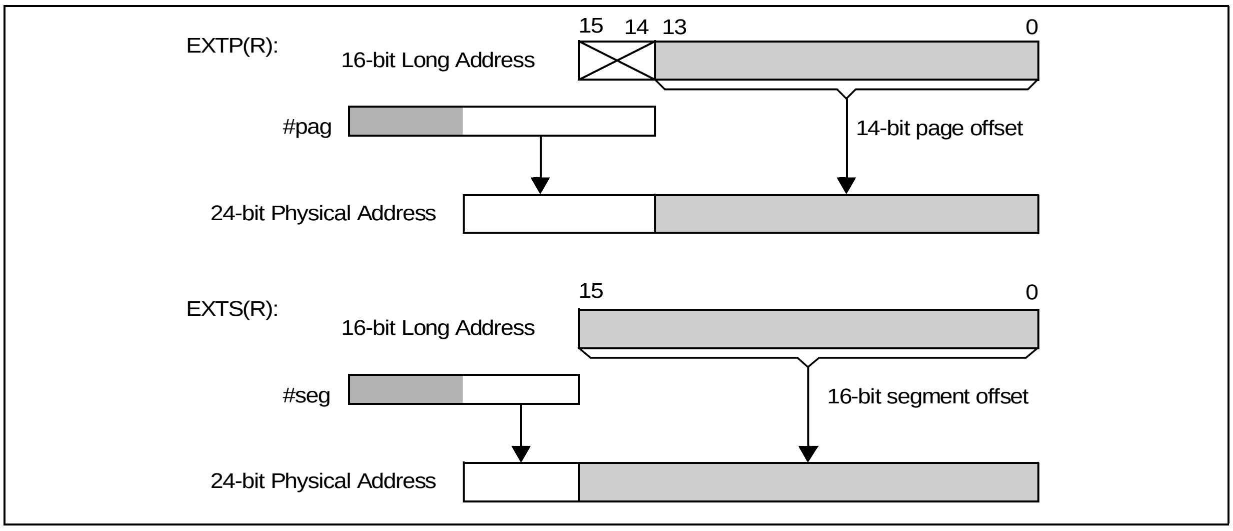
4. Calculate the physical 18-bit or 24-bit address using the resulting long address and the corresponding DPP register content (see long 'mem' addressing modes).

Physical Address = (DPPi) + Long Address^3FFFh

5. Post-Incremented indirect address pointers ('Rw+') are incremented by a data-type-dependent value (Δ = 1 for byte operations, Δ = 2 for word operations):

(GPR Address) = (GPR Address) + Δ [optional step!]

Figure 2 : Overriding the DPP mechanism



The following indirect addressing modes are provided:

Table 3 : Table of indirect address modes

Mnemonic	Notes
[Rw]	Most instructions accept any GPR (R15...R0) as indirect address pointer. Some instructions, however, only accept the lower four GPRs (R3...R0).
[Rw+]	The specified indirect address pointer is automatically incremented by 2 or 1 (for word or byte data operations) after the access.
[-Rw]	The specified indirect address pointer is automatically decremented by 2 or 1 (for word or byte data operations) before the access.
[Rw+#data$_{16}$]	A 16-bit constant and the contents of the indirect address pointer are added before the long 16-bit address is calculated.

2.1.5 - Constants

The ST10 Family instruction set supports the use of wordwide or bytewide immediate constants.

For optimum utilization of the available code storage, these constants are represented in the instruction formats by either 3, 4, 8 or 16 bits.

Therefore, short constants are always zero-extended, while long constants can be truncated to match the data format required for the operation:

Table 4 : Table of constants

Mnemonic	Word operation	Byte operation
#data$_3$	0000$_h$ + data$_3$	00$_h$ + data$_3$
#data$_4$	0000$_h$ + data$_4$	00$_h$ + data$_4$
#data$_8$	0000$_h$ + data$_8$	data$_8$
#data$_{16}$	data$_{16}$	data$_{16}$ ^ FF$_h$
#mask	0000$_h$ + mask	mask

Note: Immediate constants are always signified by a leading number sign “#”.

2.1.6 - Branch target addressing modes

Jump and Call instructions use different addressing modes to specify the target address and segment.

Relative, absolute and indirect modes can be used to update the Instruction Pointer register (IP), while the Code Segment Pointer register (CSP) can only be updated with an absolute value.

A special mode is provided to address the interrupt and trap jump vector table situated in the lowest portion of code segment 0.

Table 5 : Branch target address summary

Mnemonic	Target Address		Target Segment	Valid Address Range	
caddr	(IP)	= caddr	-	caddr	= 0000h...FFFEh
rel	(IP)	= (IP) + 2*rel	-	rel	= 00h...7Fh
	(IP)	= (IP) + 2*(~rel+1)	-	rel	= 80h...FFh
[Rw]	(IP)	= ((CP) + 2*Rw)	-	Rw	= 0...15
seg	-		(CSP) = seg	seg	= 0...255
#trap$_7$	(IP)	= 0000h + 4*trap$_7$	(CSP) = 0000h	trap$_7$	= 00h...7Fh

caddr

Specifies an absolute 16-bit code address within the current segment. Branches MAY NOT be taken to odd code addresses.

Therefore, the least significant bit of ’caddr’ must always contain a ’0’, otherwise a hardware trap would occur.

rel

Represents an 8-bit signed word offset address relative to the current Instruction Pointer contents which points to the instruction after the branch instruction.

Depending on the offset address range, either forward (’rel’= 00h to 7Fh) or backward (’rel’= 80h to FFh) branches are possible.

The branch instruction itself is repeatedly executed, when ’rel’ = ’-1’ (FF_h) for a word-sized branch instruction, or ’rel’ = ’-2’ (FEh) for a double-word-sized branch instruction.

[Rw]

The 16-bit branch target instruction address is determined indirectly by the content of a word GPR. In contrast to indirect data addresses, indirectly specified code addresses are NOT calculated by additional pointer registers (e.g. DPP registers).

Branches MAY NOT be taken to odd code addresses. Therefore, to prevent a hardware trap, the least significant bit of the address pointer GPR must always contain a ’0.

seg

Specifies an absolute code segment number. All devices support 256 different code segments, so only the eight lower bits of the ’seg’ operand value are used for updating the CSP register.

#$trap_7$

Specifies a particular interrupt or trap number for branching to the corresponding interrupt or trap service routine by a jump vector table.

Trap numbers from 00h to 7Fh can be specified, which allows access to any double word code location within the address range 00’0000h...00’01FCh in code segment 0 (i.e. the interrupt jump vector table).

For further information on the relation between trap numbers and interrupt or trap sources, refer to the device user manual section on “Interrupt and Trap Functions”.

2.2 - Instruction execution times

The instruction execution time depends on where the instruction is fetched from, and where the operands are read from or written to.

The fastest processing mode is to execute a program fetched from the internal ROM. In this case most of the instructions can be processed in just one machine cycle.

All external memory accesses are performed by the on-chip External Bus Controller (EBC) which works in parallel with the CPU.

Instructions from external memory cannot be processed as fast as instructions from the internal ROM, because it is necessary to perform data transfers sequentially via the external interface.

In contrast to internal ROM program execution, the time required to process an external program additionally depends on the length of the instructions and operands, on the selected bus mode, and on the duration of an external memory cycle.

Processing a program from the internal RAM space is not as fast as execution from the internal ROM area, but it is flexible (i.e. for loading temporary programs into the internal RAM via the chip's serial interface, or end-of-line programming via the bootstrap loader).

The following description evaluates the minimum and maximum program execution times. which is sufficient for most requirements. For an exact determination of the instructions' state times, the facilities provided by simulators or emulators should be used.

This section defines measurement units, summarizes the minimum (standard) state times of the 16-bit microcontroller instructions, and describes the exceptions from the standard timing.

2.3 - Instruction set summary

The following table lists the instruction mnemonic by hex-code with operand.

Table 7 : Instruction mnemonic by hex-code with operand

High / Low	xF	xE	xD	xC	xB	xA	x9	x8	x7	x6	x5	x4	x3	x2	x1	x0
0x	BSET BITaddrQ.q	BCLR BITaddrQ.q	JMPR cc, rel	ROL Rw_n, Rw_m	MUL Rw_n, Rw_m	BFLDL BITOFF, MASK, $\#data_3$	ADDB	ADD Rw_n, [Rw_i] Rw_n, [Rw_i+] Rw_n, $\#data_3$	ADDB	ADD REG, $\#data_{16}$	ADDB	ADD MEM, REG	ADDB	ADD REG, MEM	ADDB	ADD Rw_n, Rw_m
1x				ROL Rw_n, $\#d_4$	MULU	BFLDH	ADDCB	ADDC	ADDCB	ADDC	ADDCB	ADDC	ADDCB	ADDC	ADDCB	ADDC
2x				ROR Rw_n, Rw_m	PRIOR Rw_n, Rw_m	BCMP BITadd, BITadd	SUBB	SUB Rw_n, [Rw_i] Rw_n, [Rw_i+] Rw_n, $\#data_3$	SUBB	SUB REG, $\#data_{16}$	SUBB	SUB MEM, REG	SUBB	SUB REG, MEM	SUBB	SUB Rw_n, Rw_m
3x				ROR Rw_n, $\#d_4$	–	BMOVN BITadd, BITadd	SUBCB	SUBC	SUBCB	SUBC	SUBCB	SUBC	SUBCB	SUBC	SUBCB	SUBC
4x				SHL Rw_n, Rw_m	DIV Rw_n	BMOV BITadd, BITadd	CMPB	CMP Rw_n, [Rw_i] Rw_n, [Rw_i+] Rw_n, $\#data_3$	CMPB	CMP REG, $\#data_{16}$	–	– MEM, REG	CMPB	CMP REG, MEM	CMPB	CMP Rw_n, Rw_m
5x				SHL Rw_n, $\#d_4$	DIVU Rw_n	BOR BITadd, BITadd	XORB	XOR	XORB	XOR	XORB	XOR	XORB	XOR	XORB	XOR
6x				SHR Rw_n, Rw_m	DIVL Rw_n	BAND BITadd, BITadd	ANDB	AND Rw_n, [Rw_i] Rw_n, [Rw_i+] Rw_n, $\#data_3$	ANDB	AND REG, $\#data_{16}$	ANDB	AND MEM, REG	ANDB	AND REG, MEM	ANDB	AND Rw_n, Rw_m
7x				SHR Rw_n, $\#d_4$	DIVLU Rw_n	BXOR BITadd, BITadd	ORB	OR	ORB	OR	ORB	OR	ORB	OR	ORB	OR
8x				–	–	JB BITadd, REL	MOVB	MOV [-Rw_m], Rw_n	IDLE	CMPI1 Rw_n, $\#d_{16}$	–	MOV [Rw_n], MEM	CoXXX Rw_n, [Rw_m⊗]	CMPI Rw_n, MEM	NEG Rw_n	CMPI1 Rw_n, $\#d_4$
9x				JMPI cc, [Rw_n]	TRAP #trap	JNB	MOVB	MOV Rw_n, [Rw_m+]	PWRDN	CMPI2	–	MOV MEM, [Rw_n]	CoXXX [IDXI⊗], [Rw_m⊗]	CMPD2	CPLB	CMPD2
Ax				ASHR Rw_n, Rw_m	CALLI cc, [Rw_n]	JBC BITadd, BITadd	MOVB	MOV [Rw_n], Rw_m	SRVWDT	CMPD1 Rw_n, $\#d_{16}$	DISWDT	MOVB [Rw_n], MEM	CoXXX Rw_n, Rw_m	CMPD1 Rw_n, MEM	NEGB Rw_n	CMPD1 Rw_n, $\#d_4$
Bx				ASHR Rw_n, $\#d_4$	CALLR REL	JNBS	MOVB	MOV [Rw_m], Rw_n	SRST	CMPD2	EINIT	MOVB [Rw_m + $\#d_{16}$], Rw_n	CoSTORE Rw_n, CoREG	CMPD2	CPLB	CMPD2
Cx				NOP	RET	CALLA CC, CADDR	MOVB	MOV [Rw_n], [Rw_m]	–	SCXT REG, $\#d_{16}$	MOVBZ MEM, REG	MOV Rw_n, [Rw_m + $\#d_{16}$]	CoSTORE [Rw_n ⊗], CoREG	MOVBZ REG, MEM	–	MOVBZ Rw_n, Rw_m
Dx				EXTP(R)/ EXTS(R) Rw_m, $\#d_2$	RETS	CALLS SEG, CADDDR	MOVB	MOV [Rw_{n+}], [Rw_m]	EXTP(R)/ EXTS(R) #pag, #data2	SCXT REG, MEM	MOVBS	MOV [Rw_m + $\#d_{16}$], Rw_n	CoMOV [IDXI⊗], [Rw_m⊗]	MOVBS	ATOMIC/EXTR $\#data_2$	MOVBS
Ex				PUSH REG	RETP REG	JMPA CC, CADDR	MOVB	MOV [Rw_n], [Rw_m+]	MOVB	MOV REG, Data$\#_{16}$	–	MOVB Rw_n, [Rw_m + $\#d_{16}$]	–	PCALL REG, CADDR	MOVB	MOV Rw_n, $\#data_4$
Fx				POP	RETI	JMPS SEG, CADDR	–	–	MOVB	MOV MEM, REG	–	MOVB [Rw_m + $\#d_{16}$], Rw_n	MOVB	MOV REG, MEM	MOVB	MOV Rw_n, $\#data_4$

Table 8 lists the instructions by their mnemonic and identifies the addressing modes that may be used with a specific instruction and the instruction length, depending on the selected addressing mode (in bytes).

Table 8 : Mnemonic vs address mode & number of bytes

Mnemonic	Addressing Modes	Bytes
ADD[B]	Rw_n^1, Rw_m ^{1}	2
ADDC[B]	Rw_n^1, $[Rw_i]$	2
AND[B]	Rw_n^1, $[Rw_i+]$	2
OR[B]	Rw_n^1, $\#data_3$	2
SUB[B]	reg, $\#data_{16}$	4
SUBC[B]	reg, mem	4
XOR[B]	mem, reg	4
ASHR	Rw_n, Rw_m	2
ROL / ROR	Rw_n, $\#data_4$	2
SHL / SHR		
BAND	$bitaddr_{Z.z}$, $bitaddr_{Q.q}$	4
BCMP		
BMOV		
BMOVN		
BOR / BXOR		
BCLR	$bitaddr_{Q.q}$,	2
BSET		
BFLDH	$bitoff_Q$, $\#mask_8$, $\#data_8$	4
BFLDL		
MOV[B]	Rw_n^1, Rw_m^1	2
	Rw_n^1, $\#data_4$	2
	Rw_n^1, $[Rw_m]$	2
	Rw_n^1, $[Rw_m+]$	2
	$[Rw_m]$, Rw_n^1	2
	$[-Rw_m]$, Rw_n ^{1}	2
	$[Rw_n]$, $[Rw_m]$	2
	$[Rw_n+]$, $[Rw_m]$	2
	$[Rw_n]$, $[Rw_m+]$	2
	reg, $\#data_{16}$	4
	Rw_n, $[Rw_m+\#data_{16}]^1$	4
	$[Rw_m+\#data_{16}]$, Rw_n ^{1}	4
	$[Rw_n]$, mem	4
	mem, $[Rw_n]$	4
	reg, mem	4
	mem, reg	4

Mnemonic	Addressing Modes	Bytes
CPL[B]	Rw_n^1	2
NEG[B]		
DIV	Rw_n	2
DIVL		
DIVLU		
DIVU		
MUL	Rw_n, Rw_m	2
MULU		
CMPD1/2	Rw_n, $\#data_4$	2
CMPI1/2	Rw_n, $\#data_{16}$	4
	Rw_n, mem	4
CMP[B]	Rw_n, Rw_m ^{1}	
	Rw_n, $[Rw_i]$ ^{1}	2
	Rw_n, $[Rw_i+]^1$	2
	Rw_n, $\#data_3{}^1$	2
	reg, $\#data_{16}$	4
	reg, mem	4
CALLA	cc, caddr	4
JMPA		
CALLI	cc, $[Rw_n]$	2
JMPI		
CALLS	seg, caddr	4
JMPS		
CALLR	rel	2
JMPR	cc, rel	2
JB	$bitaddr_{Q.q}$, rel	4
JBC		
JNB		
JNBS		
PCALL	reg, caddr	4
POP	reg	2
PUSH		
RETP		
SCXT	reg, $\#data_{16}$	4
	reg, mem	4
PRIOR	Rw_n, Rw_m	2

Table 8 : Mnemonic vs address mode & number of bytes (continued)

Mnemonic	Addressing Modes	Bytes		Mnemonic	Addressing Modes	Bytes
MOVBS	Rw_n, Rb_m	2		TRAP	#trap7	2
MOVBZ	reg, mem	4		ATOMIC	#$data_2$	2
	mem, reg	4		EXTR		
EXTS	Rw_m, #$data_2$	2		EXTP	Rw_m, #$data_2$	2
EXTSR	#seg, #$data_2$	4		EXTPR	#pag, #$data_2$	4
NOP RET RETI RETS	- =	2		SRST/IDLE PWRDN SRVWDT DISWDT EINIT	-	4

Note 1. Byte oriented instructions (suffix 'B') use Rb instead of Rw (not with [Rw_i]!).

2.4 - Instruction set ordered by functional group

The minimum number of state times required for instruction execution are given for the following configurations: internal ROM, internal RAM, external memory with a 16-bit demultiplexed and multiplexed bus or an 8-bit demultiplexed and multiplexed bus. These state time figures do not take into account possible wait states on external busses or possible additional state times induced by operand fetches. The following notes apply to this summary:

Data addressing modes

Rw:	Word GPR (R0, R1, … , R15).
Rb:	Byte GPR (RL0, RH0, …, RL7, RH7).
reg:	SFR or GPR (in case of a byte operation on an SFR, only the low byte can be accessed via 'reg').
mem:	Direct word or byte memory location.
[…]:	Indirect word or byte memory location. (Any word GPR can be used as indirect address pointer, except for the arithmetic, logical and compare instructions, where only R0 to R3 are allowed).
bitaddr:	Direct bit in the bit-addressable memory area.
bitoff:	Direct word in the bit-addressable memory area.
#$data_x$:	Immediate constant (the number of significant bits that can be user-specified is given by the appendix "x").
#$mask_8$:	Immediate 8-bit mask used for bit-field modifications.

Multiply and divide operations

The MDL and MDH registers are implicit source and/or destination operands of the multiply and divide instructions.

Branch target addressing modes

caddr:	Direct 16-bit jump target address (Updates the Instruction Pointer).
seg:	Direct 8-bit segment address (Updates the Code Segment Pointer).
rel:	Signed 8-bit jump target word offset address relative to the Instruction Pointer of the following instruction.
#trap7:	Immediate 7-bit trap or interrupt number.

Extension operations

The EXT* instructions override the standard DPP addressing scheme:

#pag:	Immediate 10-bit page address.
#seg:	Immediate 8-bit segment address.

Branch condition codes

cc: Symbolically specifiable condition codes

cc_UC	Unconditional	cc_NE	Not Equal
cc_Z	Zero	cc_ULT	Unsigned Less Than
cc_NZ	Not Zero	cc_ULE	Unsigned Less Than or Equal
cc_V	Overflow	cc_UGE	Unsigned Greater Than or Equal
cc_NV	No Overflow	cc_UGT	Unsigned Greater Than
cc_N	Negative	cc_SLE	Signed Less Than or Equal
cc_NN	Not Negative	cc_SLT	Signed Less Than
cc_C	Carry	cc_SGE	Signed Greater Than or Equal
cc_NC	No Carry	cc_SGT	Signed Greater Than
cc_EQ	Equal	cc_NET	Not Equal and Not End-of-Table

Table 9 : Arithmetic instructions

Mnemonic		Description	Int.ROM	Int.RAM	16-bit N-Mux	16-bit Mux	8-bit N-Mux	8-bit Mux	Bytes
ADD	Rw, Rw	Add direct word GPR to direct GPR	2	6	2	3	4	6	2
ADD	Rw, [Rw]	Add indirect word memory to direct GPR	2	6	2	3	4	6	2
ADD	Rw, [Rw+]	Add indirect word memory to direct GPR and post-increment source pointer by 2	2	6	2	3	4	6	2
ADD	Rw, #data$_3$	Add immediate word data to direct GPR	2	6	2	3	4	6	2
ADD	reg, #data$_{16}$	Add immediate word data to direct register	2	8	4	6	8	12	4
ADD	reg, mem	Add direct word memory to direct register	2	8	4	6	8	12	4
ADD	mem, reg	Add direct word register to direct memory	2	8	4	6	8	12	4
ADDB	Rb, Rb	Add direct byte GPR to direct GPR	2	6	2	3	4	6	2
ADDB	Rb, [Rw]	Add indirect byte memory to direct GPR	2	6	2	3	4	6	2
ADDB	Rb, [Rw+]	Add indirect byte memory to direct GPR and post-increment source pointer by 1	2	6	2	3	4	6	2
ADDB	Rb, #data$_3$	Add immediate byte data to direct GPR	2	6	2	3	4	6	2
ADDB	reg, #data$_{16}$	Add immediate byte data to direct register	2	8	4	6	8	12	4
ADDB	reg, mem	Add direct byte memory to direct register	2	8	4	6	8	12	4
ADDB	mem, reg	Add direct byte register to direct memory	2	8	4	6	8	12	4
ADDC	Rw, Rw	Add direct word GPR to direct GPR with Carry	2	6	2	3	4	6	2
ADDC	Rw, [Rw]	Add indirect word memory to direct GPR with Carry	2	6	2	3	4	6	2
ADDC	Rw, [Rw+]	Add indirect word memory to direct GPR with Carry and post-increment source pointer by 2	2	6	2	3	4	6	2
ADDC	Rw, #data$_3$	Add immediate word data to direct GPR with Carry	2	6	2	3	4	6	2
ADDC	reg, #data$_{16}$	Add immediate word data to direct register with Carry	2	8	4	6	8	12	4
ADDC	reg, mem	Add direct word memory to direct register with Carry	2	8	4	6	8	12	4
ADDC	mem, reg	Add direct word register to direct memory with Carry	2	8	4	6	8	12	4

Table 9 : Arithmetic instructions (continued)

Mnemonic		Description	Int.ROM	Int.RAM	16-bit N-Mux	16-bit Mux	8-bit N-Mux	8-bit Mux	Bytes
ADDCB	Rb, Rb	Add direct byte GPR to direct GPR with Carry	2	6	2	3	4	6	2
ADDCB	Rb, [Rw]	Add indirect byte memory to direct GPR with Carry	2	6	2	3	4	6	2
ADDCB	Rb, [Rw+]	Add indirect byte memory to direct GPR with Carry and post-increment source pointer by 1	2	6	2	3	4	6	2
ADDCB	Rb, #data$_3$	Add immediate byte data to direct GPR with Carry	2	6	2	3	4	6	2
ADDCB	reg, #data$_{16}$	Add immediate byte data to direct register with Carry	2	8	4	6	8	12	4
ADDCB	reg, mem	Add direct byte memory to direct register with Carry	2	8	4	6	8	12	4
ADDCB	mem, reg	Add direct byte register to direct memory with Carry	2	8	4	6	8	12	4
CPL	Rw	Complement direct word GPR	2	6	2	3	4	6	2
CPLB	Rb	Complement direct byte GPR	2	6	2	3	4	6	2
DIV	Rw	Signed divide register MDL by direct GPR (16-/16-bit)	20	24	20	21	22	24	2
DIVL	Rw	Signed long divide register MD by direct GPR (32-/16-bit)	20	24	20	21	22	24	2
DIVLU	Rw	Unsigned long divide register MD by direct GPR (32-/16-bit)	20	24	20	21	22	24	2
DIVU	Rw	Unsigned divide register MDL by direct GPR (16-/16-bit)	20	24	20	21	22	24	2
MUL	Rw, Rw	Signed multiply direct GPR by direct GPR (16-16-bit)	10	14	10	11	12	14	2
MULU	Rw, Rw	Unsigned multiply direct GPR by direct GPR (16-16-bit)	10	14	10	11	12	14	2
NEG	Rw	Negate direct word GPR	2	6	2	3	4	6	2
NEGB	Rb	Negate direct byte GPR	2	6	2	3	4	6	2
SUB	Rw, Rw	Subtract direct word GPR from direct GPR	2	6	2	3	4	6	2
SUB	Rw, [Rw]	Subtract indirect word memory from direct GPR	2	6	2	3	4	6	2
SUB	Rw, [Rw+]	Subtract indirect word memory from direct GPR & post-increment source pointer by 2	2	6	2	3	4	6	2
SUB	Rw, #data$_3$	Subtract immediate word data from direct GPR	2	6	2	3	4	6	2
SUB	reg, #data$_{16}$	Subtract immediate word data from direct register	2	8	4	6	8	12	4
SUB	reg, mem	Subtract direct word memory from direct register	2	8	4	6	8	12	4
SUB	mem, reg	Subtract direct word register from direct memory	2	8	4	6	8	12	4
SUBB	Rb, Rb	Subtract direct byte GPR from direct GPR	2	6	2	3	4	6	2
SUBB	Rb, [Rw]	Subtract indirect byte memory from direct GPR	2	6	2	3	4	6	2
SUBB	Rb, [Rw+]	Subtract indirect byte memory from direct GPR & post-increment source pointer by 1	2	6	2	3	4	6	2
SUBB	Rb, #data$_3$	Subtract immediate byte data from direct GPR	2	6	2	3	4	6	2
SUBB	reg, #data$_{16}$	Subtract immediate byte data from direct register	2	8	4	6	8	12	4

Table 9 : Arithmetic instructions (continued)

Mnemonic		Description	Int.ROM	Int.RAM	16-bit N-Mux	16-bit Mux	8-bit N-Mux	8-bit Mux	Bytes
SUBB	reg, mem	Subtract direct byte memory from direct register	2	8	4	6	8	12	4
SUBB	mem, reg	Subtract direct byte register from direct memory	2	8	4	6	8	12	4
SUBC	Rw, Rw	Subtract direct word GPR from direct GPR with Carry	2	6	2	3	4	6	2
SUBC	Rw, [Rw]	Subtract indirect word memory from direct GPR with Carry	2	6	2	3	4	6	2
SUBC	Rw, [Rw+]	Subtract indirect word memory from direct GPR with Carry and post-increment source pointer by 2	2	6	2	3	4	6	2
SUBC	Rw, #$data_3$	Subtract immediate word data from direct GPR with Carry	2	6	2	3	4	6	2
SUBC	reg, #$data_{16}$	Subtract immediate word data from direct register with Carry	2	8	4	6	8	12	4
SUBC	reg, mem	Subtract direct word memory from direct register with Carry	2	8	4	6	8	12	4
SUBC	mem, reg	Subtract direct word register from direct memory with Carry	2	8	4	6	8	12	4
SUBCB	Rb, Rb	Subtract direct byte GPR from direct GPR with Carry	2	6	2	3	4	6	2
SUBCB	Rb, [Rw]	Subtract indirect byte memory from direct GPR with Carry	2	6	2	3	4	6	2
SUBCB	Rb, [Rw+]	Subtract indirect byte memory from direct GPR with Carry and post-increment source pointer by 1	2	6	2	3	4	6	2
SUBCB	Rb, #$data_3$	Subtract immediate byte data from direct GPR with Carry	2	6	2	3	4	6	2
SUBCB	reg, #$data_{16}$	Subtract immediate byte data from direct register with Carry	2	8	4	6	8	12	4
SUBCB	reg, mem	Subtract direct byte memory from direct register with Carry	2	8	4	6	8	12	4
SUBCB	mem, reg	Subtract direct byte register from direct memory with Carry	2	8	4	6	8	12	4

Table 10 : Logical instructions

Mnemonic		Description	Int ROM	Int. RAM	16-bit N-Mux	16-bit Mux	8-bit N-Mux	8-bit MUX	Bytes
AND	Rw, Rw	Bitwise AND direct word GPR with direct GPR	2	6	2	3	4	6	2
AND	Rw, [Rw]	Bitwise AND indirect word memory with direct GPR	2	6	2	3	4	6	2
AND	Rw, [Rw+]	Bitwise AND indirect word memory with direct GPR and post-increment source pointer by 2	2	6	2	3	4	6	2
AND	Rw, #$data_3$	Bitwise AND immediate word data with direct GPR	2	6	2	3	4	6	2
AND	reg, #$data_{16}$	Bitwise AND immediate word data with direct register	2	8	4	6	8	12	4
AND	reg, mem	Bitwise AND direct word memory with direct register	2	8	4	6	8	12	4
AND	mem, reg	Bitwise AND direct word register with direct memory	2	8	4	6	8	12	4
ANDB	Rb, Rb	Bitwise AND direct byte GPR with direct GPR	2	6	2	3	4	6	2
ANDB	Rb, [Rw]	Bitwise AND indirect byte memory with direct GPR	2	6	2	3	4	6	2

Table 11 : Boolean bit map instructions (continued)

Mnemonic	Description	Int. ROM	Int. RAM	16-bit N-Mux	16-bit Mux	8-bit N-Mux	8-bit Mux	Bytes
BAND bitaddr, bitaddr	AND direct bit with direct bit	2	8	4	6	8	12	4
BCLR bitaddr	Clear direct bit	2	6	2	3	4	6	2
BCMP bitaddr, bitaddr	Compare direct bit to direct bit	2	8	4	6	8	12	4
BFLDH bitoff, #mask$_8$,#data$_8$	Bitwise modify masked high byte of bit-addressable direct word memory with immediate data	2	8	4	6	8	12	4
BFLDL bitoff, #mask$_8$, #data$_8$	Bitwise modify masked low byte of bit-addressable direct word memory with immediate data	2	8	4	6	8	12	4
BMOV bitaddr, bitaddr	Move direct bit to direct bit	2	8	4	6	8	12	4
BMOVN bitaddr, bitaddr	Move negated direct bit to direct bit	2	8	4	6	8	12	4
BOR bitaddr, bitaddr	OR direct bit with direct bit	2	8	4	6	8	12	4
BSET bitaddr	Set direct bit	2	6	2	3	4	6	2
BXOR bitaddr, bitaddr	XOR direct bit with direct bit	2	8	4	6	8	12	4
CMP Rw, Rw	Compare direct word GPR to direct GPR	2	6	2	3	4	6	2
CMP Rw, [Rw]	Compare indirect word memory to direct GPR	2	6	2	3	4	6	2
CMP Rw, [Rw+]	Compare indirect word memory to direct GPR and post-increment source pointer by 2	2	6	2	3	4	6	2
CMP Rw, #data$_3$	Compare immediate word data to direct GPR	2	6	2	3	4	6	2
CMP reg, #data$_{16}$	Compare immediate word data to direct register	2	8	4	6	8	12	4
CMP reg, mem	Compare direct word memory to direct register	2	8	4	6	8	12	4
CMPB Rb, Rb	Compare direct byte GPR to direct GPR	2	6	2	3	4	6	2
CMPB Rb, [Rw]	Compare indirect byte memory to direct GPR	2	6	2	3	4	6	2
CMPB Rb, [Rw+]	Compare indirect byte memory to direct GPR and post-increment source pointer by 1	2	6	2	3	4	6	2
CMPB Rb, #data$_3$	Compare immediate byte data to direct GPR	2	6	2	3	4	6	2
CMPB reg, #data$_{16}$	Compare immediate byte data to direct register	2	8	4	6	8	12	4
CMPB reg, mem	Compare direct byte memory to direct register	2	8	4	6	8	12	4

Table 14 : Shift and rotate instructions (continued)

Mnemonic		Description	Int. ROM	Int. RAM	16-bit N-Mux	16-bit Mux	8-bit N-Mux	8-bit Mux	Bytes
ASHR	Rw, Rw	Arithmetic (sign bit) shift right direct word GPR; number of shift cycles specified by direct GPR	2	6	2	3	4	6	2
ASHR	Rw, #$data_4$	Arithmetic (sign bit) shift right direct word GPR; number of shift cycles specified by immediate data	2	6	2	3	4	6	2
ROL	Rw, Rw	Rotate left direct word GPR; number of shift cycles specified by direct GPR	2	6	2	3	4	6	2
ROL	Rw, #$data_4$	Rotate left direct word GPR; number of shift cycles specified by immediate data	2	6	2	3	4	6	2
ROR	Rw, Rw	Rotate right direct word GPR; number of shift cycles specified by direct GPR	2	6	2	3	4	6	2
ROR	Rw, #$data_4$	Rotate right direct word GPR; number of shift cycles specified by immediate data	2	6	2	3	4	6	2
SHL	Rw, Rw	Shift left direct word GPR; number of shift cycles specified by direct GPR	2	6	2	3	4	6	2
SHL	Rw, #$data_4$	Shift left direct word GPR; number of shift cycles specified by immediate data	2	6	2	3	4	6	2
SHR	Rw, Rw	Shift right direct word GPR; number of shift cycles specified by direct GPR	2	6	2	3	4	6	2
SHR	Rw, #$data_4$	Shift right direct word GPR; number of shift cycles specified by immediate data	2	6	2	3	4	6	2

Table 15 : Data movement instructions

Mnemonic		Description	Int. ROM	Int. RAM	16-bit N-Mux	16-bit Mux	8-bit N-Mux	8-bit Mux	Bytes
MOV	Rw, Rw	Move direct word GPR to direct GPR	2	6	2	3	4	6	2
MOV	Rw, #$data_4$	Move immediate word data to direct GPR	2	6	2	3	4	6	2
MOV	reg, #$data_{16}$	Move immediate word data to direct register	2	8	4	6	8	12	4
MOV	Rw, [Rw]	Move indirect word memory to direct GPR	2	6	2	3	4	6	2
MOV	Rw, [Rw+]	Move indirect word memory to direct GPR and post-increment source pointer by 2	2	6	2	3	4	6	2
MOV	[Rw], Rw	Move direct word GPR to indirect memory	2	6	2	3	4	6	2
MOV	[-Rw], Rw	Pre-decrement destination pointer by 2 and move direct word GPR to indirect memory	2	6	2	3	4	6	2
MOV	[Rw], [Rw]	Move indirect word memory to indirect memory	2	6	2	3	4	6	2
MOV	[Rw+], [Rw]	Move indirect word memory to indirect memory & post-increment destination pointer by 2	2	6	2	3	4	6	2
MOV	[Rw], [Rw+]	Move indirect word memory to indirect memory & post-increment source pointer by 2	2	6	2	3	4	6	2

Table 15 : Data movement instructions (continued)

Mnemonic		Description	Int. ROM	Int. RAM	16-bit N-Mux	16-bit Mux	8-bit N-Mux	8-bit Mux	Bytes
MOV	Rw, [Rw+ #$data_{16}$]	Move indirect word memory by base plus constant to direct GPR	4	10	6	8	10	14	4
MOV	[Rw+ #$data_{16}$], Rw	Move direct word GPR to indirect memory by base plus constant	2	8	4	6	8	12	4
MOV	[Rw], mem	Move direct word memory to indirect memory	2	8	4	6	8	12	4
MOV	mem, [Rw]	Move indirect word memory to direct memory	2	8	4	6	8	12	4
MOV	reg, mem	Move direct word memory to direct register	2	8	4	6	8	12	4
MOV	mem, reg	Move direct word register to direct memory	2	8	4	6	8	12	4
MOVB	Rb, Rb	Move direct byte GPR to direct GPR	2	6	2	3	4	6	2
MOVB	Rb, #$data_4$	Move immediate byte data to direct GPR	2	6	2	3	4	6	2
MOVB	reg, #$data_{16}$	Move immediate byte data to direct register	2	8	4	6	8	12	4
MOVB	Rb, [Rw]	Move indirect byte memory to direct GPR	2	6	2	3	4	6	2
MOVB	Rb, [Rw+]	Move indirect byte memory to direct GPR and post-increment source pointer by 1	2	6	2	3	4	6	2
MOVB	[Rw], Rb	Move direct byte GPR to indirect memory	2	6	2	3	4	6	2
MOVB	[-Rw], Rb	Pre-decrement destination pointer by 1 and move direct byte GPR to indirect memory	2	6	2	3	4	6	2
MOVB	[Rw], [Rw]	Move indirect byte memory to indirect memory	2	6	2	3	4	6	2
MOVB	[Rw+], [Rw]	Move indirect byte memory to indirect memory and post-increment destination pointer by 1	2	6	2	3	4	6	2
MOVB	[Rw], [Rw+]	Move indirect byte memory to indirect memory and post-increment source pointer by 1	2	6	2	3	4	6	2
MOVB Rb, [Rw+ #$data_{16}$]		Move indirect byte memory by base plus constant to direct GPR	4	10	6	8	10	14	4
MOVB [Rw+ #$data_{16}$], Rb		Move direct byte GPR to indirect memory by base plus constant	2	8	4	6	8	12	4
MOVB	[Rw], mem	Move direct byte memory to indirect memory	2	8	4	6	8	12	4
MOVB	mem, [Rw]	Move indirect byte memory to direct memory	2	8	4	6	8	12	4
MOVB	reg, mem	Move direct byte memory to direct register	2	8	4	6	8	12	4
MOVB	mem, reg	Move direct byte register to direct memory	2	8	4	6	8	12	4
MOVBS	Rw, Rb	Move direct byte GPR with sign extension to direct word GPR	2	6	2	3	4	6	2
MOVBS	reg, mem	Move direct byte memory with sign extension to direct word register	2	8	4	6	8	12	4
MOVBS	mem, reg	Move direct byte register with sign extension to direct word memory	2	8	4	6	8	12	4
MOVBZ	Rw, Rb	Move direct byte GPR with zero extension to direct word GPR	2	6	2	3	4	6	2
MOVBZ	reg, mem	Move direct byte memory with zero extension to direct word register	2	8	4	6	8	12	4
MOVBZ	mem, reg	Move direct byte register with zero extension to direct word memory	2	8	4	6	8	12	4

Table 16 : Jump and Call Instructions (continued)

Mnemonic		Description	Int. ROM	Int. RAM	16-bit N-Mux	16-bit Mux	8-bit N-Mux	8-bit Mux	Bytes
CALLA	cc, caddr	Call absolute subroutine if condition is met	4/2	10/8	6/4	8/6	10/8	14/12	4
CALLI	cc, [Rw]	Call indirect subroutine if condition is met	4/2	8/6	4/2	5/3	6/4	8/6	2
CALLR	rel	Call relative subroutine	4	8	4	5	6	8	2
CALLS	seg, caddr	Call absolute subroutine in any code segment	4	10	6	8	10	14	4
JB	bitaddr, rel	Jump relative if direct bit is set	4	10	6	8	10	14	4
JBC	bitaddr, rel	Jump relative and clear bit if direct bit is set	4	10	6	8	10	14	4
JMPA	cc, caddr	Jump absolute if condition is met	4/2	10/8	6/4	8/6	10/8	14/12	4
JMPI	cc, [Rw]	Jump indirect if condition is met	4/2	8/6	4/2	5/3	6/4	8/6	2
JMPR	cc, rel	Jump relative if condition is met	4/2	8/6	4/2	5/3	6/4	8/6	2
JMPS	seg, caddr	Jump absolute to a code segment	4	10	6	8	10	14	4
JNB	bitaddr, rel	Jump relative if direct bit is not set	4	10	6	8	10	14	4
JNBS	bitaddr, rel	Jump relative and set bit if direct bit is not set	4	10	6	8	10	14	4
PCALL	reg, caddr	Push direct word register onto system stack and call absolute subroutine	4	10	6	8	10	14	4
TRAP	#trap7	Call interrupt service routine via immediate trap number	4	8	4	5	6	8	2

Table 17 : System Stack Instructions

Mnemonic		Description	Int. ROM	Int. RAM	16-bit	16-bit	8-bit	8-bit	Bytes
POP	reg	Pop direct word register from system stack	2	6	2	3	4	6	2
PUSH	reg	Push direct word register onto system stack	2	6	2	3	4	6	2
SCXT	reg, #data$_{16}$	Push direct word register onto system stack and update register with immediate data	2	8	4	6	8	12	4
SCXT	reg, mem	Push direct word register onto system stack and update register with direct memory	2	8	4	6	8	12	4

Table 18 : Return Instructions

Mnemonic		Description	Int. ROM	Int. RAM	16-bit	16-bit	8-bit	8-bit	Bytes
RET		Return from intra-segment subroutine	4	8	4	5	6	8	2
RETI		Return from interrupt service subroutine	4	8	4	5	6	8	2
RETP	reg	Return from intra-segment subroutine and pop direct word register from system stack	4	8	4	5	6	8	2
RETS		Return from inter-segment subroutine	4	8	4	5	6	8	2

2.5 - Instruction set ordered by opcodes

The following pages list the instruction set ordered by their hexadecimal opcodes. This is used to identify specific instructions when reading executable code, i.e. during the debugging phase.

Notes for Opcode Lists

1. Some instructions are encoded by means of additional bits in the operand field of the instruction

```
x0h - x7h:Rw, #data3 or Rb, #data3
x8h - xBh:Rw, [Rw]  or Rb, [Rw]
xCh - xFh Rw, [Rw+] or Rb, [Rw+]
```

For these instructions only the lowest four GPRs, R0 to R3, can be used as indirect address pointers.

2. Some instructions are encoded by means of additional bits in the operand field of the instruction.

```
00xx.xxxx: EXTS   or   ATOMIC
01xx.xxxx: EXTP
00xx.xxxx: EXTS   or   ATOMIC
10xx.xxxx: EXTSR  or   EXTR
11xx.xxxx: EXTPR
```

Notes on the JMPR instructions

The condition code to be tested for the JMPR instructions is specified by the opcode. Two mnemonic representation alternatives exist for some of the condition codes.

Notes on the BCLR and BSET instructions

The position of the bit to be set or to be cleared is specified by the opcode. The operand "bitaddr$_{Q.q}$" (where q=0 to 15) refers to a particular bit within a bit-addressable word.

Notes on the undefined opcodes

A hardware trap occurs when one of the undefined opcodes signified by '----' is decoded by the CPU.

Table 21 : Instruction set ordered by Hex code

Hex- code	Number of Bytes	Mnemonic	Operand
00	2	ADD	Rw$_n$, Rw$_m$
01	2	ADDB	Rb$_n$, Rb$_m$
02	4	ADD	reg, mem
03	4	ADDB	reg, mem
04	4	ADD	mem, reg
05	4	ADDB	mem, reg
06	4	ADD	reg, #data$_{16}$
07	4	ADDB	reg, #data$_{16}$
08	2	ADD	Rw$_n$, [Rw$_i$+] or Rw$_n$, [Rw$_i$] or Rw$_n$, #data$_3$
09	2	ADDB	Rb$_n$, [Rw$_i$+] or Rb$_n$, [Rw$_i$] or Rb$_n$, #data$_3$
0A	4	BFLDL	bitoff$_Q$, #mask$_8$, #data$_8$
0B	2	MUL	Rw$_n$, Rw$_m$
0C	2	ROL	Rw$_n$, Rw$_m$
0D	2	JMPR	cc_UC, rel
0E	2	BCLR	bitaddr$_{Q.0}$
0F	2	BSET	bitaddr$_{Q.0}$
10	2	ADDC	Rw$_n$, Rw$_m$
11	2	ADDCB	Rb$_n$, Rb$_m$

Table 21 : Instruction set ordered by Hex code (continued)

Hex- code	Number of Bytes	Mnemonic	Operand
12	4	ADDC	reg, mem
13	4	ADDCB	reg, mem
14	4	ADDC	mem, reg
15	4	ADDCB	mem, reg
16	4	ADDC	reg, #$data_{16}$
17	4	ADDCB	reg, #$data_{16}$
18	2	ADDC	Rw_n, [Rw_i+] or Rw_n, [Rw_i] or Rw_n, #$data_3$
19	2	ADDCB	Rb_n, [Rw_i+] or Rb_n, [Rw_i] or Rb_n, #$data_3$
1A	4	BFLDH	$bitoff_Q$, #$mask_8$, #$data_8$
1B	2	MULU	Rw_n, Rw_m
1C	2	ROL	Rw_n, #$data_4$
1D	2	JMPR	cc_NET, rel
1E	2	BCLR	$bitaddr_{Q.1}$
1F	2	BSET	$bitaddr_{Q.1}$
20	2	SUB	Rw_n, Rw_m
21	2	SUBB	Rb_n, Rb_m
22	4	SUB	reg, mem
23	4	SUBB	reg, mem
24	4	SUB	mem, reg
25	4	SUBB	mem, reg
26	4	SUB	reg, #$data_{16}$
27	4	SUBB	reg, #$data_{16}$
28	2	SUB	Rw_n, [Rw_i+] or Rw_n, [Rw_i] or Rw_n, #$data_3$
29	2	SUBB	Rb_n, [Rw_i+] or Rb_n, [Rw_i] or Rb_n, #$data_3$
2A	4	BCMP	$bitaddr_{Z.z}$, $bitaddr_{Q.q}$
2B	2	PRIOR	Rw_n, Rw_m
2C	2	ROR	Rw_n, Rw_m
2D	2	JMPR	cc_EQ, rel or cc_Z, rel
2E	2	BCLR	$bitaddr_{Q.2}$
2F	2	BSET	$bitaddr_{Q.2}$
30	2	SUBC	Rw_n, Rw_m
31	2	SUBCB	Rb_n, Rb_m
32	4	SUBC	reg, mem
33	4	SUBCB	reg, mem

Table 21 : Instruction set ordered by Hex code (continued)

Hex- code	Number of Bytes	Mnemonic	Operand
34	4	SUBC	mem, reg
35	4	SUBCB	mem, reg
36	4	SUBC	reg, #$data_{16}$
37	4	SUBCB	reg, #$data_{16}$
38	2	SUBC	Rw_n, [Rw_i+] or Rw_n, [Rw_i] or Rw_n, #$data_3$
39	2	SUBCB	Rb_n, [Rw_i+] or Rb_n, [Rw_i] or Rb_n, #$data_3$
3A	4	BMOVN	$bitaddr_{Z.z}$, $bitaddr_{Q.q}$
3B	-	-	-
3C	2	ROR	Rw_n, #$data_4$
3D	2	JMPR	cc_NE, rel or cc_NZ, rel
3E	2	BCLR	$bitaddr_{Q.3}$
3F	2	BSET	$bitaddr_{Q.3}$
40	2	CMP	Rw_n, Rw_m
41	2	CMPB	Rb_n, Rb_m
42	4	CMP	reg, mem
43	4	CMPB	reg, mem
44	-	-	-
45	-	-	-
46	4	CMP	reg, #$data_{16}$
47	4	CMPB	reg, #$data_{16}$
48	2	CMP	Rw_n, [Rw_i+] or Rw_n, [Rw_i] or Rw_n, #$data_3$
49	2	CMPB	Rb_n, [Rw_i+] or Rb_n, [Rw_i] or Rb_n, #$data_3$
4A	4	BMOV	$bitaddr_{Z.z}$, $bitaddr_{Q.q}$
4B	2	DIV	Rw_n
4C	2	SHL	Rw_n, Rw_m
4D	2	JMPR	cc_V, rel
4E	2	BCLR	$bitaddr_{Q.4}$
4F	2	BSET	$bitaddr_{Q.4}$
50	2	XOR	Rw_n, Rw_m
51	2	XORB	Rb_n, Rb_m
52	4	XOR	reg, mem
53	4	XORB	reg, mem
54	4	XOR	mem, reg
55	4	XORB	mem, reg

Table 21 : Instruction set ordered by Hex code (continued)

Hex- code	Number of Bytes	Mnemonic	Operand
78	2	OR	Rw_n, [Rw_i+] or Rw_n, [Rw_i] or Rw_n, #$data_3$
79	2	ORB	Rb_n, [Rw_i+] or Rb_n, [Rw_i] or Rb_n, #$data_3$
7A	4	BXOR	$bitaddr_{Z.z}$, $bitaddr_{Q.q}$
7B	2	DIVLU	Rw_n
7C	2	SHR	Rw_n, #$data_4$
7D	2	JMPR	cc_NN, rel
7E	2	BCLR	$bitaddr_{Q.7}$
7F	2	BSET	$bitaddr_{Q.7}$
80	2	CMPI1	Rw_n, #$data_4$
81	2	NEG	Rw_n
82	4	CMPI1	Rw_n, mem
83	4	CoXXX^1	Rw_n, [Rw_m⊗]
84	4	MOV	[Rw_n], mem
85	-	-	-
86	4	CMPI1	Rw_n, #$data_{16}$
87	4	IDLE	
88	2	MOV	[-Rw_m], Rw_n
89	2	MOVB	[-Rw_m], Rb_n
8A	4	JB	$bitaddr_{Q.q}$, rel
8B	-	-	-
8C	-	-	-
8D	2	JMPR	cc_C, rel or cc_ULT, rel
8E	2	BCLR	$bitaddr_{Q.8}$
8F	2	BSET	$bitaddr_{Q.8}$
90	2	CMPI2	Rw_n, #$data_4$
91	2	CPL	Rw_n
92	4	CMPI2	Rw_n, mem
93	4	CoXXX^1	[IDXi⊗], [Rw_n⊗]
94	4	MOV	mem, [Rw_n]
95	-	-	-
96	4	CMPI2	Rw_n, #$data_{16}$
97	4	PWRDN	
98	2	MOV	Rw_n, [Rw_m+]
99	2	MOVB	Rb_n, [Rw_m+]

Table 21 : Instruction set ordered by Hex code (continued)

Hex- code	Number of Bytes	Mnemonic	Operand
9A	4	JNB	bitaddr$_{Q.q}$, rel
9B	2	TRAP	#trap7
9C	2	JMPI	cc, [Rw$_n$]
9D	2	JMPR	cc_NC, rel or cc_UGE, rel
9E	2	BCLR	bitaddr$_{Q.9}$
9F	2	BSET	bitaddr$_{Q.9}$
A0	2	CMPD1	Rw$_n$, #data$_4$
A1	2	NEGB	Rb$_n$
A2	4	CMPD1	Rw$_n$, mem
A3	4	CoXXX[1]	Rw$_n$, Rw$_m$
A4	4	MOVB	[Rw$_n$], mem
A5	4	DISWDT	
A6	4	CMPD1	Rw$_n$, #data$_{16}$
A7	4	SRVWDT	
A8	2	MOV	Rw$_n$, [Rw$_m$]
A9	2	MOVB	Rb$_n$, [Rw$_m$]
AA	4	JBC	bitaddr$_{Q.q}$, rel
AB	2	CALLI	cc, [Rw$_n$]
AC	2	ASHR	Rw$_n$, Rw$_m$
AD	2	JMPR	cc_SGT, rel
AE	2	BCLR	bitaddr$_{Q.10}$
AF	2	BSET	bitaddr$_{Q.10}$
B0	2	CMPD2	Rw$_n$, #data$_4$
B1	2	CPLB	Rb$_n$
B2	4	CMPD2	Rw$_n$, mem
B3	4	CoSTORE[1]	[Rw$_n$⊗], CoReg
B4	4	MOVB	mem, [Rw$_n$]
B5	4	EINIT	
B6	4	CMPD2	Rw$_n$, #data$_{16}$
B7	4	SRST	
B8	2	MOV	[Rw$_m$], Rw$_n$
B9	2	MOVB	[Rw$_m$], Rb$_n$
BA	4	JNBS	bitaddr$_{Q.q}$, rel
BB	2	CALLR	rel

Table 21 : Instruction set ordered by Hex code (continued)

Hex- code	Number of Bytes	Mnemonic	Operand
BC	2	ASHR	Rw_n, $\#data_4$
BD	2	JMPR	cc_SLE, rel
BE	2	BCLR	$bitaddr_{Q.11}$
BF	2	BSET	$bitaddr_{Q.11}$
C0	2	MOVBZ	Rb_n, Rb_m
C1	-	-	-
C2	4	MOVBZ	reg, mem
C3	4	CoSTORE[1]	Rw_n, CoReg
C4	4	MOV	$[Rw_m+\#data_{16}]$, Rw_n
C5	4	MOVBZ	mem, reg
C6	4	SCXT	reg, $\#data_{16}$
C7	-	-	-
C8	2	MOV	$[Rw_n]$, $[Rw_m]$
C9	2	MOVB	$[Rw_n]$, $[Rw_m]$
CA	4	CALLA	cc, caddr
CB	2	RET	
CC	2	NOP	
CD	2	JMPR	cc_SLT, rel
CE	2	BCLR	$bitaddr_{Q.12}$
CF	2	BSET	$bitaddr_{Q.12}$
D0	2	MOVBS	Rb_n, Rb_m
D1	2	ATOMIC/EXTR	$\#data_2$
D2	4	MOVBS	reg, mem
D3	4	CoMOV[1]	$[IDXi\otimes]$, $[Rw_n\otimes]$
D4	4	MOV	Rw_n, $[Rw_m+\#data_{16}]$
D5	4	MOVBS	mem, reg
D6	4	SCXT	reg, mem
D7	4	EXTP(R)/EXTS(R)	#pag, $\#data_2$
D8	2	MOV	$[Rw_n+]$, $[Rw_m]$
D9	2	MOVB	$[Rw_n+]$, $[Rw_m]$
DA	4	CALLS	seg, caddr
DB	2	RETS	
DC	2	EXTP(R)/EXTS(R)	Rw_m, $\#data_2$
DD	2	JMPR	cc_SGE, rel

Table 21 : Instruction set ordered by Hex code (continued)

Hex- code	Number of Bytes	Mnemonic	Operand
DE	2	BCLR	$\text{bitaddr}_{Q.13}$
DF	2	BSET	$\text{bitaddr}_{Q.13}$
E0	2	MOV	Rw_n, $\#data_4$
E1	2	MOVB	Rb_n, $\#data_4$
E2	4	PCALL	reg, caddr
E3	-	-	-
E4	4	MOVB	$[Rw_m+\#data_{16}]$, Rb_n
E5	-	-	-
E6	4	MOV	reg, $\#data_{16}$
E7	4	MOVB	reg, $\#data_{16}$
E8	2	MOV	$[Rw_n]$, $[Rw_m+]$
E9	2	MOVB	$[Rw_n]$, $[Rw_m+]$
EA	4	JMPA	cc, caddr
EB	2	RETP	reg
EC	2	PUSH	reg
ED	2	JMPR	cc_UGT, rel
EE	2	BCLR	$\text{bitaddr}_{Q.14}$
EF	2	BSET	$\text{bitaddr}_{Q.14}$
F0	2	MOV	Rw_n, Rw_m
F1	2	MOVB	Rb_n, Rb_m
F2	4	MOV	reg, mem
F3	4	MOVB	reg, mem
F4	4	MOVB	Rb_n, $[Rw_m+\#data_{16}]$
F5	-	-	-
F6	4	MOV	mem, reg
F7	4	MOVB	mem, reg
F8	-	-	-
F9	-	-	-
FA	4	JMPS	seg, caddr
FB	2	RETI	
FC	2	POP	reg
FD	2	JMPR	cc_ULE, rel
FE	2	BCLR	$\text{bitaddr}_{Q.15}$
FF	2	BSET	$\text{bitaddr}_{Q.15}$

Note 1. This instruction only applies to products including the MAC.

2.6 - Instruction conventions

This section details the conventions used in the individual instruction descriptions. Each individual instruction description is described in a standard format in separate sections under the following headings:

2.6.1 - Instruction name

Specifies the mnemonic opcode of the instruction.

2.6.2 - Syntax

Specifies the mnemonic opcode and the required formal operands of the instruction. Instructions can have either none, one, two or three operands which are separated from each other by commas: MNEMONIC {op1 {,op2 {,op3 } } }.

The operand syntax depends on the addressing mode. All of the available addressing modes are summarized at the end of each single instruction description.

2.6.3 - Operation

The following symbols are used to represent data movement, arithmetic or logical operators (see Table 22).

Missing or existing parentheses signifies that the operand specifies an immediate constant value, an address, or a pointer to an address as follows:

opX	Specifies the immediate constant value of opX.
(opX)	Specifies the contents of opX.
(opX_n)	Specifies the contents of bit n of opX.
((opX))	Specifies the contents of the contents of opX (i.e. opX is used as pointer to the actual operand).

Table 22 : Instruction operation symbols

Diadic operations				operator (opY)
	(opx) <-- (opy)	(opY)	is	MOVED into (opX)
	(opx) + (opy)	(opX)	is	ADDED to (opY)
	(opx) - (opy)	(opY)	is	SUBTRACTED from (opX)
	(opx) * (opy)	(opX)	is	MULTIPLIED by (opY)
	(opx) / (opy)	(opX)	is	DIVIDED by (opY)
	(opx) ^ (opy)	(opX)	is	logically ANDed with (opY)
	(opx) v (opy)	(opX)	is	logically ORed with (opY)
	(opx) ⊕ (opy)	(opX)	is	logically EXCLUSIVELY ORed with (opY)
	(opx) <--> (opy)	(opX)	is	COMPARED against (opY)
	(opx) mod (opy)	(opX)	is	divided MODULO (opY)
Monadic operations				operator (opX)
	(opx) ¬	(opX)	is	logically COMPLEMENTED

The following abbreviations are used to describe operands:

Table 23 : Operand abbreviations

Abbreviation	Description
CP	Context Pointer register.
CSP	Code Segment Pointer register.
IP	Instruction Pointer.
MD	Multiply/Divide register (32 bits wide, consists of MDH and MDL).
MDL, MDH	Multiply/Divide Low and High registers (each 16 bit wide).
PSW	Program Status Word register.
SP	System Stack Pointer register.
SYSCON	System Configuration register.
C	Carry flag in the PSW register.
V	Overflow flag in the PSW register.
SGTDIS	Segmentation Disable bit in the SYSCON register.
count	Temporary variable for an intermediate storage of the number of shift or rotate cycles which remain to complete the shift or rotate operation.
tmp	Temporary variable for an intermediate result.
0, 1, 2,...	Constant values due to the data format of the specified operation.

2.6.4 - Data types

Specifies the particular data type according to the instruction. Basically, the following data types are used: BIT, BYTE, WORD, DOUBLEWORD

Except for those instructions which extend byte data to word data, all instructions have only one particular data type.

Note that the data types mentioned here do not take into account accesses to indirect address pointers or to the system stack which are always performed with word data. Moreover, no data type is specified for System Control Instructions and for those of the branch instructions which do not access any explicitly addressed data.

2.6.5 - Description

Describes the operation of the instruction.

2.6.6 - Condition code

The following table summarizes the 16 possible condition codes that can be used within Call and Branch instructions and shows the mnemonic abbreviations, the test executed for a specific condition and the 4-bit condition code number.

Table 24 : Condition codes

Condition Code Mnemonic cc	**Test**	**Description**	**Condition Code Number c**
cc_UC	1 = 1	Unconditional	0h
cc_Z	Z = 1	Zero	2h
cc_NZ	Z = 0	Not zero	3h
cc_V	V = 1	Overflow	4h
cc_NV	V = 0	No overflow	5h
cc_N	N = 1	Negative	6h
cc_NN	N = 0	Not negative	7h
cc_C	C = 1	Carry	8h
cc_NC	C = 0	No carry	9h
cc_EQ	Z = 1	Equal	2h
cc_NE	Z = 0	Not equal	3h
cc_ULT	C = 1	Unsigned less than	8h
cc_ULE	(Z v C) = 1	Unsigned less than or equal	Fh
cc_UGE	C = 0	Unsigned greater than or equal	9h
cc_UGT	(Z v C) = 0	Unsigned greater than	Eh
cc_SLT	$(N \oplus V) = 1$	Signed less than	Ch
cc_SLE	$(Z \vee (N \oplus V)) = 1$	Signed less than or equal	Bh
cc_SGE	$(N \oplus V) = 0$	Signed greater than or equal	Dh
cc_SGT	$(Z \vee (N \oplus V)) = 0$	Signed greater than	Ah
cc_NET	(Z v E) = 0	Not equal AND not end of table	1h

2.6.7 - Flags

This section shows the state of the N, C, V, Z and E flags in the PSW register. The resulting state of the flags is represented by the following symbols (see Table 25).

If the PSW register is specified as the destination operand of an instruction, the flags can not be interpreted as described.

This is because the PSW register is modified according to the data format of the instruction:

– For word operations, the PSW register is overwritten with the word result.

– For byte operations, the non-addressed byte is cleared and the addressed byte is overwritten.

– For bit or bit-field operations on the PSW register, only the specified bits are modified.

If the flags are not selected as destination bits, they stay unchanged i.e. they maintain the state existing after the previous instruction.

In all cases, if the PSW is the destination operand of an instruction, the PSW flags do NOT represent the flags of this instruction, in the normal way.

Table 25 : List of flags

Symbol	Description	
*	The flag is set according to the following standard rules	
	N = 1 :	Most significant bit of the result is set
	N = 0 :	Most significant bit of the result is not set
	C = 1 :	Carry occurred during operation
	C = 0 :	No Carry occurred during operation
	V = 1 :	Arithmetic Overflow occurred during operation
	V = 0 :	No Arithmetic Overflow occurred during operation
	Z = 1 :	Result equals zero
	Z = 0 :	Result does not equal zero
	E = 1 :	Source operand represents the lowest negative number, either 8000h for word data or 80h for byte data.
	E = 0 :	Source operand does not represent the lowest negative number for the specified data type
“S”	The flag is set according to non-standard rules. Individual instruction pages or the ALU status flags description.	
“-”	The flag is not affected by the operation	
“0”	The flag is cleared by the operation.	
“NOR”	The flag contains the logical NORing of the two specified bit operands.	
“AND”	The flag contains the logical ANDing of the two specified bit operands.	
“‘OR”	The flag contains the logical ORing of the two specified bit operands.	
“XOR”	The flag contains the logical XORing of the two specified bit operands.	
“B”	The flag contains the original value of the specified bit operand.	
“$\overline{B}$”	The flag contains the complemented value of the specified bit operand	

2.6.8 - Addressing modes

Specifies available combinations of addressing modes. The selected addressing mode combination is generally specified by the opcode of the corresponding instruction.

However, there are some arithmetic and logical instructions where the addressing mode combination is not specified by the (identical) opcodes but by particular bits within the operand field.

In the individual instruction description, the addressing mode is described in terms of mnemonic, format and number of bytes.

– **Mnemonic** gives an example of which operands the instruction will accept.

– **Format** specifies the format of the instruction as used in the assembler listing. *Figure 3* shows the reference between the instruction format representation of the assembler and the corresponding internal organization of the instruction format (N = nibble = 4 bits). The following symbols are used to describe the instruction formats:

Table 26 : Instruction format symbols

00_h through FF_h	**Instruction Opcodes**
0, 1	Constant Values
:....	Each of the 4 characters immediately following a colon represents a single bit
:..ii	2-bit short GPR address (Rw_i)
ss	8-bit code segment number (seg).
:..##	2-bit immediate constant ($\#data_2$)
:.###	3-bit immediate constant ($\#data_3$)
c	4-bit condition code specification (cc)
n	4-bit short GPR address (Rw_n or Rb_n)
m	4-bit short GPR address (Rw_m or Rb_m)
q	4-bit position of the source bit within the word specified by QQ
z	4-bit position of the destination bit within the word specified by ZZ
#	4-bit immediate constant ($\#data_4$)
QQ	8-bit word address of the source bit (bitoff)
rr	8-bit relative target address word offset (rel)
RR	8-bit word address reg
ZZ	8-bit word address of the destination bit (bitoff)
##	8-bit immediate constant ($\#data_8$)
@@	8-bit immediate constant ($\#mask_8$)
pp 0:00pp	10-bit page address (#pag10)
MM MM	16-bit address (mem or caddr; low byte, high byte)
## ##	16-bit immediate constant ($\#data_{16}$; low byte, high byte)

Number of bytes Specifies the size of an instruction in bytes. All ST10 instructions are either 2 or 4 bytes. Instructions are classified as either single word or double word instructions (see Figure 3).

2.7 - ATOMIC and EXTended instructions

ATOMIC, EXTR, EXTP, EXTS, EXTPR, EXTSR instructions disable standard and PEC interrupts and class A traps during a sequence of the following 1...4 instructions. The length of the sequence is determined by an operand (op1 or op2, depending on the instruction). The EXTended instructions also change the addressing mechanism during this sequence (see detailed instruction description).

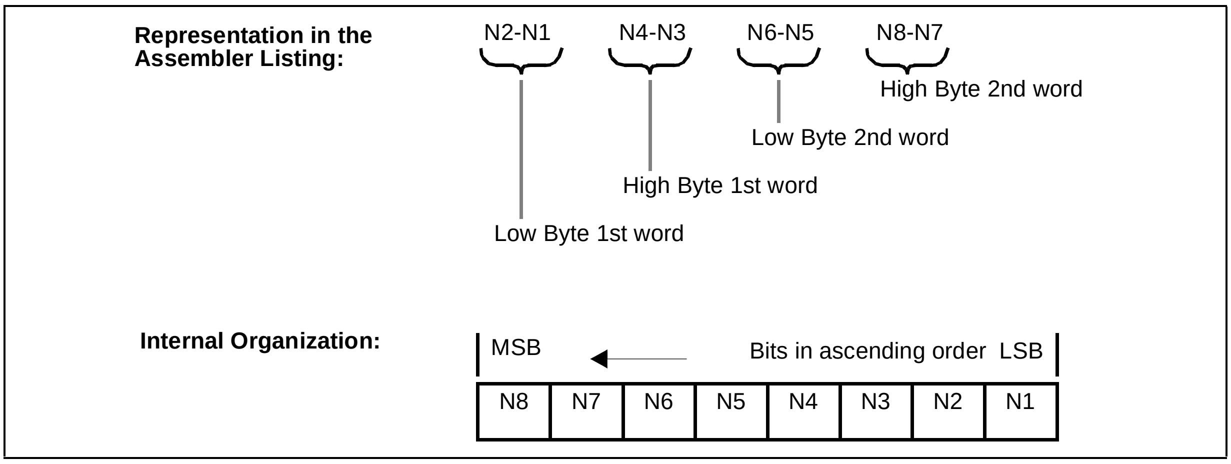
The ATOMIC and EXTended instructions become active immediately, so no additional NOPs are required. All instructions requiring multiple cycles or hold states to be executed are regarded as one instruction in this sense. Any instruction type can be used with the ATOMIC and EXTended instructions.

CAUTION: When a Class B trap interrupts an ATOMIC or EXTended sequence, this sequence is terminated, the interrupt lock is removed and the standard condition is restored, before the trap routine is executed! The remaining instructions of the terminated sequence that are executed after returning from the trap routine, will run under standard conditions!

CAUTION: When using the ATOMIC and EXTended instructions with other system control or branch instructions.

CAUTION: When using nested ATOMIC and EXTended instructions. There is ONE counter to control the length of this sort of sequence, i.e. issuing an ATOMIC or EXTended instruction within a sequence will reload the counter with value of the new instruction.

Figure 3 : Instruction format representation



2.8 - Instruction descriptions

This section contains a detailed description of each instruction, listed in alphabetical order.

ADD	**Integer Addition**	
Syntax	ADD	op1, op2
Operation	(op1)	<-- (op1) + (op2)
Data Types	WORD	

Description

Performs a 2's complement binary addition of the source operand specified by op2 and the destination operand specified by op1. The sum is then stored in op1.

Flags

E	Z	V	C	N
*	*	*	*	*

E	Set if the value of op2 represents the lowest possible negative number. Cleared otherwise. Used to signal the end of a table.
Z	Set if result equals zero. Cleared otherwise.
V	Set if an arithmetic overflow occurred, i.e. the result cannot be represented in the specified data type. Cleared otherwise.
C	Set if a carry is generated from the most significant bit of the specified data type. Cleared otherwise.
N	Set if the most significant bit of the result is set. Cleared otherwise.

Addressing Modes

Mnemonic		Format	Bytes
ADD	Rw_n, Rw_m	00 nm	2
ADD	Rw_n, [Rw_i]	08 n:10ii	2
ADD	Rw_n, [Rw_i+]	08 n:11ii	2
ADD	Rw_n, #$data_3$	08 n:0###	2
ADD	reg, #$data_{16}$	06 RR ## ##	4
ADD	reg, mem	02 RR MM MM	4
ADD	mem, reg	04 RR MM MM	4

ADDB	**Integer Addition**	
Syntax	ADDB	op1, op2
Operation	(op1)	<-- (op1) + (op2)
Data Types	BYTE	

Description

Performs a 2's complement binary addition of the source operand specified by op2 and the destination operand specified by op1. The sum is then stored in op1.

Flags

E	Z	V	C	N
*	*	*	*	*

E	Set if the value of op2 represents the lowest possible negative number. Cleared otherwise. Used to signal the end of a table.
Z	Set if result equals zero. Cleared otherwise.
V	Set if an arithmetic overflow occurred, i.e. the result cannot be represented in the specified data type. Cleared otherwise.
C	Set if a carry is generated from the most significant bit of the specified data type. Cleared otherwise.
N	Set if the most significant bit of the result is set. Cleared otherwise.

Addressing Modes

Mnemonic		Format	Bytes
ADDB	Rb_n, Rb_m	01 nm	2
ADDB	Rb_n, $[Rw_i]$	09 n:10ii	2
ADDB	Rb_n, $[Rw_i+]$	09 n:11ii	2
ADDB	Rb_n, #$data_3$	09 n:0###	2
ADDB	reg, #$data_{16}$	07 RR ## ##	4
ADDB	reg, mem	03 RR MM MM	4
ADDB	mem, reg	05 RR MM MM	4

ADDC	**Integer Addition with Carry**	
Syntax	ADDC	op1, op2
Operation	(op1)	<-- (op1) + (op2) + (C)
Data Types	WORD	

Description

Performs a 2's complement binary addition of the source operand specified by op2, the destination operand specified by op1 and the previously generated carry bit. The sum is then stored in op1. This instruction can be used to perform multiple precision arithmetic.

Flags

E	Z	V	C	N
*	S	*	*	*

E	Set if the value of op2 represents the lowest possible negative number. Cleared otherwise. Used to signal the end of a table.
Z	Set if result equals zero and previous Z flag was set. Cleared otherwise.
V	Set if an arithmetic overflow occurred, i.e. the result cannot be represented in the specified data type. Cleared otherwise.
C	Set if a carry is generated from the most significant bit of the specified data type. Cleared otherwise.
N	Set if the most significant bit of the result is set. Cleared otherwise.

Addressing Modes

Mnemonic		Format	Bytes
ADDC	Rw_n, Rw_m	10 nm	2
ADDC	Rw_n, [Rw_i]	18 n:10ii	2
ADDC	Rw_n, [Rw_i+]	18 n:11ii	2
ADDC	Rw_n, #$data_3$	18 n:0###	2
ADDC	reg, #$data_{16}$	16 RR ## ##	4
ADDC	reg, mem	12 RR MM MM	4
ADDC	mem, reg	14 RR MM MM	4

ADDCB	**Integer Addition with Carry**	
Syntax	ADDCB	op1, op2
Operation	(op1)	<-- (op1) + (op2) + (C)
Data Types	BYTE	

Description

Performs a 2's complement binary addition of the source operand specified by op2, the destination operand specified by op1 and the previously generated carry bit. The sum is then stored in op1. This instruction can be used to perform multiple precision arithmetic.

Flags

E	Z	V	C	N
*	S	*	*	*

E	Set if the value of op2 represents the lowest possible negative number. Cleared otherwise. Used to signal the end of a table.
Z	Set if result equals zero and previous Z flag was set. Cleared otherwise.
V	Set if an arithmetic overflow occurred, i.e. the result cannot be represented in the specified data type. Cleared otherwise.
C	Set if a carry is generated from the most significant bit of the specified data type. Cleared otherwise.
N	Set if the most significant bit of the result is set. Cleared otherwise.

Addressing Modes

Mnemonic		Format	Bytes
ADDCB	Rb_n, Rb_m	11 nm	2
ADDCB	Rb_n, [Rw_i]	19 n:10ii	2
ADDCB	Rb_n, [Rw_i+]	19 n:11ii	2
ADDCB	Rb_n, #$data_3$	19 n:0###	2
ADDCB	reg, #$data_{16}$	17 RR ## ##	4
ADDCB	reg, mem	13 RR MM MM	4
ADDCB	mem, reg	15 RR MM MM	4

BMOV	**Bit to Bit Move**	
Syntax	BMOV	op1, op2
Operation	(op1)	<-- (op2)
Data Types	BIT	

Description

Moves a single bit from the source operand specified by op2 into the destination operand specified by op1. The source bit is examined and the flags are updated accordingly.

Flags

E	Z	V	C	N
0	$\overline{B}$	0	0	B

E	Always cleared.
Z	Contains the logical negation of the previous state of the source bit.
V	Always cleared.
C	Always cleared.
N	Contains the previous state of the source bit.

Addressing Modes

Mnemonic	Format	Bytes
BMOV bitaddr$_{Z.z}$, bitaddr$_{Q.q}$	4A QQ ZZ qz	4

BMOVN	**Bit to Bit Move & Negate**	
Syntax	BMOVN	op1, op2
Operation	(op1)	<-- ¬(op2)
Data Types	BIT	

Description

Moves the complement of a single bit from the source operand specified by op2 into the destination operand specified by op1. The source bit is examined and the flags are updated accordingly.

Flags

E	Z	V	C	N
0	$\overline{B}$	0	0	B

E	Always cleared.
Z	Contains the logical negation of the previous state of the source bit.
V	Always cleared.
C	Always cleared.
N	Contains the previous state of the source bit.

Addressing Modes

Mnemonic	Format	Bytes
BMOVN bitaddr$_{Z.z}$, bitaddr$_{Q.q}$	3A QQ ZZ qz	4

JMPS	**Absolute Inter-Segment Jump**	
Syntax	JMPS	op1, op2
Operation	(CSP) (IP)	<-- op1 <-- op2

Description

Branches unconditionally to the absolute address specified by op2 within the segment specified by op1.

Flags

E	Z	V	C	N
-	-	-	-	-

E	Not affected
Z	Not affected
V	Not affected
C	Not affected
N	Not affected

Addressing Modes

Mnemonic		Format	Bytes
JMPS	seg, caddr	FA ss MM MM	4

MOV	**Move Data**	
Syntax	MOV	op1, op2
Operation	(op1)	<-- (op2)
Data Types	WORD	

Description

Moves the contents of the source operand specified by op2 to the location specified by the destination operand op1. The contents of the moved data is examined, and the flags are updated accordingly.

Flags

E	Z	V	C	N
*	*	-	-	*

E	Set if the value of op2 represents the lowest possible negative number. Cleared otherwise. Used to signal the end of a table.
Z	Set if the value of the source operand op2 equals zero. Cleared otherwise.
V	Not affected.
C	Not affected.
N	Set if the most significant bit of the source operand op2 is set. Cleared otherwise.

Addressing Modes

Mnemonic		Format	Bytes
MOV	Rw_n, Rw_m	F0 nm	2
MOV	Rw_n, #$data_4$	E0 #n	2
MOV	reg, #$data_{16}$	E6 RR ## ##	4
MOV	Rw_n, [Rw_m]	A8 nm	2
MOV	Rw_n, [Rw_m+]	98 nm	2
MOV	[Rw_m], Rw_n	B8 nm	2
MOV	[-Rw_m], Rw_n	88 nm	2
MOV	[Rw_n], [Rw_m]	C8 nm	2
MOV	[Rw_n+], [Rw_m]	D8 nm	2
MOV	[Rw_n], [Rw_m+]	E8 nm	2
MOV	Rw_n, [Rw_m+#$data_{16}$]	D4 nm ## ##	4
MOV	[Rw_m+#$data_{16}$], Rw_n	C4 nm ## ##	4
MOV	[Rw_n], mem	84 0n MM MM	4
MOV	mem, [Rw_n]	94 0n MM MM	4
MOV	reg, mem	F2 RR MM MM	4
MOV	mem, reg	F6 RR MM MM	4

MOVB	**Move Data**	
Syntax	MOVB	op1, op2
Operation	(op1)	<-- (op2)
Data Types	BYTE	

Description

Moves the contents of the source operand specified by op2 to the location specified by the destination operand op1. The contents of the moved data is examined, and the flags are updated accordingly.

Flags

E	Z	V	C	N
*	*	-	-	*

E	Set if the value of op2 represents the lowest possible negative number. Cleared otherwise. Used to signal the end of a table.
Z	Set if the value of the source operand op2 equals zero. Cleared otherwise.
V	Not affected.
C	Not affected.
N	Set if the most significant bit of the source operand op2 is set. Cleared otherwise.

Addressing Modes

Mnemonic		Format	Bytes
MOVB	Rb_n, Rb_m	F1 nm	2
MOVB	Rb_n, $\#data_4$	E1 #n	2
MOVB	reg, $\#data_{16}$	E7 RR ## ##	4
MOVB	Rb_n, [Rw_m]	A9 nm	2
MOVB	Rb_n, [Rw_m+]	99 nm	2
MOVB	[Rw_m], Rb_n	B9 nm	2
MOVB	[-Rw_m], Rb_n	89 nm	2
MOVB	[Rw_n], [Rw_m]	C9 nm	2
MOVB	[Rw_n+], [Rw_m]	D9 nm	2
MOVB	[Rw_n], [Rw_m+]	E9 nm	2
MOVB	Rb_n, [Rw_m+$\#data_{16}$]	F4 nm ## ##	4
MOVB	[Rw_m+$\#data_{16}$], Rb_n	E4 nm ## ##	4
MOVB	[Rw_n], mem	A4 0n MM MM	4
MOVB	mem, [Rw_n]	B4 0n MM MM	4
MOVB	reg, mem	F3 RR MM MM	4
MOVB	mem, reg	F7 RR MM MM	4

MOVBS	**Move Byte Sign Extend**
Syntax	`MOVBS op1, op2`
Operation	see below
Data Types	WORD, BYTE

```
(low byte op1)        <-- (op2)
IF (op2_7) = 1 THEN
   (high byte op1)    <-- FF_h
ELSE
   (high byte op1)    <-- 00_h
END IF
```

Description

Moves and sign extends the contents of the source byte specified by op2 to the word location specified by the destination operand op1. The contents of the moved data is examined, and the flags are updated accordingly.

Flags

E	Z	V	C	N
0	*	-	-	*

E	Always cleared.
Z	Set if the value of the source operand op2 equals zero. Cleared otherwise.
V	Not affected.
C	Not affected.
N	Set if the most significant bit of the source operand op2 is set. Cleared otherwise.

Addressing Modes

Mnemonic		Format	Bytes
MOVBS	Rb_n, Rb_m	D0 mn	2
MOVBS	reg, mem	D2 RR MM MM	4
MOVBS	mem, reg	D5 RR MM MM	4

MOVBZ	**Move Byte Zero Extend**	
Syntax	MOVBZ	op1, op2
Operation	(low byte op1) (high byte op1)	<-- (op2) <-- 00_h
Data Types	WORD, BYTE	

Description

Moves and zero extends the contents of the source byte specified by op2 to the word location specified by the destination operand op1. The contents of the moved data is examined, and the flags are updated accordingly.

Flags

E	Z	V	C	N
0	*	-	-	0

E	Always cleared.
Z	Set if the value of the source operand op2 equals zero. Cleared otherwise.
V	Not affected.
C	Not affected.
N	Always cleared.

Addressing Modes

Mnemonic		Format	Bytes
MOVBZ	Rb_n, Rb_m	C0 mn	2
MOVBZ	reg, mem	C2 RR MM MM	4
MOVBZ	mem, reg	C5 RR MM MM	4

SUB	**Integer Subtraction**	
Syntax	SUB	op1, op2
Operation	(op1)	<-- (op1) - (op2)
Data Types	WORD	

Description

Performs a 2's complement binary subtraction of the source operand specified by op2 from the destination operand specified by op1. The result is then stored in op1.

Flags

E	Z	V	C	N
*	*	*	S	*

E	Set if the value of op2 represents the lowest possible negative number. Cleared otherwise. Used to signal the end of a table.
Z	Set if result equals zero. Cleared otherwise.
V	Set if an arithmetic underflow occurred, i.e. the result cannot be represented in the specified data type. Cleared otherwise.
C	Set if a borrow is generated. Cleared otherwise.
N	Set if the most significant bit of the result is set. Cleared otherwise.

Addressing Modes

Mnemonic		Format	Bytes
SUB	Rw_n, Rw_m	20 nm	2
SUB	Rw_n, [Rw_i]	28 n:10ii	2
SUB	Rw_n, [Rw_i+]	28 n:11ii	2
SUB	Rw_n, #$data_3$	28 n:0###	2
SUB	reg, #$data_{16}$	26 RR ## ##	4
SUB	reg, mem	22 RR MM MM	4
SUB	mem, reg	24 RR MM MM	4

SUBB	**Integer Subtraction**	
Syntax	SUBB	op1, op2
Operation	(op1)	<-- (op1) - (op2)
Data Types	BYTE	

Description

Performs a 2's complement binary subtraction of the source operand specified by op2 from the destination operand specified by op1. The result is then stored in op1.

Flags

E	Z	V	C	N
*	*	*	S	*

E	Set if the value of op2 represents the lowest possible negative number. Cleared otherwise. Used to signal the end of a table.
Z	Set if result equals zero. Cleared otherwise.
V	Set if an arithmetic underflow occurred, ie. the result cannot be represented in the specified data type. Cleared otherwise.
C	Set if a borrow is generated. Cleared otherwise.
N	Set if the most significant bit of the result is set. Cleared otherwise.

Addressing Modes

Mnemonic		Format	Bytes
SUBB	Rb_n, Rb_m	21 nm	2
SUBB	Rb_n, $[Rw_i]$	29 n:10ii	2
SUBB	Rb_n, $[Rw_i+]$	29 n:11ii	2
SUBB	Rb_n, #$data_3$	29 n:0###	2
SUBB	reg, #$data_{16}$	27 RR ## ##	4
SUBB	reg, mem	23 RR MM MM	4
SUBB	mem, reg	25 RR MM MM	4

SUBC	**Integer Subtraction with Carry**	
Syntax	SUBC	op1, op2
Operation	(op1)	<-- (op1) - (op2) - (C)
Data Types	WORD	

Description

Performs a 2's complement binary subtraction of the source operand specified by op2 and the previously generated carry bit from the destination operand specified by op1. The result is then stored in op1. This instruction can be used to perform multiple precision arithmetic.

Flags

E	Z	V	C	N
*	S	*	S	*

E	Set if the value of op2 represents the lowest possible negative number. Cleared otherwise. Used to signal the end of a table.
Z	Set if result equals zero and the previous Z flag was set. Cleared otherwise.
V	Set if an arithmetic underflow occurred, i.e. the result cannot be represented in the specified data type. Cleared otherwise.
C	Set if a borrow is generated. Cleared otherwise.
N	Set if the most significant bit of the result is set. Cleared otherwise.

Addressing Modes

Mnemonic		Format	Bytes
SUBC	Rw_n, Rw_m	30 nm	2
SUBC	Rw_n, [Rw_i]	38 n:10ii	2
SUBC	Rw_n, [Rw_i+]	38 n:11ii	2
SUBC	Rw_n, #$data_3$	38 n:0###	2
SUBC	reg, #$data_{16}$	36 RR ## ##	4
SUBC	reg, mem	32 RR MM MM	4
SUBC	mem, reg	34 RR MM MM	4

SUBCB	**Integer Subtraction with Carry**	
Syntax	SUBCB	op1, op2
Operation	(op1)	<-- (op1) - (op2) - (C)
Data Types	BYTE	

Description

Performs a 2's complement binary subtraction of the source operand specified by op2 and the previously generated carry bit from the destination operand specified by op1. The result is then stored in op1. This instruction can be used to perform multiple precision arithmetic.

Flags

E	Z	V	C	N
*	S	*	S	*

E	Set if the value of op2 represents the lowest possible negative number. Cleared otherwise. Used to signal the end of a table.
Z	Set if result equals zero and the previous Z flag was set. Cleared otherwise.
V	Set if an arithmetic underflow occurred, i.e. the result cannot be represented in the specified data type. Cleared otherwise.
C	Set if a borrow is generated. Cleared otherwise.
N	Set if the most significant bit of the result is set. Cleared otherwise.

Addressing Modes

Mnemonic		Format	Bytes
SUBCB	Rb_n, Rb_m	31 nm	2
SUBCB	Rb_n, [Rw_i]	39 n:10ii	2
SUBCB	Rb_n, [Rw_i+]	39 n:11ii	2
SUBCB	Rb_n, #$data_3$	39 n:0###	2
SUBCB	reg, #$data_{16}$	37 RR ## ##	4
SUBCB	reg, mem	33 RR MM MM	4
SUBCB	mem, reg	35 RR MM MM	4