Senior
Instruction Set Manual

November 13, 2024

Contents

1 Senior hardware description **5**
- 1.1 General . . . 5
- 1.2 Accumulators . . . 5
- 1.3 Registers . . . 8
- 1.4 Memory . . . 8
- 1.5 Coding convention . . . 9
- 1.6 Constants . . . 10
- 1.7 Operations . . . 11
  - 1.7.1 Scaling . . . 11
  - 1.7.2 Rounding . . . 11
  - 1.7.3 Saturation . . . 12
  - 1.7.4 8 bit mode . . . 12
- 1.8 Software stack . . . 12
- 1.9 FFT addressing . . . 12

2 Move-load-store instructions **15**

3 Short arithmetic instructions, 16'b **31**

4 Short logic instructions, 16'b **45**

5 Short shift instructions, 16'b **53**

6 Long arithmetic instructions, 32'b **63**

7 Iterative instructions **83**

8 Flow control instructions **85**

9 Alias instructions **91**

1 Senior hardware description

1.1 General

The **Senior** processor is a single issue DSP processor for applications including voice codec, audio decoder, bit manipulations and program flow controller for video codec. The instruction set supports running basic kernels of BDTI benchmarking.

The core includes data path, control path and address path. The data path consists of a general register file, ALU and a MAC. The addressing path consists of four circuit modules, AG0–AG3. AG0–AG1 supporting modulo, variable size post increment addressing and bit reversal addressing. The control path consists of a PC FSM, a loop controller supporting REPEAT of subroutine up to 127 instructions.

Interrupt handler and timer are directly connected to the core. The instruction decoder checks interrupt interface first before decoding an instruction. All access requests from other peripheral components to the core are controlled by the interrupt handler. Pipeline of different instructions is listed in Table 1.1. The pipeline architecture for all instructions except `conv` (convolution) is given in the simplified Fig. 1.1. The pipeline architecture for `conv` is given in the simplified Fig. 1.2.

The typical pipeline for RISC instructions is `IF(P1)→ID(P2)→OF(P3)→EX(P4)`. Write back operations are executed in `P4` right after execution during the execution cycle.

1.2 Accumulators

There are 4 accumulators (32 bits wide), `acr0` to `acr3`, used for double precision computing. There are 8 guard bits in the MAC unit giving a total of 40 bits internal resolution for the accumulators during computing.

Pipe	RISC-E1/E2	RISC memory load/store	CISC-convolution
P1	IF: instr fetch	IF: instr fetch	IF: instr fetch
P2	ID: instr decode	ID: instr decode	ID: instr decode
P3	OF: operand fetch	OF+AG: compute addr	OF+AG: compute addr
P4	EX1:execution (set flags)	MEM:read/write	MEM: read
P5	EX2:only for MAC, RWB	WB: write back (if load)	send data mem→mul
P6			MUL
P7			accumulation

Table 1.1: Pipeline specification

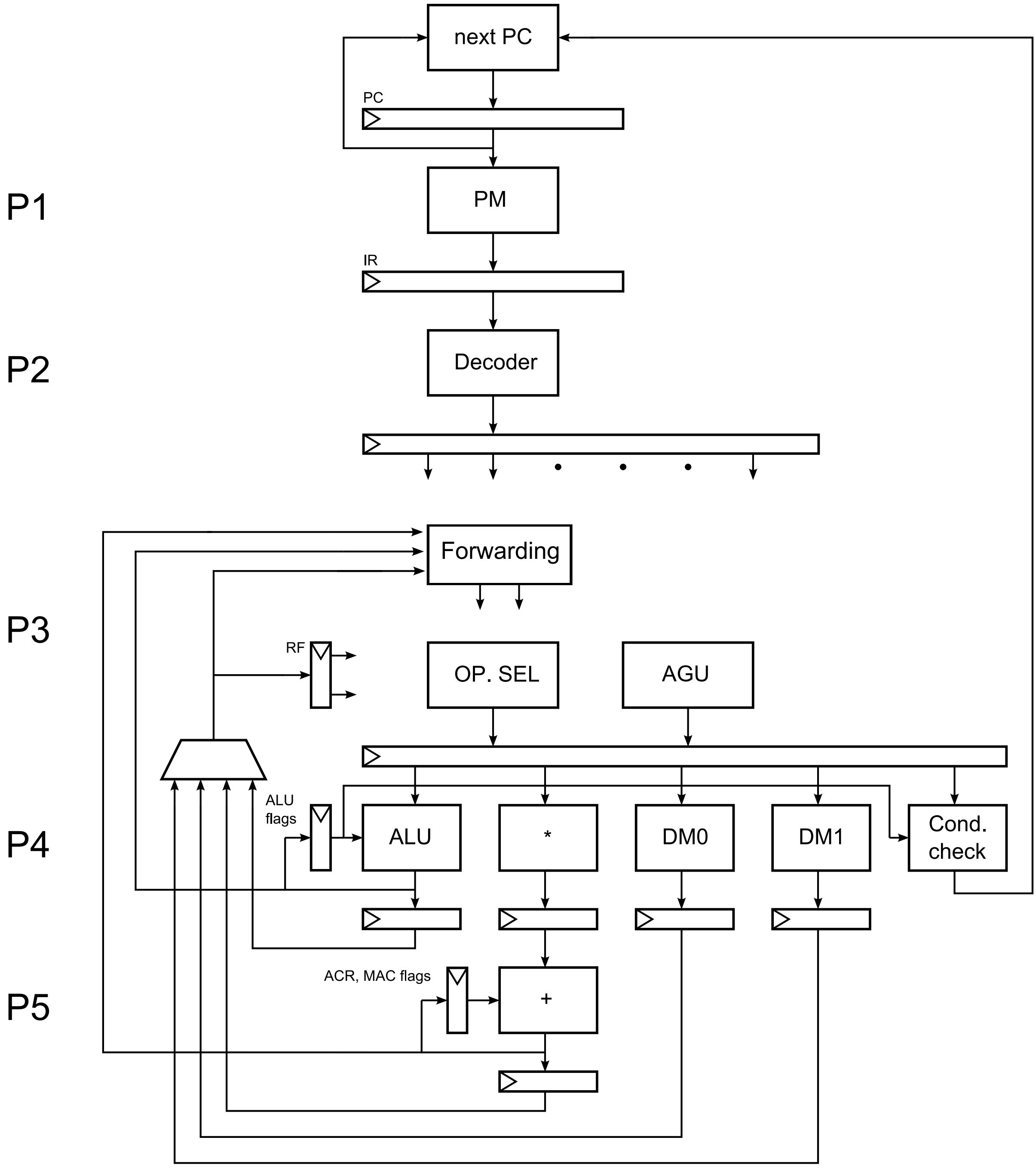


Figure 1.1: Pipeline processor architecture for all instructions but `conv`

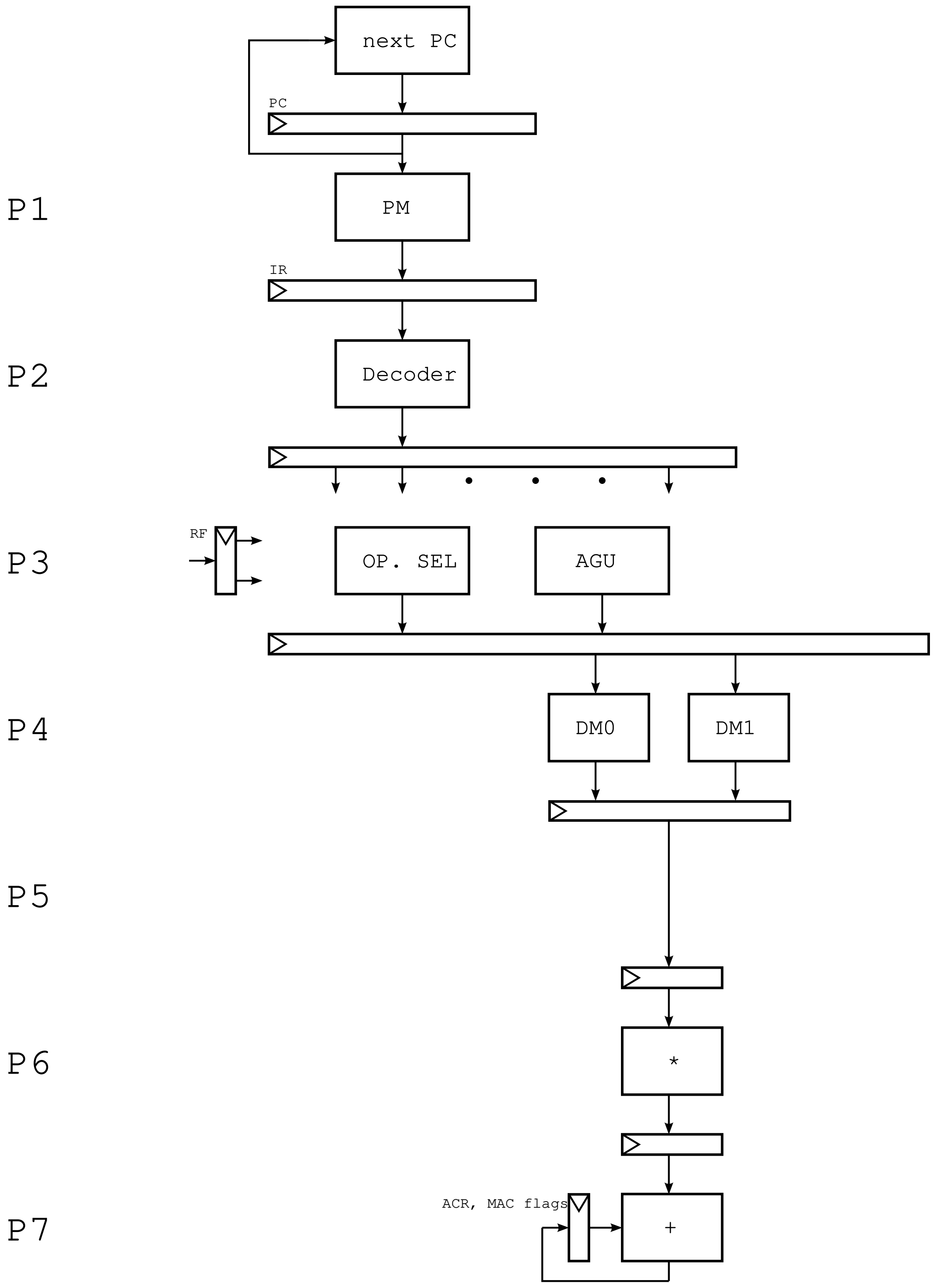


Figure 1.2: Pipeline processor architecture for `conv`

1.3 Registers

There are 32 general registers (16 bits wide), `r0` to `r31`, used as computing buffers. General registers can be addressed using 5 bits binary code. There are also 32 special registers (16 bits wide, except bitrev that is 3 bits wide), `sr0` to `sr31`, that have special functions. They are addressed using 5 bits binary code and allocated either in the control path or in the address generator AGU or in the MAC (guard bits). Defined special registers can be seen in Table 1.2.

Mnemonic	Location	Address code	Specification
ar0	AG	00000	Address register 0
ar1	AG	00001	Address register 1
ar2	AG	00010	Address register 2
ar3	AG	00011	Address register 3
sp	AG	00100	Stack pointer
bot0	AG	00101	Bottom for AR0
top0	AG	00110	Top for AR0
step0	AG	00111	Step size for AR0
bot1	AG	01000	Bottom for AR1
top1	AG	01001	Top for AR1
step1	AG	01010	Step size for AR1
bitrev	AG	01011	Number of bits to reverse-6
fftbase	AG	01100	Base address for FFT addressing
fftstage	AG	01101	Current stage of FFT addressing
intaddr	AG	01110	Start address for interrupts
fl0	CP	01111	Flags, processor status register
fl1	CP	10000	Flags, core control register
loopn	CP	10001	Number of iterations in loop
loopb	CP	10010	Loop start address
loope	CP	10011	Loop end address
intmask	CP	10100	*Reserved(interrupt mask)*
guards01	MAC	10101	Guard for ACR0 and ACR1
guards23	MAC	10110	Guard for ACR2 and ACR3

Table 1.2: Special purpose registers

Special registers (`sr0` to `sr31`) can only be accessed with move instructions. Only general registers involves in arithmetic and load-store instructions.

The `bitrev` special purpose register determines the number of bits to reverse when using bit reversed addressing mode. `bitrev`+6 bits will be reversed starting at the second least significant bit. That is the least significant bit will not be affected by the reversal.

Processor status register (`fl0`) and core control vector register (`fl1`) are in the following Table 1.3.

1.4 Memory

Three memories can be directly connected to the core, the PM (program memory) which is 32 bits wide and at most 2^{16} words and DM0 (data memory 0) and DM1 (data memory 1) which are 16 bits wide and at most 2^{15} words. The data memories are are addressed with 16 bits and the lsb of

Name	Bit assignment	Specification
AZ	fl0[0]	ALU zero flag
AN	fl0[1]	ALU sign flag
AC	fl0[2]	ALU carry/saturation flag
AV	fl0[3]	ALU ALU overflow flag
MZ	fl0[4]	MAC zero flag
MN	fl0[5]	MAC sign flag
MS	fl0[6]	MAC saturation flag
MV	fl0[7]	MAC overflow flag (sticky)
FFT	fl1[0]	Actives FFT addressing for DM0
TR	fl1[]	*Reserved(Trace mode)*
IE	fl1[]	*Reserved(Global interrupt enable)*

Table 1.3: Control and status registers

Group code	Type[31:30]	Specification
00	Move--load--store	Memory access and register move instructions
01	Arithmetic operations	16'b arithmetic/logic/shift and 32'b arithmetic instructions
10	Program flow control	Jumps, calls, REPEAT, NOP, SLEEP, SW-trap instructions
11	Accelerations	Reserved for all accelerations

Table 1.4: Instruction groups

the address is used to point out which half of the word is being addressed. 8 bits can be read or written at any address but 16 bits can only be read or written at even addresses. 8 bit reading or writing is specified with the flag `b8`.

1.5 Coding convention

Instruction are classified into four groups, as seen in Table 1.4. Flags of the DSP core expose behaviour of the most recent results from ALU and MAC. Flags are specified in Table 1.5. Most move and arithmetic instructions can be conditionally executed. Because of the limitation of the instruction word width, conditional executions are not available when carrying long constants as

Flag	Signed computing	Unsigned computing
AZ	ALU result is zero	ALU result is zero
AN	ALU result is negative	ALU result is zero
AC	ALU saturated (or carry out)	ALU carry out
AV	ALU result overflowed	No meaning
MZ	MAC result is zero	N/A, MAC does not support unsigned computing
MN	MAC result is negative	N/A, MAC does not support unsigned computing
MS	MAC saturated, sticky	N/A, MAC does not support unsigned computing
MV	MAC result overflow (>40bits), sticky	N/A, MAC does not support unsigned computing

Table 1.5: Flag specification (when flag is "1")

Mnemonic	Code	Description/specification	Flag test
	00000	Unconditionally true	none
	00001	*Unused*	
eq	00010	ALU equal/zero	AZ=1
ne	00011	ALU not equal/not zero	AZ=0
ugt	00100	ALU unsigned greater than	AC=1 and AZ=0
uge/cs	00101	ALU unsigned greater than or equal	AC=1
ule	00110	ALU unsigned less than or equal	AC=0 or AZ=1
ult/cc	00111	ALU unsigned less than	AC=0
sgt	01000	ALU signed greater than	AN=AV and AZ=0
sge	01001	ALU signed greater than or equal	AN=AV
sle	01010	ALU signed less than or equal	AZ=1 or AN≠AV
slt	01011	ALU signed less than	AN≠AV
mi	01100	ALU negative	AN=1
pl	01101	ALU positive	AN=0
vs	01110	ALU has overflowed	AV=1
vc	01111	ALU has not overflowed	AV=0
meq	10000	MAC or MUL equal	MZ=1
mne	10001	MAC or MUL not equal	MZ=0
mgt	10010	MAC or MUL greater than	MN=0 and MZ=0
mge/mpl	10011	MAC or MUL positive or zero	MN=0
mle	10100	MAC or MUL less than or equal	MN=1 or MZ=1
mlt/mmi	10101	MAC or MUL negative or less than	MN=1
mvs	10110	MAC was saturated	MS=1
mvc	10111	MAC was not saturated	MS=0

Table 1.6: Testing flags for condition

immediate data or immediate address. Some coding fields are coded for all applications through the manual. CDT is a 5-bit coding field for conditional execution and conditional jumps. All Address computing algorithms are "right-aligned" using either data type without saturation and rounding. There is no overflow check in hardware. Overflow is actually an implicit mode of the 2^{16} operation.

1.6 Constants

A binary constant is introduced by `%`, a decimal by (nothing) and a hexadecimal by `$` or `0x`. Therefore `%` and `$` are only used to denote data types. For example:

- `%0001 1111 1111 1111 = $7FFF = 0x7FFF = 32767 in case of 16 bits`
- `%0111 11111 1111 = $7FF = 0x7FF = 2047 in case of 12 bits (right aligned, integer mode)`
- `%0111 1 = $F = 0xF = 15 in case of 5 bits`

AM	Code	Addressing model	Coding	Algorithm specification
INDR	000	Reg indirect	Any R0-R31	A←Rn
INDX	001	Indexed	Any AR0-AR3 or R0-R31	A←ARn + Rm
INC	010	Post-add	Any AR0-AR3	A←ARn; ARn+=STn
DEC	011	Pre-subtract	Any AR0-AR3	ARn-=STn; A←ARn
OFS	100	Offset addressing	Any AR0-AR3, offset	A← + signed offset
MINC	101	Post add, mod addr	Any AR0-AR3	A←ARn; Arn+=STn
ABS	110	Absolute	Immediate 16'b	A←abs(16b immediate)
BRV	111	Bit reversal	Any AR0-AR3	A←BR(ARn); ARn+=STn

Table 1.7: Addressing modes

1.7 Operations

1.7.1 Scaling

Scaling operation can be performed for the instructions `move`, `postop`, `mul`, `mac`, `mdm` and `conv`. It operates on the source or destination accumulator (depending on the instruction) together with the instruction operation itself. Table 1.8 lists the optional scale factors.

Mnemonics	Code	Scale factor	Description
	000	1	No scaling
mul2	001	2	Multiply by 2
mul4	010	4	Multiply by 4
div2	011	0.5	Divide by 2
div4	100	0.25	Divide by 4
div8	101	0.125	Divide by 8
div16	110	0.0625	Divide by 16
mul65536	111	2^{16}	Multiply by 2^{16}

Table 1.8: Scale factors

1.7.2 Rounding

Rounding operation can be performed for the instructions `move`, `addl`, `sublst` and `postop`. It operates on the source or destination accumulator (depending on the instruction) together with the instruction operation itself. Table 1.9 lists the rounding operator.

Mnemonics	Code	Description
	0	No rounding
rnd	1	Rounding to nearest integer value

Table 1.9: Rounding

1.7.3 Saturation

Saturation operation can be performed for the instructions `move`, `abs`, `absl`, `negl`, `addl`, `sublst` and `postop`. It operates on the destination operand together with the instruction operation itself. Table 1.10 lists the saturation operator.

Mnemonics	Code	Description
	0	No saturation
sat	1	Saturate to within signed destination operand bitsize range

Table 1.10: Saturation

1.7.4 8 bit mode

8 bit mode can be performed for the instructions `ld0`, `ld1`, `st0` and `st1`. If the *b8* flag is left out, it will be a normal 16 bit operation. When using the *b8* flag, a store at an even address will result in putting the 8 lsb of the register into the 8 lsb of the memory. A store at an odd address will result in putting the 8 lsb of the register into the 8 msb of the memory. The other 8 bits of the line in memory is not affected. When loading with 8 bit mode, an even address will load from the 8 lsb and an odd address will load from the 8 msb. The loaded data will be put into the 8 lsb of the register and the 8 msb of the register will be cleared. 8 bit mode causes the stepsize of `ar2` and `ar3` to become 1, instead of 2 in normal mode. The stepsize of `ar0` and `ar1` is not affected by 8 bit mode.

1.8 Software stack

The software stack is located in `DM1` and grows towards lower addresses. `sp`++ in this manual will therefore in reality be handled as `sp` = `sp` − 2. The stack is implicitly used for subroutine calls and interrupts. The special purpose register `sp` is used as the address pointer for the stack and may only be used for post-incremental, pre-decremental and offset addressing using `ld1` and `st1`. The `sp` register must be set before any subroutine calls are made or any interrupts occur.

1.9 FFT addressing

The Senior processor contains special hardware for generating addresses during FFT computations. When `fl1[0]` is set, all memory accessing instructions on `DM0` uses the address from this hardware, no matter the addressing mode specified in the instruction.

If the input samples for the FFT is located in a continuous order in `DM0` starting with the real part and then the imaginary part for each sample, then the first generated address will point at the real part of the second input sample to the first butterfly. The next address points to the imaginary part. Then the real and the imaginary part of the first input is pointed out. Next the real part of first input is addressed again followed by the imaginary part and then the second input. This to store the results of the butterfly operation in an in-place FFT. The next generated address points at the real part of the second input of the next butterfly.

The special register `fftbase` is used to tell the FFT addressing hardware where the input samples are located in `DM0`. It should be set to the address of the real value of the first input to the first butterfly. This must be set in the beginning of each stage (should be the same value for each stage) as `fftbase` is updated by the hardware on each memory accessing instruction using `DM0`.

The get the addressing inside a stage correct, the special register `fftstage` is used to tell the FFT addressing hardware which stage of the FFT that is currently being computed. `fftstage` $= 1$ corresponds to the first stage that is using bit reverse addressing in DIT FFTs. Note that bit reverse addressing is not supported by the FFT addressing hardware.

2 Move-load-store instructions

Move-load-store instructions concern operations of 16 bit data transfer between general registers, special registers, accumulators, IO ports and data memories. The move instructions (**`move`** and **`set`**) can only work with registers and accumulators (**`move`** only) for data access, as the load-store instructions (**`ld0`**, **`ld1`**, **`st0`**, **`st1`**, **`dblld`**, **`dblst`**, **`in`**, **`out`**) can use various addressing modes for data memory access. Table 2.1 lists the move-load-store group of instructions.

Mnemonic	Description	Page
move	copy data to/from registers and accumulators	16
set	set constant to register	17
ld0	load register from data memory 0	18
ld1	load register from data memory 1	19
st0	store register to data memory 0	20
st1	store register to data memory 1	21
dblld	double load	22
dblst	double store	23
in	read IO port to register	24
out	write register to IO port	25
dblin	write dm0 and dm1 from IO port	26
dblout	write IO port from dm0 and dm1	27

Table 2.1: move-load-store instructions

MOVE — move

Syntax

1 **move** **r***d*,**sr***a*
2 **move** **sr***d*,**r***a*
3 **move**[.*cdt*] **r***d*,[**sat**] [**rnd**] [*sf*] **acr***A*
4 **move** **acr***D*[**.h**|**.l**],**r***a*
5 **move**[.*cdt*] **r***d*,**r***a*

$a \epsilon [0..31]$, $d \epsilon [0..31]$, $A \epsilon [0..3]$, $D \epsilon [0..3]$
sf ϵ[**mul2**|**mul4**|**div2**|**div4**|**div8**|**div16**|**mul65536**]

Operation

1 **r***d*←**sr***a*
2 **sr***d*←**r***a*
3 if *cdt* is true: **r***d*←saturation(round(scale(**acr***A*)))
4 if **h** is used: **acr***D*[31..16]←**r***a*, if **l** is used: **acr***D*[15..0]←**r***a*
5 if *cdt* is true: **r***d*←**r***a*

Description

Copy contents from source to destination.

Flags

MV	MS	MN	MZ
	U		

Comment

Syntax no.5 above is implemented as `orn[.cdt]ra,rb,rb`. Use the instruction `exch` to exchange contents between special register and register.

Example

```
move     r1,sr2
move     sr1,r2
move.eq  r22,rnd mul2 acr3
move     acr2.h,r12
move     r7,r14
```

Instruction 2.1: MOVE — move

SET — set register to constant

Syntax

1 **set** **r**d**,**K
2 **set** **sr**d**,**K
$d \epsilon$[0..31], $K \epsilon$[\$0000..\$FFFF]

Operation

1 **r**$d \leftarrow K$
2 **sr**$d \leftarrow K$

Description

Set destination register to an unsigned 16'b constant

Flags

No flags affected

Comment

Example

```
set     r21,711
set     sr12,$2C7
set     r3,%1111000111
```

Instruction 2.2: SET — set register to constant

LD0 — load register from DM0

Syntax

1	**ld0**	[**b8**]	**r**d**,(r**b**)**
2	**ld0**	[**b8**]	**r**d**,(ar**a**,r**b**)**
3	**ld0**	[**b8**]	**r**d**,(ar**c**++)**
4	**ld0**	[**b8**]	**r**d**,(--ar**c**)**
5	**ld0**	[**b8**]	**r**d**,(ar**a**,**K**)**
6	**ld0**	[**b8**]	**r**d**,(ar**e**++%)**
7	**ld0**	[**b8**]	**r**d**,(**L**)**
8	**ld0**	**r**d**,br(ar**c**)**	

$a\epsilon$[0..3], $b\epsilon$[0..31], $c\epsilon$[0..3], $d\epsilon$[0..31], $e\epsilon$[0..1], $K\epsilon$[-2048..2047], $L\epsilon$[\$0000..\$7FFF]

Operation

1. **r**d←DM0(**r**a)
2. **r**d←DM0(**ar**a+**r**b)
3. **r**d←DM0(**ar**c), if $c\epsilon$[0..1] then **ar**c←**ar**c+**step**c else **ar**c←**ar**c+1/2
4. if $c\epsilon$[0..1] then **ar**c←**ar**c-**step**c else **ar**c←**ar**c-1/2, **r**d←DM0(**ar**c)
5. **r**d←DM0(**ar**a+K)
6. **r**d←DM0(**ar**e), if **ar**e=**top**e then **ar**e←**bot**e else **ar**e←**ar**e+**step**e
7. **r**d←DM0(L)
8. **r**d←DM0(bitrev(**ar**c)), if $c\epsilon$[0..1] then **ar**c←**ar**c+**step**c else **ar**c←**ar**c+2

Description

Load a general register with data from data memory 0 using various addressing modes. If 8 bit mode (b8) is used, 8 bits are stored in the register and the 8 msb are cleared.

Flags

No flags affected

Comment

Syntax no.5 can be used without the constant K, in which case it will be assumed to be 0.
Operation no.6: observe that **ar**e will only be set to **bot**e when the condition **ar**e=**top**e is met. There is nothing preventing **ar**e from becoming larger than **top**e at the **ar**e←**ar**e+**step**e operation.
1/2 in stepsize means 1 in 8-bit mode and 2 otherwise.

Example

```
ld0      r1,(ar1,r9)
```

Instruction 2.3: LD0 — load register from DM0

LD1 — load register from DM1

Syntax

1	**ld1**	[**b8**]	**r***d*,(**r***b*)
2	**ld1**	[**b8**]	**r***d*,(**ar***a*,**r***b*)
3	**ld1**	[**b8**]	**r***d*,(**ar***c*++)
4	**ld1**	[**b8**]	**r***d*,(--**ar***c*)
5	**ld1**	[**b8**]	**r***d*,(**ar***a*,*K*)
6	**ld1**	[**b8**]	**r***d*,(**ar***e*++%)
7	**ld1**	[**b8**]	**r***d*,(*L*)
8	**ld1**		**r***d*,**br**(**ar***c*)
9	**ld1**		**r***d*,(--**sp**)
10	**ld1**		**r***d*,(**sp**,*K*)

$a\epsilon[0..3]$, $b\epsilon[0..31]$, $c\epsilon[0..3]$, $d\epsilon[0..31]$, $e\epsilon[0..1]$, $K\epsilon[-2048..2047]$, $L\epsilon[\$0000..\$7FFF]$

Operation

1 **r***d*←DM1(**r***a*)
2 **r***d*←DM1(**ar***a*+**r***b*)
3 **r***d*←DM1(**ar***c*), if $c\epsilon[0..1]$ then **ar***c*←**ar***c*+**step***c* else **ar***c*←**ar***c*+1/2
4 if $c\epsilon[0..1]$ then **ar***c*←**ar***c*-**step***c* else **ar***c*←**ar***c*-1/2, **r***d*←DM1(**ar***c*)
5 **r***d*←DM1(**ar***a*+*K*)
6 **r***d*←DM1(**ar***e*), if **ar***e*=**top***e* then **ar***e*←**bot***e* else **ar***e*←**ar***e*+**step***e*
7 **r***d*←DM1(*L*)
8 **r***d*←DM1(bitrev(**ar***c*)), if $c\epsilon[0..1]$ then **ar***c*←**ar***c*+**step***c* else **ar***c*←**ar***c*+2
9 **sp**←**sp**-1, **r***d*←DM1(**sp**)
10 **r***d*←DM1(**sp**+*K*)

Description

Load a general register with data from data memory 1 using various addressing modes. In 8 bit mode the 8 msb in the register are cleared.

Flags

No flags affected

Comment

Syntax no.5 can be used without the constant *K*, in which case it will be assumed to be 0.
Operation no.6: observe that **ar***e* will only be set to **bot***e* when the condition **ar***e*=**top***e* is met. There is nothing preventing **ar***e* from becoming larger than **top***e* at the **ar***e*←**ar***e*+**step***e* operation.
1/2 in stepsize means 1 in 8-bit mode and 2 otherwise.

Example

```
ld1     r7,(channel)     ;channel is a previously defined constant
```

ST0 — store register to DM0

Syntax

1	**st0**	[**b8**]	**(r***d***),r***b*
2	**st0**	[**b8**]	**(ar***f***,r***d***),r***b*
3	**st0**	[**b8**]	**(ar***g***++),r***b*
4	**st0**	[**b8**]	**(--ar***g***),r***b*
5	**st0**	[**b8**]	**(ar***f***,***K***),r***b*
6	**st0**	[**b8**]	**(ar***h***++%),r***b*
7	**st0**	[**b8**]	**(***L***),r***b*
8	**st0**		**br(ar***g***),r***b*

*b*ϵ[0..31], *d*ϵ[0..31], *f*ϵ[0..3], *g*ϵ[0..3], *h*ϵ[0..1], *K*ϵ[-2048..2047], *L*ϵ[$0000..$7FFF]

Operation

1 DM0(**r***d*)←**r***b*
2 DM0(**ar***f*+**r***d*)←**r***b*
3 DM0(**ar***g*)←**r***b*, if *g*ϵ[0..1] then **ar***g*←**ar***g*+**step***g* else **ar***g*←**ar***g*+1/2
4 if *g*ϵ[0..1] then **ar***g*←**ar***g*-**step***g* else **ar***g*←**ar***g*-1/2, DM0(**ar***g*)←**r***b*
5 DM0(**ar***f*+*K*)←**r***b*
6 DM0(**ar***h*)←**r***b*, if **ar***h*=**top***h* then **ar***h*←**bot***h* else **ar***h*←**ar***h*+**step***h*
7 DM0(*L*)←**r***b*
8 DM0(bitrev(**ar***g*))←**r***b*, if *c*ϵ[0..1] then **ar***g*←**ar***g*+**step***g* else **ar***g*←**ar***g*+2

Description

Store contents of a general register to DM0 (data memory 0) using various addressing modes. When 8 bit mode (b8) is used, the 8 lsb of the register are placed in the memory at given address. The other 8 bits in the same word in the memory are not affected.

Flags

No flags affected

Comment

Syntax no.5 can be used without the constant *K*, in which case it will be assumed to be 0.
Operation no.6: observe that **ar***h* will only be set to **bot***h* when the condition **ar***h*=**top***h* is met. There is nothing preventing **ar***h* from becoming larger than **top***h* at the **ar***h*←**ar***h*+**step***h* operation.
1/2 in stepsize means 1 in 8-bit mode and 2 otherwise.

Example

```
st0      (ar2++),r31
```

Instruction 2.5: ST0 — store register to DM0

ST1 — store register to DM1

Syntax

1	**st1**	[**b8**]	**(r**d**),r**b
2	**st1**	[**b8**]	**(ar**f**,r**d**),r**b
3	**st1**	[**b8**]	**(ar**g**++),r**b
4	**st1**	[**b8**]	**(--ar**g**),r**b
5	**st1**	[**b8**]	**(ar**f**,**K**),r**b
6	**st1**	[**b8**]	**(ar**h**++%),r**b
7	**st1**	[**b8**]	**(**L**),r**b
8	**st1**	**br(ar**g**),r**b	
9	**st1**	**(sp++),r**b	
10	**st1**	**(sp,**K**),r**b	

$b\epsilon$[0..31], $d\epsilon$[0..31], $f\epsilon$[0..3], $g\epsilon$[0..3], $h\epsilon$[0..1], $K\epsilon$[-2048..2047], $L\epsilon$[\$0000..\$7FFF]

Operation

1	DM1(**r**d)←**r**b
2	DM1(**ar**f+**r**d)←**r**b
3	DM1(**ar**g)←**r**b, if $g\epsilon$[0..1] then **ar**g←**ar**g+**step**g else **ar**g←**ar**g+1/2
4	if $g\epsilon$[0..1] then **ar**g←**ar**g-**step**g else **ar**g←**ar**g-1/2, DM1(**ar**g)←**r**b
5	DM1(**ar**f+K)←**r**b
6	DM1(**ar**h)←**r**b, if **ar**h=**top**h then **ar**h←**bot**h else **ar**h←**ar**h+**step**h
7	DM1(L)←**r**b
8	DM1(bitrev(**ar**g))←**r**b, if $c\epsilon$[0..1] then **ar**g←**ar**g+**step**g else **ar**g←**ar**g+1
9	DM1(**sp**)←**r**b, **sp**←**sp**+2
10	DM1(**sp**+K)←**r**b

Description

Store contents of a general register to DM1 (data memory 1) using various addressing modes. When 8 bit mode (b8) is used, the 8 lsb of the register are placed in the memory at given address. The other 8 bits in the same word in the memory are not affected.

Flags

No flags affected

Comment

Syntax no.5 can be used without the constant K, in which case it will be assumed to be 0.
Operation no.6: **ar**h will only be set to **bot**h when **ar**h=**top**h. There is nothing preventing **ar**h from becoming larger than **top**h at the **ar**h←**ar**h+**step**h operation.

Example

```
st1     (ar0++%),r13
```

DBLLD — double load

Syntax

1 **dblld** **r**d**,(ar**a**),r**f**,(ar**b**)**
2 **dblld** **r**d**,(ar**a**++),r**f**,(ar**b**++)**
3 **dblld** **r**d**,(--ar**a**),r**f**,(--ar**b**)**
4 **dblld** **r**d**,(ar**c**++%),r**f**,(ar**e**++%)**
5 **dblld** **r**d**,br(ar**a**),r**f**,br(ar**b**)**

$a\epsilon[0..3]$, $b\epsilon[0..3]$, $c\epsilon[0..1]$, $d\epsilon[0..31]$, $e\epsilon[0..1]$, $f\epsilon[0..31]$

Operation

1 **r**d←DM0(**ar**a) and **r**f←DM1(**ar**b)
2 **r**d←DM0(**ar**a), if $a\epsilon[0..1]$ then **ar**a←**ar**a+**step**a else **ar**a←**ar**a+2, and
rf←DM1(**ar**b), if $b\epsilon[0..1]$ then **ar**b←**ar**b+**step**b else **ar**b←**ar**b+2
3 if $a\epsilon[0..1]$ then **ar**a←**ar**a-**step**a else **ar**a←**ar**a-2, **r**d←DM0(**ar**a), and
if $b\epsilon[0..1]$ then **ar**b←**ar**b-**step**b else **ar**b←**ar**b-2, **r**f←DM1(**ar**b)
4 **r**d←DM0(**ar**c), if **ar**c=**top**c then **ar**c←**bot**c else **ar**c←**ar**c+**step**c, and
rf←DM1(**ar**e), if **ar**e=**top**e then **ar**e←**bot**e else **ar**e←**ar**e+**step**e
5 **r**d←DM0(bitrev(**ar**a)), if $a\epsilon[0..1]$ then **ar**a←**ar**a+**step**a else **ar**a←**ar**a+2, and
rf←DM1(bitrev(**ar**b)), if $b\epsilon[0..1]$ then **ar**b←**ar**b+**step**b else **ar**b←**ar**b+2

Description

Load one general register with data from DM0 (data memory 0) and another general register with data from DM1 (data memory 1).

Flags

No flags affected

Comment

In the syntax section the supported addressing modes are listed. DM0 and DM1 do not have to use the same adressing mode.

Example

```
dblld    r1,(ar1),r9,(ar0++)
```

Instruction 2.7: DBLLD — double load

DBLST — double store

Syntax

1 **dblst** **(ar*d*),r*a*,(ar*f*),r*b***
2 **dblst** **(ar*d*++),r*a*,(ar*f*++),r*b***
3 **dblst** **(--ar*d*),r*a*,(--ar*f*),r*b***
4 **dblst** **(ar*g*++%),r*a*,(ar*h*++%),r*b***
5 **dblst** **br(ar*d*),r*a*,br(ar*f*),r*b***

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..3]$, $f\epsilon[0..3]$, $g\epsilon[0..1]$, $h\epsilon[0..1]$

Operation

1 DM0(**ar***d*)←**r***a* and DM1(**ar***f*)←**r***b*
2 DM0(**ar***d*)←**r***a*, if $d\epsilon[0..1]$ then **ar***d*←**ar***d*+**step***d* else **ar***d*←**ar***d*+2, and
DM1(**ar***f*)←**r***b*, if $f\epsilon[0..1]$ then **ar***f*←**ar***f*+**step***f* else **ar***f*←**ar***f*+2
3 if $d\epsilon[0..1]$ then **ar***d*←**ar***d*-**step***d* else **ar***d*←**ar***d*-2, DM0(**ar***d*)←**r***a*, and
if $f\epsilon[0..1]$ then **ar***f*←**ar***f*-**step***f* else **ar***f*←**ar***f*-2, DM1(**ar***f*)←**r***b*
4 DM0(**ar***g*)←**r***a*), if **ar***g*=**top***g* then **ar***g*←**bot***g* else **ar***g*←**ar***g*+**step***g*, and
DM1(**ar***h*)←**r***b*), if **ar***h*=**top***h* then **ar***h*←**bot***h* else **ar***h*←**ar***h*+**step***h*
5 DM0(bitrev(**ar***d*))←**r***a*, if $d\epsilon[0..1]$ then **ar***d*←**ar***d*+**step***d* else **ar***d*←**ar***d*+2, and
DM1(bitrev(**ar***f*))←**r***b*, if $f\epsilon[0..1]$ then **ar***f*←**ar***f*+**step***f* else **ar***f*←**ar***f*+2

Description

Store contents of one general register to DM0 (data memory 0) and contents of another general register to DM1 (data memory 1).

Flags

No flags affected

Comment

In the syntax section the supported addressing modes are listed. DM0 and DM1 do not have to use the same adressing mode.

Example

```
dblst    (ar2++),r31,br(ar0),r29
```

Instruction 2.8: DBLST — double store

IN — read IO port to register

Syntax

```
1   in          rd,K
```

$d\epsilon[0..31]$, $K\epsilon[\$0000..\$FFFF]$

Operation

1 **r**$d \leftarrow \mathrm{PORT}(K)$

Description

Read data from an IO port and store in a general register.

Flags

No flags affected

Comment

Note: The address is more than the peripheral number. Read the *Senior Peripheral Firmware User Manual* for more information.

Example

```
in        r12,$0010
```

Instruction 2.9: IN — read IO port to register

OUT — write register to IO port

Syntax

```
1   out        K,ra
```

$a\epsilon[0..31]$, $K\epsilon[\$0000..\$FFFF]$

Operation

1 PORT(K)$\leftarrow$**r**a

Description

Write data from a general register to an IO port

Flags

No flags affected

Comment

Note: The address is more than the peripheral number. Read the *Senior Peripheral Firmware User Manual* for more information.

Example

```
out     $0012, r10
```

Instruction 2.10: OUT — write register to IO port

DBLIN — IO to DM0 and DM1

Syntax

1	**dblin**	**(ar**d**),(ar**f**),**K
2	**dblin**	**(ar**d**++),(ar**f**++),**K
3	**dblin**	**(--ar**d**),(--ar**f**),**K
4	**dblin**	**(ar**g**++%),(ar**h**++%),**K
5	**dblin**	**br(ar**d**),br(ar**f**),**K

$d\epsilon[0..3]$, $f\epsilon[0..3]$, $g\epsilon[0..1]$, $h\epsilon[0..1]$, $K\epsilon[0..63]$

Operation

1 DM0(**ar**d)←PORT0(K) and DM1(**ar**f)← PORT1(K)

2 DM0(**ar**d)←PORT0(K), if $d\epsilon[0..1]$ then **ar**d←**ar**d+**step**d else **ar**d←**ar**d+2, and
DM1(**ar**f)←PORT1(K), if $f\epsilon[0..1]$ then **ar**f←**ar**f+**step**f else **ar**f←**ar**f+2

3 if $d\epsilon[0..1]$ then **ar**d←**ar**d-**step**d else **ar**d←**ar**d-2, DM0(**ar**d)←PORT0(K), and
if $f\epsilon[0..1]$ then **ar**f←**ar**f-**step**f else **ar**f←**ar**f-2, DM1(**ar**f)←PORT1(K)

4 DM0(**ar**g)←PORT0(K), if **ar**g=**top**g then **ar**g←**bot**g else **ar**g←**ar**g+**step**g, and
DM1(**ar**h)←PORT1(K), if **ar**h=**top**h then **ar**h←**bot**h else **ar**h←**ar**h+**step**h

5 DM0(bitrev(**ar**d))←PORT0(K), if $d\epsilon[0..1]$ then **ar**d←**ar**d+**step**d else **ar**d←**ar**d+2, and
DM1(bitrev(**ar**f))←PORT1(K), if $f\epsilon[0..1]$ then **ar**f←**ar**f+**step**f else **ar**f←**ar**f+2

Description

Store I/O input data to DM0 and DM1. The 16 lsb of the I/O input data is stored in DM0 and the 16 msb is stored in DM1.

Flags

No flags affected

Comment

DM0 and DM1 do not have to use the same adressing mode. The address constant should be the peripheral address, and unlike the address used for IN and OUT it shouldn't contain special targeted registers in the lsb.

Example

```
dblin    (ar2++),br(ar0),0x14
```

Instruction 2.11: DBLIN — IO to DM0 and DM1

DBLOUT — IO from DM0 and DM1

Syntax

1 **dblout** K,**(ar**a**),(ar**b**)**
2 **dblout** K,**(ar**a**++),(ar**b**++)**
3 **dblout** K,**(--ar**a**),(--ar**b**)**
4 **dblout** K,**(ar**c**++%),(ar**e**++%)**
5 **dblout** K,**br(ar**a**),br(ar**b**)**

$a \epsilon [0..3]$, $b \epsilon [0..3]$, $c \epsilon [0..1]$, $e \epsilon [0..1]$, $K \epsilon [0..63]$

Operation

1 PORT0(K)←DM0(**ar**a) and PORT1(K)←DM1(**ar**b)
2 PORT0(K)←DM0(**ar**a), if $a \epsilon [0..1]$ then **ar**a←**ar**a+**step**a else **ar**a←**ar**a+2, and PORT1(K)←DM1(**ar**b), if $b \epsilon [0..1]$ then **ar**b←**ar**b+**step**b else **ar**b←**ar**b+2
3 if $a \epsilon [0..1]$ then **ar**a←**ar**a-**step**a else **ar**a←**ar**a-2, PORT0(K)←DM0(**ar**a), and if $b \epsilon [0..1]$ then **ar**b←**ar**b-**step**b else **ar**b←**ar**b-2, PORT1(K)←DM1(**ar**b)
4 PORT0(K)←DM0(**ar**c), if **ar**c=**top**c then **ar**c←**bot**c else **ar**c←**ar**c+**step**c, and PORT1(K)←DM1(**ar**e), if **ar**e=**top**e then **ar**e←**bot**e else **ar**e←**ar**e+**step**e
5 PORT0(K)←DM0(bitrev(**ar**a)), if $a \epsilon [0..1]$ then **ar**a←**ar**a+**step**a else **ar**a←**ar**a+2, and PORT1(K)←DM1(bitrev(**ar**b)), if $b \epsilon [0..1]$ then **ar**b←**ar**b+**step**b else **ar**b←**ar**b+2

Description

Output data from DM0 to the 16 lsb of I/O data out and output data from DM1 to the 16 msb of I/O data out.

Flags

No flags affected

Comment

DM0 and DM1 do not have to use the same adressing mode. The address constant should only contain the peripheral number.

Example

```
dblout   0x5,(ar1),(ar0++)
```

Instruction 2.12: DBLOUT — IO from DM0 and DM1

OUT0 — out DM0 data to IO port

Syntax

```
1   out0        K,ara
2   out0        K,(ara++)
3   out0        K,(--ara)
```

$a \epsilon [0..3]$, $K \epsilon$[$0000..$FFFF]

Operation

1 PORT(K)←DM0(**ar**a)
2 PORT(K)←DM0(**ar**a),if $a \epsilon [0..1]$ then **ar**a←**ar**a+**step**a else **ar**a←**ar**a+2
3 if $a \epsilon [0..1]$ then **ar**a←**ar**a-**step**a else **ar**a←**ar**a-2, PORT(K)←DM0(**ar**a)

Description

Write data from DM0 to an IO port, DM0 address is from register **ar**a, output data is 16 bit

Flags

No flags affected

Comment

Note: The address is more than the peripheral number. Read the *Senior Peripheral Firmware User Manual* for more information.

Example

```
out0    $0012, (ar0++)
```

Instruction 2.13: OUT0 — out DM0 data to IO port

OUT1 — out DM1 data to IO port

Syntax

1 **out1** K,**ar**a
2 **out1** K,**(ar**a**++)**
3 **out1** K,**(--ar**a**)**
$a\epsilon[0..3]$, $K\epsilon[\$0000..\$FFFF]$

Operation

1 PORT(K)←DM1(**ar**a)
2 PORT(K)←DM1(**ar**a),if $a\epsilon[0..1]$ then **ar**a←**ar**a+**step**a else **ar**a←**ar**a+2
3 if $a\epsilon[0..1]$ then **ar**a←**ar**a-**step**a else **ar**a←**ar**a-2, PORT(K)←DM1(**ar**a)

Description

Write data from DM1 to an IO port, DM1 address is from register **ar**a, output data is 16 bit

Flags

No flags affected

Comment

Note: The address is more than the peripheral number. Read the *Senior Peripheral Firmware User Manual* for more information.

Example

```
out1     $0012, (ar3)
```

Instruction 2.14: OUT1 — out DM1 data to IO port

3 Short arithmetic instructions, 16'b

The short arithmetic instructions concern 16 bit arithmetic operations. All instructions can make use of conditional execution (depending om ALU/MAC flags status) when no constant operands are used. Because of code size limitations constant operands are restricted to a size of 12 bits, but always sign extended to 16 bits before use. Only the **cmp** instruction can carry a 16 bit constant. Table 3.1 lists the short arithmetic instructions.

Mnemonic	Description	Page
add	addition of registers and constants	32
addn	addition of registers and constants with no flag change	33
addc	addition of registers and constants with carry	34
adds	addition of registers and constants with saturation	35
sub	subtraction of registers and constants	36
subn	subtraction of registers and constants with no flag change	37
subc	subtraction of registers and constants with carry	38
subs	subtraction of registers and constants with saturation	39
cmp	compare registers and constants	40
max	return maximum of registers and constants	41
min	return minimum of registers and constants	42
abs	return absolute of a register	43
ext	8 to 16 bit sign extension on registers	44

Table 3.1: short arithmetic instructions

ADD — addition

Syntax

1	**add**[.*cdt*]	**r***d*,**r***a*,**r***b*
2	**add**	**r***d*,*K*,**r***b*
3	**add**	**r***d*,**r***b*,*K*

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $K\epsilon[-2048..2047]$

Operation

1 if *cdt* is true: **r***d*←**r***a*+**r***b*

2,3 **r***d*←*K*+**r***b*

Description

1. Add general registers using a condition

2,3. Add a 12 bit signed constant (with 16 bit signed extension) and a general register

Flags

AV	AC	AN	AZ
U	U	U	U

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
add.ne   r7,r12           ;the same as add.ne r7,r12,r7
add      r23,r0,-31
add      r17,%11001001    ;the same as add r17,%11001001,r17
add      r2,c_tick,r6     ;c_tick is previously defined as a constant
```

Instruction 3.1: ADD — addition

ADDN — addition with no flag change

Syntax

1 **addn**[.*cdt*] **r***d*,**r***a*,**r***b*
2 **addn** **r***d*,*K*,**r***b*
3 **addn** **r***d*,**r***b*,*K*

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $K\epsilon[-2048..2047]$

Operation

1 if *cdt* is true: **r***d*←**r***a*+**r***b*
2,3 **r***d*←*K*+**r***b*

Description

1. Add general registers using a condition
2,3. Add a 12 bit signed constant (with 16 bit signed extension) and a general register

Flags

No flags affected

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
addn.ne  r7,r12              ;the same as addn.ne r7,r12,r7
addn     r23,r0,-31
addn     r17,%11001001       ;the same as addn r17,%11001001,r17
addn     r2,c_tick,r6        ;c_tick is previously defined as a constant
```

Instruction 3.2: ADDN — addition with no flag change

ADDC — addition with carry

Syntax

1	**addc**[.*cdt*]	**r***d*,**r***a*,**r***b*
2	**addc**	**r***d*,*K*,**r***b*
3	**addc**	**r***d*,**r***b*,*K*

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $K\epsilon[-2048..2047]$

Operation

1 if *cdt* is true: $\mathbf{r}d \leftarrow \mathbf{r}a + \mathbf{r}b + AC$

2,3 $\mathbf{r}d \leftarrow K + \mathbf{r}b + AC$

Description

1. Add general registers and carry bit using a condition

2,3. Add a 12 bit signed constant (with 16 bit signed extension), a general register and carry bit

Flags

AV	AC	AN	AZ
U	U	U	U

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
addc.ne  r7,r12           ;the same as addc.ne r7,r12,r7
addc     r23,r0,-31
addc     r17,%11001001    ;the same as addc r17,%11001001,r17
addc     r2,c_tick,r6     ;c_tick is previously defined as a constant
```

Instruction 3.3: ADDC — addition with carry

ADDS — addition with saturation

Syntax

1	**adds**[.*cdt*]	**r***d*,**r***a*,**r***b*
2	**adds**	**r***d*,K,**r***b*
3	**adds**	**r***d*,**r***b*,K

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $K\epsilon[-2048..2047]$

Operation

1 if *cdt* is true: **r***d*←saturation(**r***a*+**r***b*)0

2,3 **r***d*←saturation(K+**r***b*)

Description

1. Add with saturation general registers using a condition

2,3. Add with saturation a 12 bit signed constant (with 16 bit signed extension) and a general register

Flags

AV	AC	AN	AZ
U	U	U	U

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
adds.ne  r7,r12             ;the same as adds.ne r7,r12,r7
adds     r23,r0,-31
adds     r17,%11001001      ;the same as adds r17,%11001001,r17
adds     r2,c_tick,r6       ;c_tick is previously defined as a constant
```

Instruction 3.4: ADDS — addition with saturation

SUB — subtraction

Syntax

1 **sub**[**.**cdt] **r**d**,r**a**,r**b
2 **sub** **r**d**,**K**,r**b
$a\epsilon$[0..31], $b\epsilon$[0..31], $d\epsilon$[0..31], $K\epsilon$[-2048..2047]

Operation

1 if *cdt* is true: **r**d←**r**a-**r**b
2 **r**d←K-**r**b

Description

1. Subtract general registers using a condition
2. Subtract a general register from a 12 bit signed constant (with 16 bit signed extension)

Flags

AV	AC	AN	AZ
U	U	U	U

Comment

It is possible to omit one source register (**r**a or **r**b) in the syntax, in which case the destination register (**r**d) will be used in its place.

Example

```
sub.mi   r3,r4          ;the same as sub.mi r3,r4,r3
sub      r14,0x7A,r0
```

Instruction 3.5: SUB — subtraction

SUBN — subtraction with no flag change

Syntax

1	**subn**[.*cdt*]	**r***d*,**r***a*,**r***b*
2	**subn**	**r***d*,*K*,**r***b*

$a \epsilon [0..31]$, $b \epsilon [0..31]$, $d \epsilon [0..31]$, $K \epsilon [-2048..2047]$

Operation

1 if *cdt* is true: **r***d*←**r***a*-**r***b*
2 **r***d*←*K*-**r***b*

Description

1. Subtract general registers using a condition
2. Subtract a general register from a 12 bit signed constant (with 16 bit signed extension)

Flags

No flags affected

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
subn.mi  r3,r4            ;the same as subn.mi r3,r4,r3
subn     r14,0x7A,r0
```

Instruction 3.6: SUBN — subtraction with no flag change

SUBC — subtraction with carry

Syntax

1	**subc**[.*cdt*]	**r***d*,**r***a*,**r***b*
2	**subc**	**r***d*,*K*,**r***b*

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $K\epsilon[-2048..2047]$

Operation

1 if *cdt* is true: $\mathbf{r}d \leftarrow \mathbf{r}a - \mathbf{r}b - 1 + AC = \mathbf{r}a + \mathbf{r}b \oplus \$FFFF + AC$

2 $\mathbf{r}d \leftarrow K - \mathbf{r}b - 1 + AC = K + \mathbf{r}b \oplus \$FFFF + AC$

Description

1. Subtract general registers and add carry if condition is true
2. Subtract a general register from a 12 bit signed constant (with 16 bit signed extension) and add carry

Flags

AV	AC	AN	AZ
U	U	U	U

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
subc.mi  r3,r4           ;the same as subc.mi r3,r4,r3
subc     r14,0x7A,r0
```

Instruction 3.7: SUBC — subtraction with carry

SUBS — subtraction with saturation

Syntax

1 **subs**[.*cdt*] **r***d*,**r***a*,**r***b*
2 **subs** **r***d*,*K*,**r***b*

$a \epsilon [0..31]$, $b \epsilon [0..31]$, $d \epsilon [0..31]$, $K \epsilon [-2048..2047]$

Operation

1 if *cdt* is true: **r***d*←saturation(**r***a*-**r***b*)
2 **r***d*←saturation(*K*-**r***b*)

Description

1. Subtract with saturation general registers using a condition
2. Subtract with saturation a general register from a 12 bit signed constant (with 16 bit signed extension)

Flags

AV	AC	AN	AZ
U	U	U	U

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
subs.mi  r3,r4            ;the same as subs.mi r3,r4,r3
subs     r14,0x7A,r0
```

Instruction 3.8: SUBS — subtraction with saturation

CMP — compare

Syntax

1	**cmp**[.*cdt*]	**r***a*,**r***b*
2	**cmp**	*K*,**r***b*

$a \epsilon$[0..31], $b \epsilon$[0..31], $K \epsilon$[-32768..32767]

Operation

1 if *cdt* is true: ALUflags$\curvearrowleft$**r***a*-**r***b*

2 ALUflags$\curvearrowleft$*K*-**r***b*

Description

1. Compare contents of general registers using a condition
2. Compare a constant and a general register

Flags

AV	AC	AN	AZ
U	U	U	U

Comment

Example

```
cmp.pl   r7,r12
cmp      -23456,r3
```

Instruction 3.9: CMP — compare

MAX — return maximum value

Syntax

1	**max**[**.**cdt]	**r**d**,r**a**,r**b
2	**max**	**r**d**,**K**,r**a
3	**max**	**r**d**,r**a**,**K

$a \epsilon$[0..31], $b \epsilon$[0..31], $d \epsilon$[0..31], $K \epsilon$[-2048..2047]

Operation

1 if *cdt* is true: **r**d←signed maximum(**r**a,**r**b)
2,3 **r**d←signed maximum(**r**a,K)

Description

1. Return maximum of the general source registers, using a condition
2,3. Return maximum of a 12 bit signed constant (with 16 bit signed extension) and a general register

Flags

No flags affected

Comment

It is possible to omit one source register (**r**a or **r**b) in the syntax, in which case the destination register (**r**d) will be used in its place.

Example

```
max.eq   r11,r7         ;the same as max.eq r11,r7,r11
max      r3,$F44        ;the same as max r3,$F44,r3
max      r1,r2,-3
```

Instruction 3.10: MAX — return maximum value

MIN — return minimum value

Syntax

1	**min**[**.** *cdt*]	**r***d***,r***a***,r***b*
2	**min**	**r***d***,***K***,r***a*
3	**min**	**r***d***,r***a***,***K*

$a \in [0..31]$, $b \in [0..31]$, $d \in [0..31]$, $K \in [-2048..2047]$

Operation

1	if *cdt* is true: **r***d*←signed minimum(**r***a*,**r***b*)
2,3	**r***d*←signed minimum(**r***a*,*K*)

Description

1. Return minimum of the general source registers, using a condition

2,3. Return minimum of a 12 bit signed constant (with 16 bit signed extension) and a general register

Flags

No flags affected

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
min.eq   r11,r7         ;the same as min.eq r11,r7,r11
min      r3,$F44        ;the same as min r3,$F44,r3
min      r1,r2,-3
```

Instruction 3.11: MIN — return minimum value

ABS — return absolute value

Syntax

1 **abs**[.*cdt*] [**sat**] **r***d*,**r***a*

$a\epsilon[0..31]$, $d\epsilon[0..31]$

Operation

1 if *cdt* is true: **r***d*←absolute(**r***a*) or **r***d*←absolute(saturation(**r***a*))

Description

Return absolute value of an optionally saturated general register, using a condition

Flags

No flags affected

Comment

Observe: for the special case when **r***a*=$8000 saturation is needed

Example

```
abs      r12,r13
```

Instruction 3.12: ABS — return absolute value

EXT — 8 to 16 bit sign extension

Syntax

1 **ext**[.*cdt*] **r***d*,**r***a*

$a \epsilon [0..31]$, $d \epsilon [0..31]$

Operation

1 if *cdt* is true: **r***d*←sign_extension(**r***a*[7..0])

Description

1. Extending a 8 bit value into a 16 bit value, using a condition.

Flags

No flags affected

Comment

It is possible to omit the source register (**r***a*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
ext.eq   r3,r2
ext      r1                 ;Write back to r1
```

Instruction 3.13: EXT — 8 to 16 bit sign extension

4 Short logic instructions, 16'b

The short logic instructions concern 16 bit logic operations. All instructions can make use of conditional execution (depending om ALU/MAC flags status) when no constant operands are used. Because of code size limitations constant operands are restricted to a size of 12 bits, but always sign extended to 16 bits before use. Table 4.1 lists the short logic instructions.

Mnemonic	Description	Page
and	logic and between registers and constants	46
andn	logic and between registers and constants with no flag change	47
or	logic or between registers and constants	48
orn	logic or between registers and constants with no flag change	49
xor	logic xor between registers and constants	50
xorn	logic xor between registers and constants with no flag change	51

Table 4.1: short logic instructions

AND — logic and

Syntax

1	**and**[.*cdt*]	**r***d*,**r***a*,**r***b*
2	**and**	**r***d*,*K*,**r***a*
3	**and**	**r***d*,**r***a*,*K*

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $K\epsilon[-2048..2047]$

Operation

1 if *cdt* is true: **r***d*←**r***a*•**r***b*

2,3 **r***d*←**r***a*•*K*

Description

1. Logic and between general registers, using a condition

2,3. Logic and between a 12 bit signed constant (with 16 bit signed extension) and a general register

Flags

AV	AC	AN	AZ
U	U	U	U

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
and.ugt  r3,r2            ;the same as and.ugt r3,r2,r3
and      r14,$83A         ;the constant $83A will sign extend to $F83A
```

Instruction 4.1: AND — logic and

ANDN — logic and with no flag change

Syntax

1 **andn**[.*cdt*] **r***d*,**r***a*,**r***b*
2 **andn** **r***d*,*K*,**r***a*
3 **andn** **r***d*,**r***a*,*K*
$a\epsilon$[0..31], $b\epsilon$[0..31], $d\epsilon$[0..31], $K\epsilon$[-2048..2047]

Operation

1 if *cdt* is true: **r***d*←**r***a*•**r***b*
2,3 **r***d*←**r***a*•*K*

Description

1. Logic and between general registers, using a condition
2,3. Logic and between a 12 bit signed constant (with 16 bit signed extension) and a general register

Flags

No flags affected

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
andn.ugt r3,r2            ;the same as andn.ugt r3,r2,r3
andn     r14,$83A         ;the constant $83A will sign extend to $F83A
```

Instruction 4.2: ANDN — logic and with no flag change

OR — logic or

Syntax

1	**or**[**.** *cdt*]	**r***d* **,r***a* **,r***b*
2	**or**	**r***d* **,***K***,r***a*
3	**or**	**r***d* **,r***a* **,***K*

$a\epsilon$[0..31], $b\epsilon$[0..31], $d\epsilon$[0..31], $K\epsilon$[-2048..2047]

Operation

1 if *cdt* is true: **r***d*←**r***a*∨**r***b*

2,3 **r***d*←**r***a*∨*K*

Description

1. Logic or between general registers, using a condition

2,3. Logic or between a 12 bit signed constant (with 16 bit signed extension) and a general register

Flags

AV	AC	AN	AZ
U	U	U	U

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
or.ugt   r3,r2           ;the same as or.ugt r3,r2,r3
or       r14,$83A        ;the constant $83A will sign extend to $F83A
```

Instruction 4.3: OR — logic or

ORN — logic or with no flag change

Syntax

1 **orn**[.*cdt*] **r***d*,**r***a*,**r***b*
2 **orn** **r***d*,*K*,**r***a*
3 **orn** **r***d*,**r***a*,*K*
$a\epsilon$[0..31], $b\epsilon$[0..31], $d\epsilon$[0..31], $K\epsilon$[-2048..2047]

Operation

1 if *cdt* is true: $\mathbf{r}d \leftarrow \mathbf{r}a \vee \mathbf{r}b$
2,3 $\mathbf{r}d \leftarrow \mathbf{r}a \vee K$

Description

1. Logic or between general registers, using a condition
2,3. Logic or between a 12 bit signed constant (with 16 bit signed extension) and a general register

Flags

No flags affected

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
orn.ugt  r3,r2          ;the same as orn.ugt r3,r2,r3
orn      r14,$83A       ;the constant $83A will sign extend to $F83A
```

Instruction 4.4: ORN — logic or with no flag change

XOR — logic Xor

Syntax

1	**xor**[**.** *cdt*]	**r***d* **,r***a* **,r***b*
2	**xor**	**r***d* **,** *K* **,r***a*
3	**xor**	**r***d* **,r***a* **,** *K*

$a \epsilon [0..31]$, $b \epsilon [0..31]$, $d \epsilon [0..31]$, $K \epsilon [-2048..2047]$

Operation

1 if *cdt* is true: $\mathbf{r}d \leftarrow \mathbf{r}a \oplus \mathbf{r}b$

2,3 $\mathbf{r}d \leftarrow \mathbf{r}a \oplus K$

Description

1. Logic xor between general registers, using a condition

2,3. Logic xor between a 12 bit signed constant (with 16 bit signed extension) and a general register

Flags

AV	AC	AN	AZ
U	U	U	U

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
xor.ugt  r3,r2              ;the same as xor.ugt r3,r2,r3
xor      r14,$83A           ;the constant $83A will sign extend to $F83A
```

Instruction 4.5: XOR — logic Xor

XORN — logic xor with no flag change

Syntax

1 **xorn**[.*cdt*] **r***d*,**r***a*,**r***b*
2 **xorn** **r***d*,*K*,**r***a*
3 **xorn** **r***d*,**r***a*,*K*

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $K\epsilon[-2048..2047]$

Operation

1 if *cdt* is true: $\mathbf{r}d \leftarrow \mathbf{r}a \oplus \mathbf{r}b$
2,3 $\mathbf{r}d \leftarrow \mathbf{r}a \oplus K$

Description

1. Logic xor between general registers, using a condition
2,3. Logic xor between a 12 bit signed constant (with 16 bit signed extension) and a general register

Flags

No flags affected

Comment

It is possible to omit one source register (**r***a* or **r***b*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
xorn.ugt r3,r2          ;the same as xorn.ugt r3,r2,r3
xorn     r14,$83A       ;the constant $83A will sign extend to $F83A
```

Instruction 4.6: XORN — logic xor with no flag change

5 Short shift instructions, 16'b

The short shift instructions concern 16 bit shift and rotate operations. All instructions can make use of conditional execution (depending om ALU/MAC flags status). All instructions use either a register or a constant to note the number of shift/rotate operations.
OBSERVE, when more than 16 shift/rotate operations are used results may be unexpected. Table 5.1 lists the short shift instructions.

Mnemonic	Description	Page
asr	arithmetic shift right	54
asl	arithmetic shift left	55
lsr	logic shift right	56
lsl	logic shift left	57
ror	rotate right	58
rol	rotate left	59
rcr	rotate right through carry	60
rcl	rotate left through carry	61

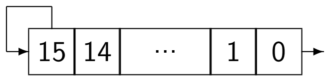
Table 5.1: short shift instructions

ASR — arithmetic shift right

Syntax

1 **asr**[.*cdt*] **r***d*,**r***a*,**r***b*
2 **asr**[.*cdt*] **r***d*,**r***a*,*K*
$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $K\epsilon[0..31]$

Operation

1,2 

Description

r*a* is the register to be shifted, **r***b* or *K* holds the number of steps to shift, the result is stored in register **r***d*

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (**r***a*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
asr.ne   r12,7              ;the same as asr.ne r12,r12,7
```

Instruction 5.1: ASR — arithmetic shift right

ASL — arithmetic shift left

Syntax

1 **asl**[.*cdt*] **r***d*,**r***a*,**r***b*
2 **asl**[.*cdt*] **r***d*,**r***a*,*K*
$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $K\epsilon[0..31]$

Operation

1,2	←	15	14	···	1	0	← 0

Description

r*a* is the register to be shifted, **r***b* or *K* holds the number of steps to shift, the result is stored in register **r***d*

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (**r***a*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
asl.eq   r12,r3            ;the same as asl.eq r12,r12,r3
```

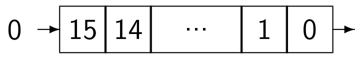
Instruction 5.2: ASL — arithmetic shift left

LSR — logic shift right

Syntax

1 **lsr**[.*cdt*] **r***d*,**r***a*,**r***b*
2 **lsr**[.*cdt*] **r***d*,**r***a*,*K*
$a \epsilon [0..31]$, $b \epsilon [0..31]$, $d \epsilon [0..31]$, $K \epsilon [0..31]$

Operation

1,2 

Description

r*a* is the register to be shifted, **r***b* or *K* holds the number of steps to shift, the result is stored in register **r***d*

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (**r***a*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
lsr.ne   r4,22             ;the same as lsr.ne r4,r4,22
```

Instruction 5.3: LSR — logic shift right

LSL — logic shift left

Syntax

1 **lsl**[.*cdt*] **r***d*,**r***a*,**r***b*
2 **lsl**[.*cdt*] **r***d*,**r***a*,*K*
$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $K\epsilon[0..31]$

Operation

1,2	←	15	14	···	1	0	←	0

Description

r*a* is the register to be shifted, **r***b* or *K* holds the number of steps to shift, the result is stored in register **r***d*

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (**r***a*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
lsl.eq   r3,7              ;the same as lsl.eq r3,r3,7
```

Instruction 5.4: LSL — logic shift left

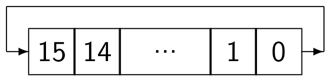
ROR — rotate right

Syntax

1 **ror**[**.** *cdt*] **r***d* **,r***a* **,r***b*
2 **ror**[**.** *cdt*] **r***d* **,r***a* **,***K*
$a \epsilon [0..31]$, $b \epsilon [0..31]$, $d \epsilon [0..31]$, $K \epsilon [0..15]$

Operation

1,2



Description

r*a* is the register to be rotated, **r***b* or K holds the number of steps to rotate, the result is stored in register **r***d*

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (**r***a*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
ror.ne   r5,2                ;the same as ror.ne r5,r5,2
```

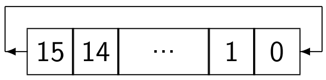
Instruction 5.5: ROR — rotate right

ROL — rotate left

Syntax

1 **rol**[.*cdt*] **r***d*,**r***a*,**r***b*
2 **rol**[.*cdt*] **r***d*,**r***a*,*K*
$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $K\epsilon[0..15]$

Operation

1,2 

Description

r*a* is the register to be rotated, **r***b* or *K* holds the number of steps to rotate, the result is stored in register **r***d*

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (**r***a*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
rol.eq   r15,r1            ;the same as rol.eq r15,r15,r1
```

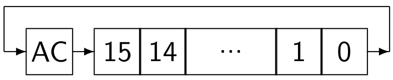
Instruction 5.6: ROL — rotate left

RCR — rotate right through carry

Syntax

1 **rcr**[.*cdt*] **r***d*,**r***a*,**r***b*
2 **rcr**[.*cdt*] **r***d*,**r***a*,*K*
$a \epsilon [0..31]$, $b \epsilon [0..31]$, $d \epsilon [0..31]$, $K \epsilon [0..15]$

Operation

1,2 

Description

r*a* is the register to be rotated, **r***b* or *K* holds the number of steps to rotate, the result is stored in register **r***d*

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (**r***a*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
rcr.ne   r7,7                ;the same as rcr.ne r7,r7,7
```

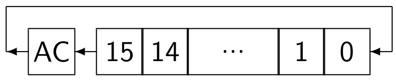
Instruction 5.7: RCR — rotate right through carry

RCL — rotate left through carry

Syntax

1 **rcl**[.*cdt*] **r***d*,**r***a*,**r***b*
2 **rcl**[.*cdt*] **r***d*,**r***a*,*K*
$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $K\epsilon[0..15]$

Operation

1,2 

Description

r*a* is the register to be rotated, **r***b* or *K* holds the number of steps to rotate, the result is stored in register **r***d*

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (**r***a*) in the syntax, in which case the destination register (**r***d*) will be used in its place.

Example

```
rcl.eq   r17,r2            ;the same as rcl.eq r17,r17,r2
```

Instruction 5.8: RCL — rotate left through carry

6 Long arithmetic instructions, 32'b

The long arithmetic instructions concern 32 bit arithmetic operations. All instructions (except **sublst0**, **sublst1**, **mul**, **mulld0**, **mulld1**, **muldblld**, **mac**, **macld0**, **macld1**, **mdm** and **conv**) can make use of conditional execution (depending om ALU/MAC flags status). Operations are carried out with 8 guard bits on all operands giving 40 bits internal resolution in the MAC unit. Table 6.1 lists the long arithmetic instructions.

Mnemonic	Description	Page
addl	long addition between accumulators and registers	64
subl	long subtraction between accumulators and registers	65
sublst0	long subtraction and store register to data memory 0	66
sublst1	long subtraction and store register to data memory 1	67
cmpl	long compare between accumulators and registers	68
absl	long absolute of accumulator or register	69
negl	long negation of accumulator or register	70
movel	long move of register to accumulator	71
clr	clear of accumulator	72
postop	post accumulator operation	73
mul	multiply	74
mulld0	multiply and load register from data memory 0	75
mulld1	multiply and load register from data memory 1	76
muldblld	multiply and double load	77
mac	multiply and accumulate	78
macld0	multiply, accumulate and load register from data memory 0	79
macld1	multiply, accumulate and load register from data memory 1	80
mdm	multiply and diminish	81
conv	convolution	82

Table 6.1: long arithmetic instructions

ADDL — long addition

Syntax

1 **addl**[.*cdt*] **acr***D*,**acr***A*,**acr***B*
2 **addl**[.*cdt*] **acr***D*,**acr***A*,**r***b*:**0**
3 **addl**[.*cdt*] **acr***D*,**acr***A*,**1:r***b*
4 **addl**[.*cdt*] **acr***D*,**acr***A*,**r***a*:**r***b*
5 **addl**[.*cdt*] [**sat**] [**rnd**] **r***d*,**acr***A*,**r***b*:**0**
6 **addl**[.*cdt*] [**sat**] [**rnd**] **r***d*,**acr***A*,**1:r***b*

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $A\epsilon[0..3]$, $B\epsilon[0..3]$, $D\epsilon[0..3]$

Operation

1 if *cdt* is true: **acr***D*←**acr***A*+**acr***B*
2 if *cdt* is true: **acr***D*←**acr***A*+zero_extension(**r***b*)
3 if *cdt* is true: **acr***D*←**acr***A*+sign_extension(**r***b*)
4 if *cdt* is true: **acr***D*←**acr***A*+register_extension(**r***b*)
5 if *cdt* is true: **r***d*←saturation(round(**acr***A*+zero_extension(**r***b*)))
6 if *cdt* is true: **r***d*←saturation(round(**acr***A*+sign_extension(**r***b*)))

Description

Operations are carried out with 8 guard bits on all operands

Flags

MV	MS	MN	MZ
U	U	U	U

Comment

Example

```
addl.meq acr2,acr1,acr0
addl     acr1,acr3,r2:0
addl.mvs sat rnd r15,acr0,1:r4
```

Instruction 6.1: ADDL — long addition

SUBL — long subtraction

Syntax

1 **subl**[.*cdt*] **acr***D*,**acr***A*,**acr***B*
2 **subl**[.*cdt*] **acr***D*,**acr***A*,**r***b*:**0**
3 **subl**[.*cdt*] **acr***D*,**acr***A*,**1:r***b*
4 **subl**[.*cdt*] **acr***D*,**acr***A*,**r***a*:**r***b*

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $A\epsilon[0..3]$, $B\epsilon[0..3]$, $D\epsilon[0..3]$

Operation

1 if *cdt* is true: **acr***D*←**acr***A*−**acr***B*
2 if *cdt* is true: **acr***D*←**acr***A*−zero_extension(**r***b*)
3 if *cdt* is true: **acr***D*←**acr***A*−sign_extension(**r***b*)
4 if *cdt* is true: **acr***D*←**acr***A*−register_extension(**r***b*)

Description

Operations are carried out with 8 guard bits on all operands

Flags

MV	MS	MN	MZ
U		U	U

Comment

Example

```
subl.mle acr2,acr1,1:r14
subl     acr1,acr3,r7:r8
```

Instruction 6.2: SUBL — long subtraction

SUBLST0 — long subtraction and store register to DM0

Syntax

1 **sublst0** [**sat**] [**rnd**] **r**d**,r**a**:0,acr**B**,(ar**f**++),r**b
2 **sublst0** [**sat**] [**rnd**] **r**d**,1:r**a**,acr**B**,(ar**f**++),r**b

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $f\epsilon[0..3]$, $B\epsilon[0..3]$

Operation

1 **r**d←saturation(round(zero_extension(**r**a)–**acr**B)) and
DM0(**ar**f)←**r**b, if $f\epsilon[0..1]$ then **ar**f←**ar**f+**step**f else **ar**f←**ar**f+1
2 **r**d←saturation(round(sign_extension(**r**a)–**acr**B)) and
DM0(**ar**f)←**r**b, if $f\epsilon[0..1]$ then **ar**f←**ar**f+**step**f else **ar**f←**ar**f+1

Description

Subtract an accumulator from an extended general purpose register and write back the result to a general purpose register. And store contents of a general purpose register to DM0 (data memory 0) using post-incremental addressing mode.

Flags

MV	MS	MN	MZ
U	U	U	U

Comment

Example

```
sublst0  sat rnd r5,1:r3,acr0,(ar3++),r7
```

Instruction 6.3: SUBLST0 — long subtraction and store register to DM0

SUBLST1 — long subtraction and store register to DM1

Syntax

1 **sublst1** [**sat**] [**rnd**] **r***d*,**r***a*:**0**,**acr***B*,(**ar***f*++),**r***b*
2 **sublst1** [**sat**] [**rnd**] **r***d*,**1**:**r***a*,**acr***B*,(**ar***f*++),**r***b*
$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $f\epsilon[0..3]$, $B\epsilon[0..3]$

Operation

1 **r***d*←saturation(round(zero_extension(**r***a*)−**acr***B*)) and
DM1(**ar***f*)←**r***b*, if $f\epsilon[0..1]$ then **ar***f*←**ar***f*+**step***f* else **ar***f*←**ar***f*+1
2 **r***d*←saturation(round(sign_extension(**r***a*)−**acr***B*)) and
DM1(**ar***f*)←**r***b*, if $f\epsilon[0..1]$ then **ar***f*←**ar***f*+**step***f* else **ar***f*←**ar***f*+1

Description

Subtract an accumulator from an extended general purpose register and write back the result to a general purpose register. And store contents of a general purpose register to DM1 (data memory 1) using post-incremental addressing mode.

Flags

MV	MS	MN	MZ
U	U	U	U

Comment

Example

```
sublst1  r2,r4:0,acr1,(ar0++),r14
```

Instruction 6.4: SUBLST1 — long subtraction and store register to DM1

CMPL — long compare

Syntax

1 **cmpl**[**.**cdt] **acr**A**,acr**B
2 **cmpl**[**.**cdt] **acr**A**,r**b**:0**
3 **cmpl**[**.**cdt] **acr**A**,1:r**b
4 **cmpl**[**.**cdt] **acr**A**,r**a**:r**b
$a\epsilon[0..31]$, $b\epsilon[0..31]$, $A\epsilon[0..3]$, $B\epsilon[0..3]$

Operation

1 if *cdt* is true: MACflags$\curvearrowleft$saturation(**acr**A−**acr**B)
2 if *cdt* is true: MACflags$\curvearrowleft$saturation(**acr**A−zero_extension(**r**b))
3 if *cdt* is true: MACflags$\curvearrowleft$saturation(**acr**A−sign_extension(**r**b))
4 if *cdt* is true: MACflags$\curvearrowleft$saturation(**acr**A−register_extension(**r**b))

Description

Operations are carried out with 8 guard bits on all operands

Flags

MV	MS	MN	MZ
U	U	U	U

Comment

Example

```
cmpl.mne acr3,acr2
cmpl     acr1,r7:0
```

Instruction 6.5: CMPL — long compare

ABSL — long absolute

Syntax

1 **absl**[**.** *cdt*] [**sat**] **acr***D* **,acr***A*
2 **absl**[**.** *cdt*] [**sat**] **acr***D*[**.h**|**.l**]**,r***a*
*a*ϵ[0..31], *A*ϵ[0..3], *D*ϵ[0..3]

Operation

1 if *cdt* is true: **acr***D*←saturation(absolute(**acr***A*))
2 if *cdt* is true; if **h** is used **acr***D*←saturation(absolute(**r***a*:[0..0])) else **acr***D*←saturation(absolute([0..0]:**r***a*))

Description

Return the absolute value of an accumulator or an extended general register.
The [**.h**|**.l**] option will select either the high order bits (bits 31 through 16) when using **h** or the low order bits (bits 15 through 0) when using **l** as destination for **r***a* in **acr***D*. When using **l**, the value will be sign extended up to 32 bits. Using **h** will clear the low order bits.

Flags

MV	MS	MN	MZ
U	U	U	U

Comment

Example

```
absl.mle sat acr2,acr3
absl     acr0.h,r4
```

Instruction 6.6: ABSL — long absolute

NEGL — long negation

Syntax

1 **negl**[.*cdt*] [**sat**] **acr***D*,**acr***A*
2 **negl**[.*cdt*] [**sat**] **acr***D*[.**h**|.**l**],**r***a*
a∊[0..31], *A*∊[0..3], *D*∊[0..3]

Operation

1 if *cdt* is true: **acr***D*←sat(neg(**acr***A*))
2 if *cdt* is true; if **h** is used **acr***D*←sat(neg(**r***a*:[0..0]) else **acr***D*←sat(neg([0..0]:**r***a*))

Description

Return the negated value of an accumulator or an extended general register.
The [.**h**|.**l**] option will select either the high order bits (bits 31 through 16) when using **h** or the low order bits (bits 15 through 0) when using **l** (default) as destination for **r***a* in **acr***D*. When using **l**, the value will be sign extended up to 32 bits. Using **h** will clear the low order bits.

Flags

MV	MS	MN	MZ
U	U	U	U

Comment

Example

```
negl.mne acr1,acr2
negl     sat acr3.l,r17
```

Instruction 6.7: NEGL — long negation

MOVEL — long move

Syntax

1 **movel**[**.***cdt*] **acr***D***,r***a***:r***b*
2 **movel**[**.***cdt*] **acr***D*[**.h**|**.l**]**,r***a*
$a\epsilon[0..31]$, $b\epsilon[0..31]$, $D\epsilon[0..3]$

Operation

1 if *cdt* is true: **acr***D*←**r***a*:**r***b*
2 if *cdt* is true: if **h** is used **acr***D*←sat(**r***a*:[0..0]) else **acr***D*←sat([0..0]:**r***a*)

Description

The [**.h**|**.l**] option will select either the high order bits (bits 31 through 16) when using **h** or the low order bits (bits 15 through 0) when using **l** (default) as destination for **r***a* in **acr***D*. When using **l**, the value will be sign extended up to 32 bits. Using **h** will clear the low order bits.

Flags

MV	MS	MN	MZ
		U	U

Comment

Example

```
movel    acr2,r14:r2
movel    acr0,r5             ;low order bits of acr0 will be target
```

Instruction 6.8: MOVEL — long move

CLR — clear accumulator

Syntax

1 **clr**[**.**cdt] **acr**D
$D \epsilon [0..3]$

Operation

1 if *cdt* is true: **acr**$D \leftarrow 0$

Description

Clear an accumulator, using a condition.

Flags

MV	MS	MN	MZ
		U	U

Comment

Example

```
clr.eq   acr2
```

Instruction 6.9: CLR — clear accumulator

POSTOP — post accumulator operation

Syntax

1 **postop**[**.**cdt] [sf] [**rnd**] [**sat**] **acr**D**,acr**A

$A \epsilon$[0..3], $D \epsilon$[0..3]

$sf \epsilon$[**mul2**|**mul4**|**div2**|**div4**|**div8**|**div16**|**mul65536**]

Operation

1 if *cdt* is true: **acr**$D \leftarrow$saturation(round(scale(**acr**A)))

Description

The postop instruction offers the possibility to perform a scale, round and/or saturation only operation for an accumulator.

Flags

MV	MS	MN	MZ
U	U	U	U

Comment

Example

```
postop   mul2 rnd sat acr0,acr1
postop   rnd acr0,acr1
```

Instruction 6.10: POSTOP — post accumulator operation

MUL*xx* — multiply

Syntax

1 **mul***xx* [*sf*] **acr***D*,**r***a*,**r***b*

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $D\epsilon[0..3]$

sf ϵ[**mul2**|**mul4**|**div2**|**div4**|**div8**|**div16**|**mul65536**]

Operation

1 **acr***D*←scale(signed/unsigned(**r***a*) * signed/unsigned(**r***b*))

Description

Multiply two general registers with chosen signed/unsigned interpretation and scale the result.
The *xx* notation in the instruction shall be replaced with a two character combination of the letters **s** or **u**, and chooses signed or unsigned interpretation respectively for the source registers **r***a* and **r***b*, in that order.

Flags

MV	MS	MN	MZ
U		U	U

Comment

Example

```
muluu    div2 acr0,r2,r4    ;interpret r2 unsigned, r4 unsigned
mulsu    acr1,r7,r23        ;interpret r7 signed, r23 unsigned
```

Instruction 6.11: MUL*xx* — multiply

MULLD0 — multiply and load register from DM0

Syntax

1	**mulld0**	**acr***D***,r***a***,r***b***,r***d***,(ar***c***)**
2	**mulld0**	**acr***D***,r***a***,r***b***,r***d***,(ar***c***++)**
3	**mulld0**	**acr***D***,r***a***,r***b***,r***d***,(--ar***c***)**
4	**mulld0**	**acr***D***,r***a***,r***b***,r***d***,(ar***e***++%)**
5	**mulld0**	**acr***D***,r***a***,r***b***,r***d***,br(ar***c***)**

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $c\epsilon[0..3]$, $d\epsilon[0..31]$, $e\epsilon[0..1]$, $D\epsilon[0..3]$

Operation

1 **acr***D*←signed(**r***a*) * signed(**r***b*) and **r***d*←DM0(**ar***c*)

2 **acr***D*←signed(**r***a*) * signed(**r***b*) and
r*d*←DM0(**ar***c*), if $c\epsilon[0..1]$ then **ar***c*←**ar***c*+**step***c* else **ar***c*←**ar***c*+1

3 **acr***D*←signed(**r***a*) * signed(**r***b*) and
if $c\epsilon[0..1]$ then **ar***c*←**ar***c*-**step***c* else **ar***c*←**ar***c*-1, **r***d*←DM0(**ar***c*)

4 **acr***D*←signed(**r***a*) * signed(**r***b*) and
r*d*←DM0(**ar***e*), if **ar***e*=**top***e* then **ar***e*←**bot***e* else **ar***e*←**ar***e*+**step***e*

5 **acr***D*←signed(**r***a*) * signed(**r***b*) and
r*d*←DM0(bitrev(**ar***c*)), if $c\epsilon[0..1]$ then **ar***c*←**ar***c*+**step***c* else **ar***c*←**ar***c*+1

Description

Multiply two general registers with signed interpretation and mul4 scaling and load a general register with data from DM0 (data memory 0) using various addressing modes.

Flags

MV	MS	MN	MZ
U		U	U

Comment

Operation no.4: observe that **ar***e* will only be set to **bot***e* when the condition **ar***e*=**top***e* is met. There is nothing preventing **ar***e* from becoming larger than **top***e* at the **ar***e*←**ar***e*+**step***e* operation.

Example

```
mulld0   acr0,r0,r1,r20,(ar0++)
```

Instruction 6.12: MULLD0 — multiply and load register from DM0

MULLD1 — multiply and load register from DM1

Syntax

1	**mulld1**	**acr***D*,**r***a*,**r***b*,**r***d*,(**ar***c*)
2	**mulld1**	**acr***D*,**r***a*,**r***b*,**r***d*,(**ar***c*++)
3	**mulld1**	**acr***D*,**r***a*,**r***b*,**r***d*,(--**ar***c*)
4	**mulld1**	**acr***D*,**r***a*,**r***b*,**r***d*,(**ar***e*++%)
5	**mulld1**	**acr***D*,**r***a*,**r***b*,**r***d*,**br**(**ar***c*)

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $c\epsilon[0..3]$, $d\epsilon[0..31]$, $e\epsilon[0..1]$, $D\epsilon[0..3]$

Operation

1 **acr***D*←signed(**r***a*) * signed(**r***b*) and **r***d*←DM1(**ar***c*)

2 **acr***D*←signed(**r***a*) * signed(**r***b*) and
r*d*←DM1(**ar***c*), if $c\epsilon[0..1]$ then **ar***c*←**ar***c*+**step***c* else **ar***c*←**ar***c*+1

3 **acr***D*←signed(**r***a*) * signed(**r***b*) and
if $c\epsilon[0..1]$ then **ar***c*←**ar***c*-**step***c* else **ar***c*←**ar***c*-1, **r***d*←DM1(**ar***c*)

4 **acr***D*←signed(**r***a*) * signed(**r***b*) and
r*d*←DM1(**ar***e*), if **ar***e*=**top***e* then **ar***e*←**bot***e* else **ar***e*←**ar***e*+**step***e*

5 **acr***D*←signed(**r***a*) * signed(**r***b*) and
r*d*←DM1(bitrev(**ar***c*)), if $c\epsilon[0..1]$ then **ar***c*←**ar***c*+**step***c* else **ar***c*←**ar***c*+1

Description

Multiply two general registers with signed interpretation and mul4 scaling and load a general register with data from DM1 (data memory 1) using various addressing modes.

Flags

MV	MS	MN	MZ
U		U	U

Comment

Operation no.4: observe that **ar***e* will only be set to **bot***e* when the condition **ar***e*=**top***e* is met. There is nothing preventing **ar***e* from becoming larger than **top***e* at the **ar***e*←**ar***e*+**step***e* operation.

Example

```
mulld1   acr3,r14,r9,r0,(ar0)
```

Instruction 6.13: MULLD1 — multiply and load register from DM1

MULDBLLD — multiply and double load

Syntax

1 **muldblld** [**br0**] [**br1**] **acr***D* **,r***a* **,r***b* **,r***d* **,r***f*

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $d\epsilon[0..31]$, $f\epsilon[0..31]$, $D\epsilon[0..3]$

Operation

1 **acr***D*←signed(**r***a*) * signed(**r***b*), and
if **br0** is used then **r***d*←DM0(bitrev(ar0)) and ar0←ar0+step0, else **r***d*←DM0(ar0), and
if **br1** is used then **r***f*←DM1(bitrev(ar3)) else **r***f*←DM1(ar3), ar3←ar3+1

Description

Multiply two general registers with signed interpretation and mul4 scaling and load one general register with data from DM0 (data memory 0) and another general register with data from DM1 (data memory 1).

Flags

MV	MS	MN	MZ
U		U	U

Comment

Example

```
muldblld br0 acr0,r15,r5,r22,r23 ;Bit reverse addressing for DM0
                                  ;Incremental addressing for DM1
muldblld acr1,r7,r23,r4,r1        ;Indirect addressing for DM0
                                  ;Incremental addressing for DM1
```

Instruction 6.14: MULDBLLD — multiply and double load

MAC*xx* — multiply and accumulate

Syntax

1 **mac***xx* [*sf*] **acr***D***,r***a***,r***b*

$a \epsilon [0..31]$, $b \epsilon [0..31]$, $D \epsilon [0..3]$

sf ϵ[**mul2**|**mul4**|**div2**|**div4**|**div8**|**div16**|**mul65536**]

Operation

1 **acr***D*←**acr***D* + scale(signed/unsigned(**r***a*) * signed/unsigned(**r***b*))

Description

Multiply two general registers with chosen signed/unsigned interpretation, scale the result and add to the destination accumulator.

The *xx* notation in the instruction shall be replaced with a two character combination of the letters **s** or **u**, and chooses signed or unsigned interpretation respectively for the source registers **r***a* and **r***b*, in that order.

Flags

MV	MS	MN	MZ
U		U	U

Comment

Example

```
macss    div8 acr3,r17,r14  ;interpret r17 signed, r14 signed
macus    acr1,r17,r3        ;interpret r17 unsigned, r3 signed
```

Instruction 6.15: MAC*xx* — multiply and accumulate

MACLD0 — multiply, accumulate and load register from DM0

Syntax

1	**macld0**	**acr**D**,r**a**,r**b**,r**d**,(ar**c**)**
2	**macld0**	**acr**D**,r**a**,r**b**,r**d**,(ar**c**++)**
3	**macld0**	**acr**D**,r**a**,r**b**,r**d**,(--ar**c**)**
4	**macld0**	**acr**D**,r**a**,r**b**,r**d**,(ar**e**++%)**
5	**macld0**	**acr**D**,r**a**,r**b**,r**d**,br(ar**c**)**

$a\epsilon$[0..31], $b\epsilon$[0..31], $c\epsilon$[0..3], $d\epsilon$[0..31], $e\epsilon$[0..1], $D\epsilon$[0..3]

Operation

1 **acr**D←**acr**D + (signed(**r**a) * signed(**r**b)) and **r**d←DM0(**ar**c)

2 **acr**D←**acr**D + (signed(**r**a) * signed(**r**b)) and
rd←DM0(**ar**c), if $c\epsilon$[0..1] then **ar**c←**ar**c+**step**c else **ar**c←**ar**c+1

3 **acr**D←**acr**D + (signed(**r**a) * signed(**r**b)) and
if $c\epsilon$[0..1] then **ar**c←**ar**c-**step**c else **ar**c←**ar**c-1, **r**d←DM0(**ar**c)

4 **acr**D←**acr**D + (signed(**r**a) * signed(**r**b)) and
rd←DM0(**ar**e), if **ar**e=**top**e then **ar**e←**bot**e else **ar**e←**ar**e+**step**e

5 **acr**D←**acr**D + (signed(**r**a) * signed(**r**b)) and
rd←DM0(bitrev(**ar**c)), if $c\epsilon$[0..1] then **ar**c←**ar**c+**step**c else **ar**c←**ar**c+1

Description

Multiply two general registers with signed interpretation and mul4 scaling, add the result to the destination accumulator and load a general register with data from DM0 (data memory 0) using various addressing modes.

Flags

MV	MS	MN	MZ
U		U	U

Comment

Operation no.4: observe that **ar**e will only be set to **bot**e when the condition **ar**e=**top**e is met. There is nothing preventing **ar**e from becoming larger than **top**e at the **ar**e←**ar**e+**step**e operation.

Example

```
macld0   acr3,r14,r9,r0,(ar0++%)
```

Instruction 6.16: MACLD0 — multiply, accumulate and load register from DM0

MACLD1 — multiply, accumulate and load register from DM1

Syntax

1 **macld1** **acr***D*,**r***a*,**r***b*,**r***d*,**(ar***c***)**
2 **macld1** **acr***D*,**r***a*,**r***b*,**r***d*,**(ar***c***++)**
3 **macld1** **acr***D*,**r***a*,**r***b*,**r***d*,**(--ar***c***)**
4 **macld1** **acr***D*,**r***a*,**r***b*,**r***d*,**(ar***e***++%)**
5 **macld1** **acr***D*,**r***a*,**r***b*,**r***d*,**br(ar***c***)**

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $c\epsilon[0..3]$, $d\epsilon[0..31]$, $e\epsilon[0..1]$, $D\epsilon[0..3]$

Operation

1 **acr***D*←**acr***D* + (signed(**r***a*) * signed(**r***b*)) and **r***d*←DM1(**ar***c*)
2 **acr***D*←**acr***D* + (signed(**r***a*) * signed(**r***b*)) and
r*d*←DM1(**ar***c*), if $c\epsilon[0..1]$ then **ar***c*←**ar***c*+**step***c* else **ar***c*←**ar***c*+1
3 **acr***D*←**acr***D* + (signed(**r***a*) * signed(**r***b*)) and
if $c\epsilon[0..1]$ then **ar***c*←**ar***c*-**step***c* else **ar***c*←**ar***c*-1, **r***d*←DM1(**ar***c*)
4 **acr***D*←**acr***D* + (signed(**r***a*) * signed(**r***b*)) and
r*d*←DM1(**ar***e*), if **ar***e*=**top***e* then **ar***e*←**bot***e* else **ar***e*←**ar***e*+**step***e*
5 **acr***D*←**acr***D* + (signed(**r***a*) * signed(**r***b*)) and
r*d*←DM1(bitrev(**ar***c*)), if $c\epsilon[0..1]$ then **ar***c*←**ar***c*+**step***c* else **ar***c*←**ar***c*+1

Description

Multiply two general registers with signed interpretation and mul4 scaling, add the result to the destination accumulator and load a general register with data from DM1 (data memory 1) using various addressing modes.

Flags

MV	MS	MN	MZ
U		U	U

Comment

Operation no.4: observe that **ar***e* will only be set to **bot***e* when the condition **ar***e*=**top***e* is met. There is nothing preventing **ar***e* from becoming larger than **top***e* at the **ar***e*←**ar***e*+**step***e* operation.

Example

```
macld1   acr0,r2,r19,r3,(ar0++)
```

Instruction 6.17: MACLD1 — multiply, accumulate and load register from DM1

MDM*xx* — multiply and diminish

Syntax

1 **mdm***xx* [*sf*] **acr***D*,**r***a*,**r***b*

$a\epsilon[0..31]$, $b\epsilon[0..31]$, $D\epsilon[0..3]$

sf ϵ[**mul2**|**mul4**|**div2**|**div4**|**div8**|**div16**|**mul65536**]

Operation

1 **acr***D*←**acr***D* – scale(signed/unsigned(**r***a*) * signed/unsigned(**r***b*))

Description

Multiply two general registers with chosen signed/unsigned interpretation, scale the result and subtract from the destination accumulator.

The *xx* notation in the instruction shall be replaced with a two character combination of the letters **s** or **u**, and chooses signed or unsigned interpretation respectively for the source registers **r***a* and **r***b*, in that order.

Flags

MV	MS	MN	MZ
U		U	U

Comment

Example

```
mdmss    mul2 acr0,r11,r5   ;interpret r11 signed, r5 signed
mdmuu    acr2,r7,r13        ;interpret r7 unsigned, r13 unsigned
```

Instruction 6.18: MDM*xx* — multiply and diminish

CONV*xx* — convolution

Syntax

1 **conv**xx [**ap**|**am**] [sf] **acr**D,$DM0$,$DM1$

$DM0$ can be one of **(ar**a**++)**, **(--ar**a**)** or **(ar**c**++%)**

$DM1$ can be one of **(ar**b**++)**, **(--ar**b**)** or **(ar**e**++%)**

$a \epsilon [0..3]$, $b \epsilon [0..3]$, $c \epsilon [0..1]$, $e \epsilon [0..1]$, $D \epsilon [0..3]$

$sf \epsilon$[**mul2**|**mul4**|**div2**|**div4**|**div8**|**div16**|**mul65536**]

Operation

Pre operation:

The **(--ar**x**)** addressing mode will: if $x \epsilon [0..1]$ then **ar**$x \leftarrow$**ar**$x-$**step**x else **ar**$x \leftarrow$**ar**$x-1$

1 **acr**$D \leftarrow$**acr**D $+/-$ scale(signed/unsigned(DM0(ar)) * signed/unsigned(DM1(ar)))

Post operation:

The **(ar**x**++)** addressing mode will: if $x \epsilon [0..1]$ then **ar**$x \leftarrow$**ar**$x+$**step**x else **ar**$x \leftarrow$**ar**$x+1$

The **(ar**x**++%)** addressing mode will: if **ar**$x=$**top**x then **ar**$x \leftarrow$**bot**x else **ar**$x \leftarrow$**ar**$x+$**step**x

Description

Multiply two data memory operands with chosen signed/unsigned interpretation, scale the result and add to or subtract from the destination accumulator.

The xx notation in the instruction shall be replaced with a two character combination of the letters **s** or **u**, and chooses signed or unsigned interpretation respectively for the source operands $DM0$ and $DM1$, in that order.

The **ap** operator (default) denotes addition in the convolution operation.

The **am** operator denotes subtraction in the convolution operation.

Flags

MV	MS	MN	MZ
U			

Comment

Example

```
convus   am acr1,(ar2++),(--ar1)
convuu   acr0,(ar0++%),(ar1++%)
```

Instruction 6.19: CONVxx — convolution

7 Iterative instructions

The iterative instruction (**`repeat`**) makes it possible to repeat instructions in sequence a certain number of times. All **`repeat`** does is setup the internal loop registers (**`loopn`**, **`loopb`** and **`loope`**) and then the internal structure of the processor conducts the iteration. Nested iteration loops are not possible without the risk of breaking the flow of outer loops, since inner **`repeat`** instructions may rewrite the loop registers. Table 7.1 lists the iterative instructions.

Mnemonic	Description	Page
repeat	repeat instructions in sequence a certain number of times	84

Table 7.1: iterative instructions

REPEAT — repeat instructions

Syntax

1 **`repeat`** K**,**L

K-pc-1ϵ[0..127], $L\epsilon$[\$0001..\$0FFF]

Operation

1 **`loopn`**$\leftarrow L$, **`loopb`**$\leftarrow pc+1$, **`loope`**$\leftarrow K$-1

pc is the current program counter value

Description

Repeat instructions up to (not including) address K, L times.

K is typically a label that must reside within 127 addresses after the **`repeat`** instruction.

Flags

No flags affected

Comment

A nested **`repeat`** instruction, that is a **`repeat`** within range of a previous **`repeat`**, will if executed rewrite the loop registers (**`loopn`**, **`loopb`** and **`loope`**) and break the outer loop.

Example

```
    repeat   label1,700
    set      r4,$FA72           ;this and the next instruction ...
    move     r1,sr3             ;...will repeat 700 times in sequence
label1
    move     r17,r31            ;same address as label1, but outside loop
```

Instruction 7.1: REPEAT — repeat instructions

8 Flow control instructions

The flow control instructions determine the program flow, in one way or another. Instructions **`jump`**, **`call`** and **`ret`** make use of 0 to 3 delay slots. With 3 delay slots the first 3 instructions following a jump will be executed before the jump takes place. With 2 delay slots the first 2 instructions and 1 **`nop`** (2 **`nop`** for ret) will be executed before the jump takes place, and so on. Table 8.1 lists the flow control instructions.

Mnemonic	Description	Page
jump	jump to address	86
call	call to subroutine	87
ret	return from subroutine	88
reti	return from interrupt	88
nop	no operation, only used for time shimming	90

Table 8.1: flow control instructions

JUMP — jump to address

Syntax

1 **jump**[**.**cdt] [**ds**x] **r**a
2 **jump**[**.**cdt] [**ds**x] K
$a\epsilon$[0..31], $x\epsilon$[0..3], $K\epsilon$[\$0000..\$FFFF]

Operation

1 if *cdt* is true: after x cycles: $pc \leftarrow (\mathbf{r}a)$
2 if *cdt* is true: after x cycles: $pc \leftarrow K$

Description

The **ds**x option denotes a number of cycles (default is 0) to wait before executing the jump. After the jump there are 3–x empty cycles where no instructions are executed.
K is typically a label

Flags

No flags affected

Comment

Example

```
    jump.ne  ds2 label4
    move     r1,sr3           ;this will execute
    set      r2,7             ;this will execute
    move     r12,r3           ;this will NOT execute
label4
    set      r7,3
```

Instruction 8.1: JUMP — jump to address

CALL — call to subroutine

Syntax

1 **call** [**ds**x] **r**a
2 **call** [**ds**x] K
$a\epsilon$[0..31], $x\epsilon$[0..3], $K\epsilon$[$0000..$FFFF]

Operation

Pre operation:
DM1[sp]←pc+1+x, sp←sp+1
1 after x cycles: pc←(**r**a)
2 after x cycles: pc←K

Description

The **ds**x option denotes a number of cycles (default is 0) to wait before executing the call. After the call there are 3–x empty cycles where no instructions are executed.
K is typically a label

Flags

No flags affected

Comment

Example

```
    call    ds2 label7
    move    r1,sr3          ;this will execute
    set     r2,7            ;this will execute
    jump    label4          ;this will execute AFTER return (ret)
label7
    set     r7,3
    ret     ds1
    move    r1,r2           ;this will execute BEFORE return takes place
    set     r11,7           ;this will NOT execute
```

Instruction 8.2: CALL — call to subroutine

RET — return from subroutine

Syntax

1 **ret** **ds**x
$x \epsilon [0..3]$

Operation

Pre operation: $sp \leftarrow sp - 1$
1 after x cycles: $pc \leftarrow \text{DM1}[sp]$

Description

The **ds**x option denotes a number of cycles (default is 0) to wait before executing the return. After the return there are 4–x empty cycles where no instructions are executed.

Flags

No flags affected

Comment

Example

see the **call** instruction example

Instruction 8.3: RET — return from subroutine

RETI — return from interrupt

Syntax

```
1   reti
```

Operation

Pre operation: $sp \leftarrow sp-1$

1 $pc \leftarrow$DM1(sp)

one cycle later: $fl0 \leftarrow$DM1(--sp)

Description

Pops PC and then flags from the stack.

Flags

All flags in $fl0$ are updated.

Comment

Example

```
reti                              ;return from interrupt routine
```

Instruction 8.4: RETI — return from interrupt

NOP — no operation

Syntax

```
nop
```

Operation

$pc \leftarrow pc+1$

Description

No operation

Flags

No flags affected

Comment

Example

```
nop                           ;simple enough
```

Instruction 8.5: NOP — no operation

9 Alias instructions

The alias instructions are instructions with no physical implementation in the processor. They exist only for the benefit of the programmer and make use of special cases of other physical instructions for their own implementation. Table 9.1 lists the alias instructions.

Mnemonic	Description	Page
inc	increase a register by one	92
incn	increase a register by one with no flag change	93
dec	decrease a register by one	94
decn	decrease a register by one with no flag change	95
push	push register to stack	96
pop	pop stack to register	97

Table 9.1: alias instructions

INC — increase register

Syntax

```
1 inc rd
```

Operation

1 $\mathtt{r}d \leftarrow \mathtt{r}d+1$

Description

Add one to a general register

Flags

AV	AC	AN	AZ
U	U	U	U

Comment

This instruction is implemented as **add r*d*,r*d*,1**

Example

```
inc r2
```

Instruction 9.1: INC — increase register

INCN — increase register with no flag change

Syntax

```
incn        rd
```

Operation

$\mathbf{r}d \leftarrow \mathbf{r}d + 1$

Description

Add one to a general register

Flags

No flags affected

Comment

This instruction is implemented as **addn r*d*,r*d*,1**

Example

```
incn    r7
```

Instruction 9.2: INCN — increase register with no flag change

DEC — decrease register

Syntax

```
1   dec        rd
```

Operation

1 $\mathtt{r}d \leftarrow \mathtt{r}d - 1$

Description

Subtract one from a general register

Flags

AV	AC	AN	AZ
U	U	U	U

Comment

This instruction is implemented as **`add rd,rd,-1`**

Example

```
    dec      r12
```

Instruction 9.3: DEC — decrease register

DECN — decrease register with no flag change

Syntax

1 **decn** **r***d*

Operation

1 **r***d*←**r***d*-1

Description

Subtract one from a general register

Flags

No flags affected

Comment

This instruction is implemented as **addn r***d***,r***d***,-1**

Example

```
decn    r17
```

Instruction 9.4: DECN — decrease register with no flag change

PUSH — push register to stack

Syntax

1 **push** **r**a
$a \epsilon [0..31]$

Operation

1 DM1[sp]←**r**a, increment sp

Description

Copy contents of source register to top of stack

Flags

No flags affected

Comment

This instruction is implemeted as **st1 (sp++),r**a

Example

```
push    r30
```

Instruction 9.5: PUSH — push register to stack

POP — pop stack to register

Syntax

1 **pop** **r**d
$d \epsilon$[0..31]

Operation

1 decrement sp, **r**$d \leftarrow$DM1[sp]

Description

Copy contents from top of stack to destination register

Flags

No flags affected

Comment

This instruction is implemeted as **ld1 r**d**,(--sp)**

Example

```
pop     r7
```

Instruction 9.6: POP — pop stack to register

List of instructions

2.1 MOVE — move 16
2.2 SET — set register to constant 17
2.3 LD0 — load register from DM0 18
2.4 LD1 — load register from DM1 19
2.5 ST0 — store register to DM0 20
2.6 ST1 — store register to DM1 21
2.7 DBLLD — double load 22
2.8 DBLST — double store 23
2.9 IN — read IO port to register 24
2.10 OUT — write register to IO port 25
2.11 DBLIN — IO to DM0 and DM1 26
2.12 DBLOUT — IO from DM0 and DM1 27
2.13 OUT0 — out DM0 data to IO port 28
2.14 OUT1 — out DM1 data to IO port 29

3.1 ADD — addition 32
3.2 ADDN — addition with no flag change 33
3.3 ADDC — addition with carry 34
3.4 ADDS — addition with saturation 35
3.5 SUB — subtraction 36
3.6 SUBN — subtraction with no flag change 37
3.7 SUBC — subtraction with carry 38
3.8 SUBS — subtraction with saturation 39
3.9 CMP — compare 40
3.10 MAX — return maximum value 41
3.11 MIN — return minimum value 42
3.12 ABS — return absolute value 43
3.13 EXT — 8 to 16 bit sign extension 44

4.1 AND — logic and 46
4.2 ANDN — logic and with no flag change 47
4.3 OR — logic or 48
4.4 ORN — logic or with no flag change 49
4.5 XOR — logic Xor 50
4.6 XORN — logic xor with no flag change 51

5.1 ASR — arithmetic shift right 54
5.2 ASL — arithmetic shift left 55

5.3 LSR — logic shift right . . . 56
5.4 LSL — logic shift left . . . 57
5.5 ROR — rotate right . . . 58
5.6 ROL — rotate left . . . 59
5.7 RCR — rotate right through carry . . . 60
5.8 RCL — rotate left through carry . . . 61

6.1 ADDL — long addition . . . 64
6.2 SUBL — long subtraction . . . 65
6.3 SUBLST0 — long subtraction and store register to DM0 . . . 66
6.4 SUBLST1 — long subtraction and store register to DM1 . . . 67
6.5 CMPL — long compare . . . 68
6.6 ABSL — long absolute . . . 69
6.7 NEGL — long negation . . . 70
6.8 MOVEL — long move . . . 71
6.9 CLR — clear accumulator . . . 72
6.10 POSTOP — post accumulator operation . . . 73
6.11 MUL*xx* — multiply . . . 74
6.12 MULLD0 — multiply and load register from DM0 . . . 75
6.13 MULLD1 — multiply and load register from DM1 . . . 76
6.14 MULDBLLD — multiply and double load . . . 77
6.15 MAC*xx* — multiply and accumulate . . . 78
6.16 MACLD0 — multiply, accumulate and load register from DM0 . . . 79
6.17 MACLD1 — multiply, accumulate and load register from DM1 . . . 80
6.18 MDM*xx* — multiply and diminish . . . 81
6.19 CONV*xx* — convolution . . . 82

7.1 REPEAT — repeat instructions . . . 84

8.1 JUMP — jump to address . . . 86
8.2 CALL — call to subroutine . . . 87
8.3 RET — return from subroutine . . . 88
8.4 RETI — return from interrupt . . . 89
8.5 NOP — no operation . . . 90

9.1 INC — increase register . . . 92
9.2 INCN — increase register with no flag change . . . 93
9.3 DEC — decrease register . . . 94
9.4 DECN — decrease register with no flag change . . . 95
9.5 PUSH — push register to stack . . . 96
9.6 POP — pop stack to register . . . 97