

Senior Instruction Set Manual

September 4, 2008

Contents

1 Senior hardware description	5
1.1 General	5
1.2 Accumulators	5
1.3 Registers	8
1.4 Coding convention	8
1.5 Constants	11
1.6 Operations	11
1.6.1 Scaling	11
1.6.2 Rounding	11
1.6.3 Saturation	12
1.7 Software stack	12
2 Move-load-store instructions	13
3 Short arithmetic instructions, 16'b	25
4 Short logic instructions, 16'b	39
5 Short shift instructions, 16'b	47
6 Long arithmetic instructions, 32'b	57
7 Iterative instructions	77
8 Flow control instructions	79
9 Alias instructions	85

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 figure 1.1. The pipeline architecture for `conv` is given in the simplified figure 1.2.

The typical pipeline for RISC instructions is $IF(P1) \rightarrow ID(P2) \rightarrow OF(P3) \rightarrow EX(P4)$. 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

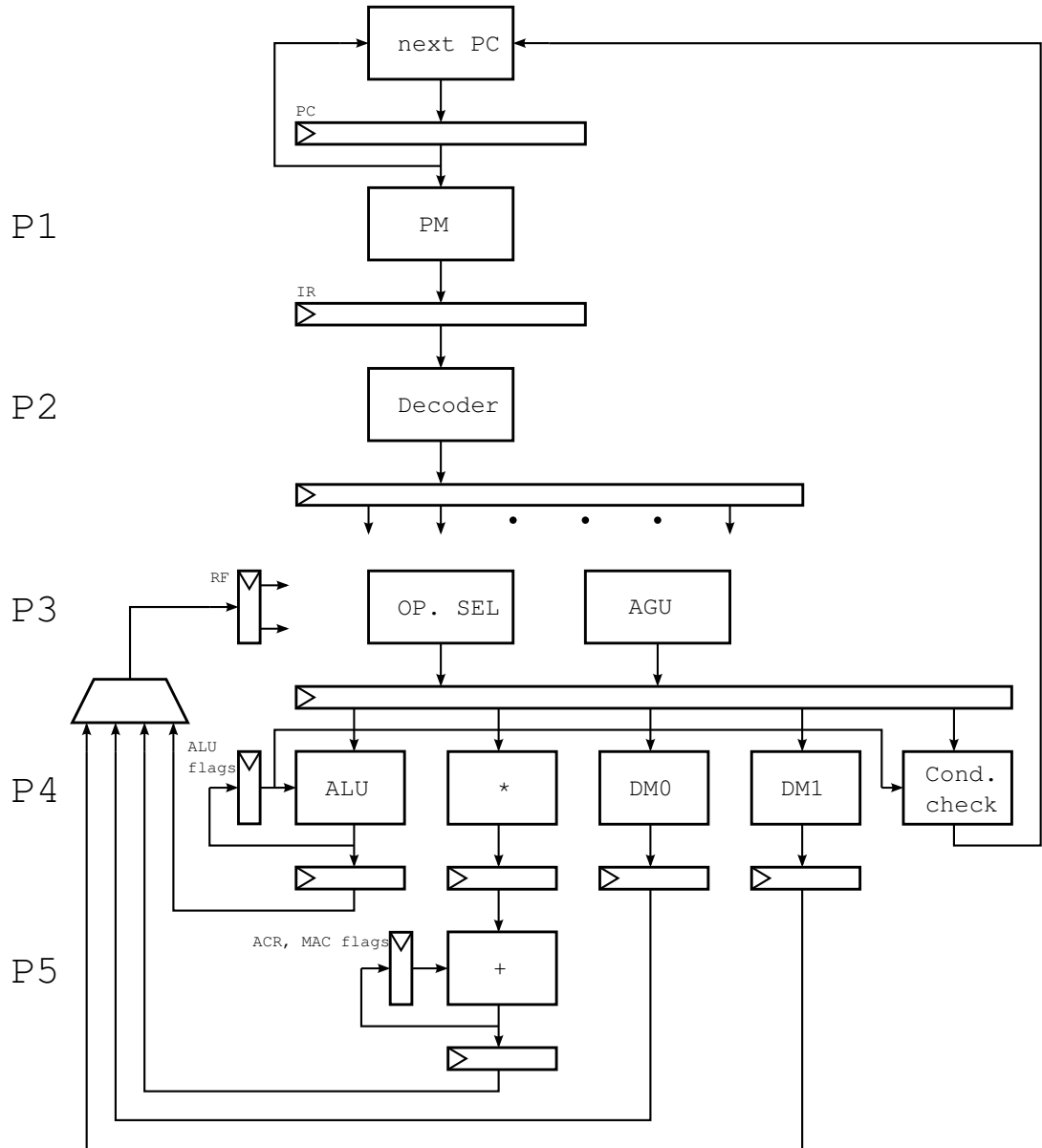


Figure 1.1: Pipeline processor architecture for all instructions but conv

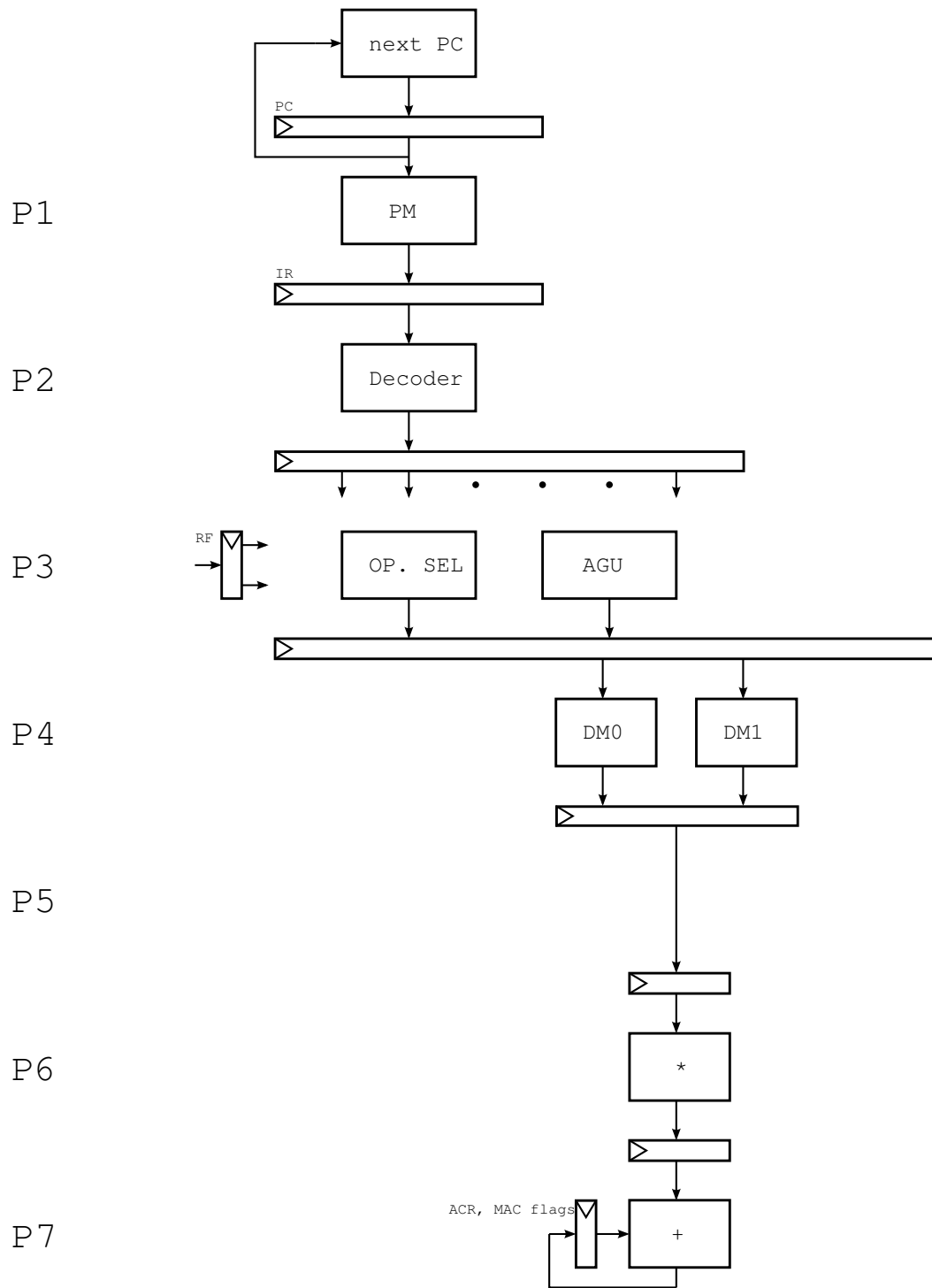


Figure 1.2: Pipeline processor architecture for conv

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
<code>ar0</code>	AG	00000	Address register 0
<code>ar1</code>	AG	00001	Address register 1
<code>ar2</code>	AG	00010	Address register 2
<code>ar3</code>	AG	00011	Address register 3
<code>sp</code>	AG	00100	Stack pointer
<code>bot0</code>	AG	00101	Bottom for AR0
<code>top0</code>	AG	00110	Top for AR0
<code>step0</code>	AG	00111	Step size for AR0
<code>bot1</code>	AG	01000	Bottom for AR1
<code>top1</code>	AG	01001	Top for AR1
<code>step1</code>	AG	01010	Step size for AR1
<code>bitrev</code>	AG	01011	Number of bits to reverse-6
<code>f10</code>	CP	01100	Flags, processor status register
<code>f11</code>	CP	01101	Flags, core control register
<code>loopn</code>	CP	01110	Number of iterations in loop
<code>loopb</code>	CP	01111	Loop start address
<code>loope</code>	CP	10000	Loop end address
<code>intmask</code>	CP	10001	<i>Reserved (interrupt mask)</i>
<code>guards01</code>	MAC	10010	Guard for ACR0 and ACR1
<code>guards23</code>	MAC	10011	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 (`f10`) and core control vector register (`f11`) are in the following table 1.3.

1.4 Coding convention

Instruction are classified into four groups, as seen in table 1.4.

Name	Bit assignment	Specification
AZ	f10[0]	ALU zero flag
AN	f10[1]	ALU sign flag
AC	f10[2]	ALU carry/saturation flag
AV	f10[3]	ALU ALU overflow flag
MZ	f10[4]	MAC zero flag
MN	f10[5]	MAC sign flag
MS	f10[6]	MAC saturation flag
MV	f10[7]	MAC overflow flag (sticky)
TR	f11[]	<i>Reserved(Trace mode)</i>
IE	f11[]	<i>Reserved(Global interrupt enable)</i>

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

Flags of the DSP core expose behaviour of the most recent results from ALU and MAC. Flags are specified in the following 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 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.

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	<i>Unused</i>	
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 \neq AV
slt	01011	ALU signed less than	AN \neq 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=0
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

AM	Code	Addressing model	Coding	Algorithm specification
INDR	000	Reg indirect	Any R0-R31	$A \leftarrow R_n$
INDX	001	Indexed	Any AR0-AR3 or R0-R31	$A \leftarrow AR_n + R_m$
INC	010	Post-add	Any AR0-AR3	$A \leftarrow AR_n; AR_n += ST_n$
DEC	011	Pre-subtract	Any AR0-AR3	$AR_n -= ST_n; A \leftarrow AR_n$
OFS	100	Offset addressing	Any AR0-AR3, offset	$A \leftarrow +$ signed offset
MINC	101	Post add, mod addr	Any AR0-AR3	$A \leftarrow AR_n; AR_n += ST_n$
ABS	110	Absolute	Immediate 16'b	$A \leftarrow \text{abs}(16\text{b immediate})$
BRV	111	Bit reversal	Any AR0-AR3	$A \leftarrow BR(AR_n); AR_n += ST_n$

Table 1.7: Addressing modes

1.5 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

1.6 Operations

1.6.1 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
<code>mul2</code>	001	2	Multiply by 2
<code>mul4</code>	010	4	Multiply by 4
<code>div2</code>	011	0.5	Divide by 2
<code>div4</code>	100	0.25	Divide by 4
<code>div8</code>	101	0.125	Divide by 8
<code>div16</code>	110	0.0625	Divide by 16
<code>mul65536</code>	111	2^{16}	Multiply by 2^{16}

Table 1.8: Scale factors

1.6.2 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
<code>rnd</code>	1	Rounding to nearest integer value

Table 1.9: Rounding

1.6.3 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
<code>sat</code>	1	Saturate to within signed destination operand bitsize range

Table 1.10: Saturation

1.7 Software stack

The software stack is located in `dm1` and grows towards higher addresses. It's 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.

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	14
set	set constant to register	15
ld0	load register from data memory 0	16
ld1	load register from data memory 1	17
st0	store register to data memory 0	18
st1	store register to data memory 0	19
dblld	double load	20
dblst	double store	21
in	read IO port to register	22
out	write register to IO port	23

Table 2.1: move-load-store instructions

MOVE — move

Syntax

```

1  move      rd,sra
2  move      srd,ra
3  move[.cdt] rd,[sat] [rnd] [sf] acrA
4  move      acrD[.h|.l],ra
5  move[.cdt] rd,ra
ae[0..31], de[0..31], Ae[0..3], Dε[0..3]
sfε[mul2|mul4|div2|div4|div8|div16|mul65536]
```

Operation

```

1  rd ← sra
2  srd ← ra
3  if cdt is true: rd ← saturation(round(scale(acrA)))
4  if h is used: acrD[31..16] ← ra, if l is used: acrD[15..0] ← ra
5  if cdt is true: rd ← ra
```

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      rd,K
2  set      srd,K
    $d \in [0..31], K \in [\$0000..\$FFFF]$ 
```

Operation

```
1   $rd \leftarrow K$ 
2   $srd \leftarrow K$ 
```

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      rd, (rb)
2  ld0      rd, (ara, rb)
3  ld0      rd, (arc++)
4  ld0      rd, (--arc)
5  ld0      rd, (ara, K)
6  ld0      rd, (are++)
7  ld0      rd, (L)
8  ld0      rd, br(arc)

```

$a \in [0..3]$, $b \in [0..31]$, $c \in [0..3]$, $d \in [0..31]$, $e \in [0..1]$, $K \in [-4096..4095]$, $L \in [\$0000..\$FFFF]$

Operation

```

1  rd ← DM0(ra)
2  rd ← DM0(ara + rb)
3  rd ← DM0(arc), if ce[0..1] then arc ← arc + stepc else arc ← arc + 1
4  if ce[0..1] then arc ← arc - stepc else arc ← arc - 1, rd ← DM0(arc)
5  rd ← DM0(ara + K)
6  rd ← DM0(are), if are = tope then are ← bote else are ← are + stepe
7  rd ← DM0(L)
8  rd ← DM0(bitrev(arc)), if ce[0..1] then arc ← arc + stepc else arc ← arc + 1

```

Description

Load a general register with data from DM0 (data memory 0) using various addressing modes.

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 are will only be set to bot_e when the condition $are = tope$ is met. There is nothing preventing are from becoming larger than $tope$ at the $are \leftarrow are + step_e$ operation.

Example

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

Instruction 2.3: LD0 — load register from DM0

LD1 — load register from DM1

Syntax

```
1  ld1      rd, (rb)
2  ld1      rd, (ara, rb)
3  ld1      rd, (arc++)
4  ld1      rd, (--arc)
5  ld1      rd, (ara, K)
6  ld1      rd, (are++%)
7  ld1      rd, (L)
8  ld1      rd, br(arc)
9  ld1      rd, (--sp)
10 ld1      rd, (sp, K)
   ae[0..3], be[0..31], ce[0..3], de[0..31], ee[0..1], Ke[-4096..4095], Le[$0000..$FFFF]
```

Operation

```
1  rd ← DM1(ra)
2  rd ← DM1(ara+rb)
3  rd ← DM1(arc), if ce[0..1] then arc ← arc+stepc else arc ← arc+1
4  if ce[0..1] then arc ← arc-stepc else arc ← arc-1, rd ← DM1(arc)
5  rd ← DM1(ara+K)
6  rd ← DM1(are), if are=tope then are ← bote else are ← are+stepe
7  rd ← DM1(L)
8  rd ← DM1(bitrev(arc)), if ce[0..1] then arc ← arc+stepc else arc ← arc+1
9  sp ← sp-1, rd ← DM1(sp)
10 rd ← DM1(sp+K)
```

Description

Load a general register with data from DM1 (data memory 1) using various addressing modes.

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 `are` will only be set to `bote` when the condition `are=tope` is met. There is nothing preventing `are` from becoming larger than `tope` at the `are ← are+stepe` operation.

Example

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

ST0 — store register to DM0

Syntax

```

1  st0      (rd),rb
2  st0      (arf,rd),rb
3  st0      (arg++),rb
4  st0      (--arg),rb
5  st0      (arf,K),rb
6  st0      (arh++%),rb
7  st0      (L),rb
8  st0      br(arg),rb

```

$b \in [0..31]$, $d \in [0..31]$, $f \in [0..3]$, $g \in [0..3]$, $h \in [0..1]$, $K \in [-4096..4095]$, $L \in [\$0000..\$FFFF]$

Operation

```

1  DM0(rd) ← rb
2  DM0(arf+rd) ← rb
3  DM0(arg) ← rb, if g ∈ [0..1] then arg ← arg+stepg else arg ← arg+1
4  if g ∈ [0..1] then arg ← arg-stepg else arg ← arg-1, DM0(arg) ← rb
5  DM0(arf+K) ← rb
6  DM0(arh) ← rb, if arh=toph then arh ← both else arh ← arh+steph
7  DM0(L) ← rb
8  DM0(bitrev(arg)) ← rb, if c ∈ [0..1] then arg ← arg+stepg else arg ← arg+1

```

Description

Store contents of a general register to DM0 (data memory 0) using various addressing modes.

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 arh will only be set to $both$ when the condition $arh=toph$ is met. There is nothing preventing arh from becoming larger than $toph$ at the $arh ← arh+steph$ operation.

Example

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

Instruction 2.5: ST0 — store register to DM0

ST1 — store register to DM1

Syntax

```
1  st1      (rd),rb
2  st1      (arf,rd),rb
3  st1      (arg++),rb
4  st1      (--arg),rb
5  st1      (arf,K),rb
6  st1      (arh++%),rb
7  st1      (L),rb
8  st1      br(arg),rb
9  st1      (sp++),rb
10 st1      (sp,K),rb
    b $\in$ [0..31], d $\in$ [0..31], f $\in$ [0..3], g $\in$ [0..3], h $\in$ [0..1], K $\in$ [-4096..4095], L $\in$ [$0000..$FFFF]
```

Operation

```
1  DM1(rd) $\leftarrow$ rb
2  DM1(arf+rd) $\leftarrow$ rb
3  DM1(arg) $\leftarrow$ rb, if g $\in$ [0..1] then arg $\leftarrow$ arg+stepg else arg $\leftarrow$ arg+1
4  if g $\in$ [0..1] then arg $\leftarrow$ arg-stepg else arg $\leftarrow$ arg-1, DM1(arg) $\leftarrow$ rb
5  DM1(arf+K) $\leftarrow$ rb
6  DM1(arh) $\leftarrow$ rb, if arh=toph then arh $\leftarrow$ both else arh $\leftarrow$ arh+steph
7  DM1(L) $\leftarrow$ rb
8  DM1(bitrev(arg)) $\leftarrow$ rb, if c $\in$ [0..1] then arg $\leftarrow$ arg+stepg else arg $\leftarrow$ arg+1
9  DM1(sp) $\leftarrow$ rb, sp $\leftarrow$ sp+1
10 DM1(sp+K) $\leftarrow$ rb
```

Description

Store contents of a general register to DM1 (data memory 1) using various addressing modes.

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 *arh* will only be set to *both* when the condition *arh=toph* is met. There is nothing preventing *arh* from becoming larger than *toph* at the *arh \leftarrow arh+steph* operation.

Example

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

DBLLD — double load

Syntax

```

1  dblld      rd, (ara), rf, (arb)
2  dblld      rd, (ara++), rf, (arb++)
3  dblld      rd, (--ara), rf, (--arb)
4  dblld      rd, (arc++%), rf, (are++%)
5  dblld      rd, br(ara), rf, br(arb)
   a $\in$ [0..3], b $\in$ [0..3], c $\in$ [0..1], d $\in$ [0..31], e $\in$ [0..1], f $\in$ [0..31]

```

Operation

```

1  rd  $\leftarrow$  DM0(ara) and rf  $\leftarrow$  DM1(arb)
2  rd  $\leftarrow$  DM0(ara), if a $\in$ [0..1] then ara  $\leftarrow$  ara + stepa else ara  $\leftarrow$  ara + 1, and
   rf  $\leftarrow$  DM1(arb), if b $\in$ [0..1] then arb  $\leftarrow$  arb + stepb else arb  $\leftarrow$  arb + 1
3  if a $\in$ [0..1] then ara  $\leftarrow$  ara - stepa else ara  $\leftarrow$  ara - 1, rd  $\leftarrow$  DM0(ara), and
   if b $\in$ [0..1] then arb  $\leftarrow$  arb - stepb else arb  $\leftarrow$  arb - 1, rf  $\leftarrow$  DM1(arb)
4  rd  $\leftarrow$  DM0(arc), if arc = topc then arc  $\leftarrow$  botc else arc  $\leftarrow$  arc + stepc, and
   rf  $\leftarrow$  DM1(are), if are = tope then are  $\leftarrow$  bote else are  $\leftarrow$  are + stepc
5  rd  $\leftarrow$  DM0(bitrev(ara)), if a $\in$ [0..1] then ara  $\leftarrow$  ara + stepa else ara  $\leftarrow$  ara + 1, and
   rf  $\leftarrow$  DM1(bitrev(arb)), if b $\in$ [0..1] then arb  $\leftarrow$  arb + stepb else arb  $\leftarrow$  arb + 1

```

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 addressing mode.

Example

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

Instruction 2.7: DBLLD — double load

DBLST — double store

Syntax

```
1  dblst      (ard),ra,(arf),rb
2  dblst      (ard++),ra,(arf++),rb
3  dblst      (--ard),ra,(--arf),rb
4  dblst      (arg++%),ra,(arh++%),rb
5  dblst      br(ard),ra,br(arf),rb
a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..3], f $\in$ [0..3], g $\in$ [0..1], h $\in$ [0..1]
```

Operation

- 1 DM0(**ard**) \leftarrow ra and DM1(**arf**) \leftarrow rb
- 2 DM0(**ard**) \leftarrow ra, if d \in [0..1] then **ard** \leftarrow **ard**+stepd else **ard** \leftarrow **ard**+1, and DM1(**arf**) \leftarrow rb, if f \in [0..1] then **arf** \leftarrow **arf**+stepf else **arf** \leftarrow **arf**+1
- 3 if d \in [0..1] then **ard** \leftarrow **ard**-stepd else **ard** \leftarrow **ard**-1, DM0(**ard**) \leftarrow ra, and if f \in [0..1] then **arf** \leftarrow **arf**-stepf else **arf** \leftarrow **arf**-1, DM1(**arf**) \leftarrow rb
- 4 DM0(**arg**) \leftarrow ra, if **arg**=topg then **arg** \leftarrow botg else **arg** \leftarrow **arg**+stepg, and DM1(**arh**) \leftarrow rb, if **arh**=toph then **arh** \leftarrow both else **arh** \leftarrow **arh**+steph
- 5 DM0(bitrev(**ard**)) \leftarrow ra, if d \in [0..1] then **ard** \leftarrow **ard**+stepd else **ard** \leftarrow **ard**+1, and DM1(bitrev(**arf**)) \leftarrow rb, if f \in [0..1] then **arf** \leftarrow **arf**+stepf else **arf** \leftarrow **arf**+1

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 addressing 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 $\in$ [0..31], K $\in$ [$0000..$FFFF]
```

Operation

```
1  rd $\leftarrow$ PORT(K)
```

Description

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

Flags

No flags affected

Comment

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 ∈ [0..31], K ∈ [$0000..$FFFF]
```

Operation

```
1  PORT(K) ← ra
```

Description

Write data from a general register to an IO port

Flags

No flags affected

Comment

Example

```
out      $0012, r10
```

Instruction 2.10: OUT — write register 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 on 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	26
addn	addition of registers and constants with no flag change	27
addc	addition of registers and constants with carry	28
adds	addition of registers and constants with saturation	29
sub	subtraction of registers and constants	30
subn	subtraction of registers and constants with no flag change	31
subc	subtraction of registers and constants with carry	32
subs	subtraction of registers and constants with saturation	33
cmp	compare registers and constants	34
max	return maximum of registers and constants	35
min	return minimum of registers and constants	36
abs	return absolute of a register	37

Table 3.1: short arithmetic instructions

ADD — addition

Syntax

```

1  add[.cdt]    rd,ra,rb
2  add         rd,K,rb
3  add         rd,rb,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]

```

Operation

```

1  if cdt is true: rd $\leftarrow$ ra+rb
2,3 rd $\leftarrow$ K+rb

```

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 (*ra* or *rb*) in the syntax, in which case the destination register (*rd*) 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]   rd,ra,rb
2  addn        rd,K,rb
3  addn        rd,rb,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]
```

Operation

```
1  if cdt is true: rd $\leftarrow$ ra+rb
2,3 rd $\leftarrow$ K+rb
```

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 (*ra* or *rb*) in the syntax, in which case the destination register (*rd*) 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]   rd,ra,rb
2  addc        rd,K,rb
3  addc        rd,rb,K
a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]

```

Operation

```

1  if cdt is true: rd $\leftarrow$ ra+rb+AC
2,3 rd $\leftarrow$ K+rb+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 (*ra* or *rb*) in the syntax, in which case the destination register (*rd*) 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]   rd,ra,rb
2  adds        rd,K,rb
3  adds        rd,rb,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]
```

Operation

```
1  if cdt is true: rd $\leftarrow$ saturation(ra+rb)0
2,3 rd $\leftarrow$ saturation(K+rb)
```

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 (*ra* or *rb*) in the syntax, in which case the destination register (*rd*) 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]    rd,ra,rb
2  sub         rd,K,rb
a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]

```

Operation

```

1  if cdt is true: rd $\leftarrow$ ra-rb
2  rd $\leftarrow$ K-rb

```

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 (ra or rb) in the syntax, in which case the destination register (rd) 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]   rd,ra,rb
2  subn         rd,K,rb
 $a \in [0..31]$ ,  $b \in [0..31]$ ,  $d \in [0..31]$ ,  $K \in [-2048..2047]$ 
```

Operation

```
1  if cdt is true:  $rd \leftarrow ra - rb$ 
2   $rd \leftarrow K - rb$ 
```

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 (ra or rb) in the syntax, in which case the destination register (rd) 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]   rd,ra,rb
2  subc        rd,K,rb
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]
```

Operation

```
1  if cdt is true:  $rd \leftarrow ra - rb - 1 + AC = ra + rb \oplus \$FFFF + AC$ 
2   $rd \leftarrow K - rb - 1 + AC = K + rb \oplus \$FFFF + AC$ 
```

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 (ra or rb) in the syntax, in which case the destination register (rd) 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]    rd,ra,rb
2  subs         rd,K,rb
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]
```

Operation

```
1  if cdt is true: rd $\leftarrow$ saturation(ra-rb)
2  rd $\leftarrow$ saturation(K-rb)
```

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 (*ra* or *rb*) in the syntax, in which case the destination register (*rd*) 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]    ra,rb
2  cmp           K,rb
   a∈[0..31], b∈[0..31], K∈[-32768..32767]

```

Operation

```

1  if cdt is true:  $ALUflags \leftarrow ra - rb$ 
2   $ALUflags \leftarrow K - rb$ 

```

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]    rd,ra,rb
2  max         rd,K,ra
3  max         rd,ra,K
a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]
```

Operation

```
1  if cdt is true: rd $\leftarrow$ signed maximum(ra,rb)
2,3 rd $\leftarrow$ signed maximum(ra,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 (*ra* or *rb*) in the syntax, in which case the destination register (*rd*) 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]    rd,ra,rb
2  min         rd,K,ra
3  min         rd,ra,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]
```

Operation

```
1  if cdt is true: rd $\leftarrow$ signed minimum(ra,rb)
2,3 rd $\leftarrow$ signed minimum(ra,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 (*ra* or *rb*) in the syntax, in which case the destination register (*rd*) 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] rd    ra
   a $\in$ [0..31], d $\in$ [0..31]
```

Operation

1 if *cdt* is true: $rd \leftarrow \text{absolute}(ra)$ or $rd \leftarrow \text{absolute}(\text{saturation}(ra))$

Description

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

Flags

No flags affected

Comment

Observe: for the special case when $ra = \$8000$ saturation is needed

Example

```
abs    r12,r13
```

Instruction 3.12: ABS — return absolute value

4 Short logic instructions, 16'b

The short logic instructions concern 16 bit logic operations. All instructions can make use of conditional execution (depending on 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	40
andn	logic and between registers and constants with no flag change	41
or	logic or between registers and constants	42
orn	logic or between registers and constants with no flag change	43
xor	logic xor between registers and constants	44
xorn	logic xor between registers and constants with no flag change	45

Table 4.1: short logic instructions

AND — logic and

Syntax

```

1  and[.cdt]    rd,ra,rb
2  and         rd,K,ra
3  and         rd,ra,K
a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]

```

Operation

```

1  if cdt is true: rd $\leftarrow$ ra•rb
2,3 rd $\leftarrow$ ra•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 (ra or rb) in the syntax, in which case the destination register (rd) 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]   rd,ra,rb
2  andn        rd,K,ra
3  andn        rd,ra,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]
```

Operation

```
1  if cdt is true: rd $\leftarrow$ ra•rb
2,3 rd $\leftarrow$ ra•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 (*ra* or *rb*) in the syntax, in which case the destination register (*rd*) 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]      rd,ra,rb
2  or           rd,K,ra
3  or           rd,ra,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]

```

Operation

```

1  if cdt is true: rd $\leftarrow$ ra $\vee$ rb
2,3 rd $\leftarrow$ ra $\vee$ 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

AV	AC	AN	AZ
U	U	U	U

Comment

It is possible to omit one source register (*ra* or *rb*) in the syntax, in which case the destination register (*rd*) 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]    rd,ra,rb
2  orn          rd,K,ra
3  orn          rd,ra,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]
```

Operation

```
1  if cdt is true: rd $\leftarrow$ ra $\vee$ rb
2,3 rd $\leftarrow$ ra $\vee$ 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 (*ra* or *rb*) in the syntax, in which case the destination register (*rd*) 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]    rd,ra,rb
2  xor         rd,K,ra
3  xor         rd,ra,K
a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]

```

Operation

```

1  if cdt is true: rd $\leftarrow$ ra $\oplus$ rb
2,3 rd $\leftarrow$ ra $\oplus$ 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 (ra or rb) in the syntax, in which case the destination register (rd) 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]    rd,ra,rb
2  xorn          rd,K,ra
3  xorn          rd,ra,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [-2048..2047]
```

Operation

```
1  if cdt is true: rd $\leftarrow$ ra $\oplus$ rb
2,3 rd $\leftarrow$ ra $\oplus$ 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 (*ra* or *rb*) in the syntax, in which case the destination register (*rd*) 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 on 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	48
asl	arithmetic shift left	49
lsr	logic shift right	50
lsl	logic shift left	51
ror	rotate right	52
rol	rotate left	53
rcr	rotate right through carry	54
rcl	rotate left through carry	55

Table 5.1: short shift instructions

ASR — arithmetic shift right

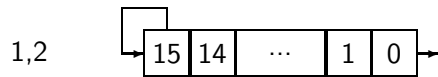
Syntax

```

1  asr[.cdt]    rd,ra,rb
2  asr[.cdt]    rd,ra,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [0..31]

```

Operation



Description

ra is the register to be shifted, rb or K holds the number of steps to shift, the result is stored in register rd

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (ra) in the syntax, in which case the destination register (rd) 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]    rd,ra,rb
2  asl[.cdt]    rd,ra,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [0..31]
```

Operation

1,2 

Description

ra is the register to be shifted, rb or K holds the number of steps to shift, the result is stored in register rd

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (ra) in the syntax, in which case the destination register (rd) will be used in its place.

Example

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

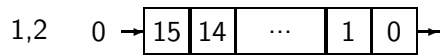
Instruction 5.2: ASL — arithmetic shift left

LSR — logic shift right

Syntax

```
1  lsr[.cdt]    rd,ra,rb
2  lsr[.cdt]    rd,ra,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [0..31]
```

Operation



Description

ra is the register to be shifted, rb or K holds the number of steps to shift, the result is stored in register rd

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (ra) in the syntax, in which case the destination register (rd) will be used in its place.

Example

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

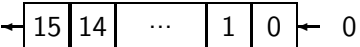
Instruction 5.3: LSR — logic shift right

LSL — logic shift left

Syntax

```
1  lsl[.cdt]    rd,ra,rb
2  lsl[.cdt]    rd,ra,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [0..31]
```

Operation

1,2 

Description

ra is the register to be shifted, rb or K holds the number of steps to shift, the result is stored in register rd

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (ra) in the syntax, in which case the destination register (rd) will be used in its place.

Example

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

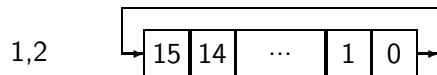
Instruction 5.4: LSL — logic shift left

ROR — rotate right

Syntax

```
1  ror[.cdt]    rd,ra,rb
2  ror[.cdt]    rd,ra,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [0..15]
```

Operation



Description

ra is the register to be rotated, rb or K holds the number of steps to rotate, the result is stored in register rd

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (ra) in the syntax, in which case the destination register (rd) will be used in its place.

Example

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

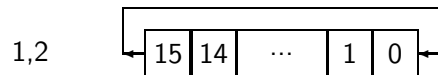
Instruction 5.5: ROR — rotate right

ROL — rotate left

Syntax

```
1  rol[.cdt]    rd,ra,rb
2  rol[.cdt]    rd,ra,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [0..15]
```

Operation



Description

ra is the register to be rotated, rb or K holds the number of steps to rotate, the result is stored in register rd

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (ra) in the syntax, in which case the destination register (rd) 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

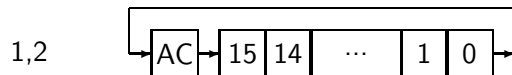
Syntax

```

1  rcr[.cdt]    rd,ra,rb
2  rcr[.cdt]    rd,ra,K
a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [0..15]

```

Operation



Description

ra is the register to be rotated, rb or K holds the number of steps to rotate, the result is stored in register rd

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (ra) in the syntax, in which case the destination register (rd) will be used in its place.

Example

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

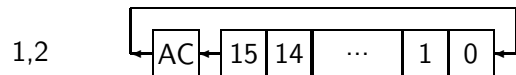
Instruction 5.7: RCR — rotate right through carry

RCL — rotate left through carry

Syntax

```
1  rcl[.cdt]    rd,ra,rb
2  rcl[.cdt]    rd,ra,K
   a $\in$ [0..31], b $\in$ [0..31], d $\in$ [0..31], K $\in$ [0..15]
```

Operation



Description

ra is the register to be rotated, rb or K holds the number of steps to rotate, the result is stored in register rd

Flags

AV	AC	AN	AZ
	U	U	U

Comment

It is possible to omit the source register (ra) in the syntax, in which case the destination register (rd) 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**, **maclld0**, **maclld1**, **mdm** and **conv**) can make use of conditional execution (depending on 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	58
subl	long subtraction between accumulators and registers	59
sublst0	long subtraction and store register to data memory 0	60
sublst1	long subtraction and store register to data memory 1	61
cmpl	long compare between accumulators and registers	62
absl	long absolute of accumulator or register	63
negl	long negation of accumulator or register	64
movel	long move of register to accumulator	65
clr	clear of accumulator	66
postop	post accumulator operation	67
mul	multiply	68
mulld0	multiply and load register from data memory 0	69
mulld1	multiply and load register from data memory 1	70
muldblld	multiply and double load	71
mac	multiply and accumulate	72
maclld0	multiply, accumulate and load register from data memory 0	73
maclld1	multiply, accumulate and load register from data memory 1	74
mdm	multiply and diminish	75
conv	convolution	76

Table 6.1: long arithmetic instructions

ADDL — long addition

Syntax

```

1  addl[.cdt]   acrD,acrA,acrB
2  addl[.cdt]   acrD,acrA,rb:0
3  addl[.cdt]   acrD,acrA,1:rb
4  addl[.cdt]   acrD,acrA,ra:rb
5  addl[.cdt]   [sat] [rnd] rd,acrA,rb:0
6  addl[.cdt]   [sat] [rnd] rd,acrA,1:rb
   ae[0..31], be[0..31], de[0..31], Ae[0..3], Be[0..3], De[0..3]

```

Operation

```

1  if cdt is true:  $acrD \leftarrow acrA + acrB$ 
2  if cdt is true:  $acrD \leftarrow acrA + zero\_extension(rb)$ 
3  if cdt is true:  $acrD \leftarrow acrA + sign\_extension(rb)$ 
4  if cdt is true:  $acrD \leftarrow acrA + register\_extension(rb)$ 
5  if cdt is true:  $rd \leftarrow saturation(round(acrA + zero\_extension(rb)))$ 
6  if cdt is true:  $rd \leftarrow saturation(round(acrA + sign\_extension(rb)))$ 

```

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]   acrD,acrA,acrB
2  subl[. cdt]   acrD,acrA,rb:0
3  subl[. cdt]   acrD,acrA,1:rb
4  subl[. cdt]   acrD,acrA,ra:rb
a∈[0..31], b∈[0..31], A∈[0..3], B∈[0..3], D∈[0..3]

```

Operation

- 1 if *cdt* is true: **acrD**←**acrA**−**acrB**
- 2 if *cdt* is true: **acrD**←**acrA**−zero_extension(**rb**)
- 3 if *cdt* is true: **acrD**←**acrA**−sign_extension(**rb**)
- 4 if *cdt* is true: **acrD**←**acrA**−register_extension(**rb**)

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] rd,ra:0,acrB,(arf++) ,rb
2  sublst0      [sat] [rnd] rd,1:ra,acrB,(arf++) ,rb
a∈[0..31], b∈[0..31], d∈[0..31], f∈[0..3], B∈[0..3]

```

Operation

```

1  rd ← saturation(round(zero_extension(ra) - acrB)) and
    DM0(arf) ← rb, if f∈[0..1] then arf ← arf + stepf else arf ← arf + 1
2  rd ← saturation(round(sign_extension(ra) - acrB)) and
    DM0(arf) ← rb, if f∈[0..1] then arf ← arf + stepf else arf ← arf + 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] rd,ra:0,acrB,(arf++),rb
2  sublst1      [sat] [rnd] rd,1:ra,acrB,(arf++),rb
a∈[0..31], b∈[0..31], d∈[0..31], f∈[0..3], B∈[0..3]

```

Operation

```

1  rd←saturation(round(zero_extension(ra)-acrB)) and
   DM1(arf)←rb, if f∈[0..1] then arf←arf+stepf else arf←arf+1
2  rd←saturation(round(sign_extension(ra)-acrB)) and
   DM1(arf)←rb, if f∈[0..1] then arf←arf+stepf else arf←arf+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]   acrA, acrB
2  cmpl[. cdt]   acrA, r7:0
3  cmpl[. cdt]   acrA, 1:r7
4  cmpl[. cdt]   acrA, ra:rb
a ∈ [0..31], b ∈ [0..31], A ∈ [0..3], B ∈ [0..3]

```

Operation

- 1 if *cdt* is true: $\text{MACflags} \wedge \text{saturation}(\mathbf{acrA} - \mathbf{acrB})$
- 2 if *cdt* is true: $\text{MACflags} \wedge \text{saturation}(\mathbf{acrA} - \text{zero_extension}(\mathbf{rb}))$
- 3 if *cdt* is true: $\text{MACflags} \wedge \text{saturation}(\mathbf{acrA} - \text{sign_extension}(\mathbf{rb}))$
- 4 if *cdt* is true: $\text{MACflags} \wedge \text{saturation}(\mathbf{acrA} - \text{register_extension}(\mathbf{rb}))$

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  abs1[.cdt]    [sat] acrD,acrA
2  abs1[.cdt]    [sat] acrD[.h|.l],ra
   a $\in$ [0..31], A $\in$ [0..3], D $\in$ [0..3]
```

Operation

- 1 if *cdt* is true: $\mathbf{acrD} \leftarrow \text{saturation}(\text{absolute}(\mathbf{acrA}))$
- 2 if *cdt* is true; if *h* is used $\mathbf{acrD} \leftarrow \text{saturation}(\text{absolute}(\mathbf{ra}:[0..0]))$ else $\mathbf{acrD} \leftarrow \text{saturation}(\text{absolute}([0..0]:\mathbf{ra}))$

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 *ra* in *acrD*. 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

```
abs1.mle sat acr2,acr3
abs1     acr0.h,r4
```

Instruction 6.6: ABSL — long absolute

NEGL — long negation

Syntax

```

1  negl[.cdt]    [sat] acrD,acrA
2  negl[.cdt]    [sat] acrD[.h|.l],ra
   a $\in$ [0..31], A $\in$ [0..3], D $\in$ [0..3]

```

Operation

```

1  if cdt is true: acrD ← sat(neg(acrA))
2  if cdt is true; if h is used acrD ← sat(neg(ra:[0..0])) else acrD ← sat(neg([0..0]:ra))

```

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 **ra** in **acrD**. 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]  acrD,ra:rb
2  movel[.cdt]  acrD[.h|.1],ra
   a $\in$ [0..31], b $\in$ [0..31], D $\in$ [0..3]
```

Operation

```
1  if cdt is true:  $acrD \leftarrow ra:rb$ 
2  if cdt is true: if h is used  $acrD \leftarrow sat(ra:[0..0])$  else  $acrD \leftarrow sat([0..0]:ra)$ 
```

Description

The [.h|.1] 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 **1** (default) as destination for **ra** in **acrD**. When using **1**, 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]    acrD  
    $D \in [0..3]$ 
```

Operation

```
1  if cdt is true: acrD ← 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] acrD,acrA
   A $\in$ [0..3], D $\in$ [0..3]
   sf $\in$ [mul2|mul4|div2|div4|div8|div16|mul65536]
```

Operation

```
1  if cdt is true: acrD $\leftarrow$ saturation(round(scale(acrA)))
```

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  mulxx          [sf] acrD,ra,rb
   a∈[0..31], b∈[0..31], D∈[0..3]
   sf∈[mul2|mul4|div2|div4|div8|div16|mul65536]
```

Operation

```
1  acrD ← scale(signed/unsigned(ra) * signed/unsigned(rb))
```

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 *ra* and *rb*, 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      acrD,ra,rb,rd,(arc)
2  mulld0      acrD,ra,rb,rd,(arc++)
3  mulld0      acrD,ra,rb,rd,(--arc)
4  mulld0      acrD,ra,rb,rd,(are++)
5  mulld0      acrD,ra,rb,rd,br(arc)
ae[0..31], be[0..31], ce[0..3], de[0..31], ee[0..1], De[0..3]

```

Operation

```

1  acrD ← signed(ra) * signed(rb) and rd ← DM0(arc)
2  acrD ← signed(ra) * signed(rb) and
   rd ← DM0(arc), if ce[0..1] then arc ← arc + stepc else arc ← arc + 1
3  acrD ← signed(ra) * signed(rb) and
   if ce[0..1] then arc ← arc - stepc else arc ← arc - 1, rd ← DM0(arc)
4  acrD ← signed(ra) * signed(rb) and
   rd ← DM0(are), if are = tope then are ← bot e else are ← are + stepe
5  acrD ← signed(ra) * signed(rb) and
   rd ← DM0(bitrev(arc)), if ce[0..1] then arc ← arc + stepc else arc ← arc + 1

```

Description

Multiply two general registers with signed interpretation 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 `are` will only be set to `bot e` when the condition `are = tope` is met. There is nothing preventing `are` from becoming larger than `tope` at the `are ← are + stepe` 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    acrD,ra,rb,rd,(arc)
2  mulld1    acrD,ra,rb,rd,(arc++)
3  mulld1    acrD,ra,rb,rd,(--arc)
4  mulld1    acrD,ra,rb,rd,(are++)
5  mulld1    acrD,ra,rb,rd,br(arc)
ae[0..31], be[0..31], ce[0..3], de[0..31], ee[0..1], De[0..3]

```

Operation

```

1  acrD ← signed(ra) * signed(rb) and rd ← DM1(arc)
2  acrD ← signed(ra) * signed(rb) and
   rd ← DM1(arc), if ce[0..1] then arc ← arc + stepc else arc ← arc + 1
3  acrD ← signed(ra) * signed(rb) and
   if ce[0..1] then arc ← arc - stepc else arc ← arc - 1, rd ← DM1(arc)
4  acrD ← signed(ra) * signed(rb) and
   rd ← DM1(are), if are = tope then are ← bot e else are ← are + stepe
5  acrD ← signed(ra) * signed(rb) and
   rd ← DM1(bitrev(arc)), if ce[0..1] then arc ← arc + stepc else arc ← arc + 1

```

Description

Multiply two general registers with signed interpretation 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 **are** will only be set to **bot e** when the condition **are = tope** is met. There is nothing preventing **are** from becoming larger than **tope** at the **are ← are + stepe** 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 muldbllld    [br0] [br1] acrD,ra,rb,rd,rf
   ae[0..31], be[0..31], de[0..31], fe[0..31], De[0..3]
```

Operation

```
1  acrD ← signed(ra) * signed(rb), and
   if br0 is used then rd ← DM0(bitrev(ar0)) else rd ← DM0(ar0), ar0 ← ar0 + step0, and
   if br1 is used then rf ← DM1(bitrev(ar1)) else rf ← DM1(ar1), ar1 ← ar1 + step1
```

Description

Multiply two general registers with signed interpretation 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

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

Instruction 6.14: MULDBLLD — multiply and double load

MAC_{xx} — multiply and accumulate

Syntax

```
1  macxx          [sf] acrD,ra,rb
   a∈[0..31], b∈[0..31], D∈[0..3]
   sf∈[mul2|mul4|div2|div4|div8|div16|mul65536]
```

Operation

```
1  acrD ← acrD + scale(signed/unsigned(ra) * signed/unsigned(rb))
```

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 **ra** and **rb**, 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      acrD,ra,rb,rd,(arc)
2  macld0      acrD,ra,rb,rd,(arc++)
3  macld0      acrD,ra,rb,rd,(--arc)
4  macld0      acrD,ra,rb,rd,(are++)
5  macld0      acrD,ra,rb,rd,br(arc)
ae[0..31], be[0..31], ce[0..3], de[0..31], ee[0..1], De[0..3]

```

Operation

```

1  acrD ← acrD + (signed(ra) * signed(rb)) and rd ← DM0(arc)
2  acrD ← acrD + (signed(ra) * signed(rb)) and
   rd ← DM0(arc), if ce[0..1] then arc ← arc + stepc else arc ← arc + 1
3  acrD ← acrD + (signed(ra) * signed(rb)) and
   if ce[0..1] then arc ← arc - stepc else arc ← arc - 1, rd ← DM0(arc)
4  acrD ← acrD + (signed(ra) * signed(rb)) and
   rd ← DM0(are), if are = tope then are ← bote else are ← are + stepe
5  acrD ← acrD + (signed(ra) * signed(rb)) and
   rd ← DM0(bitrev(arc)), if ce[0..1] then arc ← arc + stepc else arc ← arc + 1

```

Description

Multiply two general registers with signed interpretation, 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 `are` will only be set to `bote` when the condition `are=tope` is met. There is nothing preventing `are` from becoming larger than `tope` at the `are ← are + stepe` 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      acrD,ra,rb,rd,(arc)
2  macld1      acrD,ra,rb,rd,(arc++)
3  macld1      acrD,ra,rb,rd,(--arc)
4  macld1      acrD,ra,rb,rd,(are++)
5  macld1      acrD,ra,rb,rd,br(arc)
   ae[0..31], be[0..31], ce[0..3], de[0..31], ee[0..1], De[0..3]

```

Operation

```

1  acrD ← acrD + (signed(ra) * signed(rb)) and rd ← DM1(arc)
2  acrD ← acrD + (signed(ra) * signed(rb)) and
   rd ← DM1(arc), if ce[0..1] then arc ← arc + stepc else arc ← arc + 1
3  acrD ← acrD + (signed(ra) * signed(rb)) and
   if ce[0..1] then arc ← arc - stepc else arc ← arc - 1, rd ← DM1(arc)
4  acrD ← acrD + (signed(ra) * signed(rb)) and
   rd ← DM1(are), if are = tope then are ← bote else are ← are + stepe
5  acrD ← acrD + (signed(ra) * signed(rb)) and
   rd ← DM1(bitrev(arc)), if ce[0..1] then arc ← arc + stepc else arc ← arc + 1

```

Description

Multiply two general registers with signed interpretation, 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 **are** will only be set to **bote** when the condition **are=tope** is met. There is nothing preventing **are** from becoming larger than **tope** at the **are←are+stepe** 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  mdmxx          [sf] acrD,ra,rb
   a∈[0..31], b∈[0..31], D∈[0..3]
   sf∈[mul2|mul4|div2|div4|div8|div16|mul65536]
```

Operation

```
1  acrD ← acrD − scale(signed/unsigned(ra) * signed/unsigned(rb))
```

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  convxx      [ap|am] [sf] acrD,DM0,DM1
DM0 can be one of (ara++), (--ara) or (arc++%)
DM1 can be one of (arb++), (--arb) or (are++%)
a∈[0..3], b∈[0..3], c∈[0..1], e∈[0..1], D∈[0..3]
sf∈[mul2|mul4|div2|div4|div8|div16|mul65536]
```

Operation

Pre operation:

The (`--arx`) addressing mode will: if $x \in [0..1]$ then $\mathbf{arx} \leftarrow \mathbf{arx} - \mathbf{stepx}$ else $\mathbf{arx} \leftarrow \mathbf{arx} - 1$

```
1  acrD ← acrD +/- scale(signed/unsigned(DM0(ar)) * signed/unsigned(DM1(ar)))
```

Post operation:

The (`arx++`) addressing mode will: if $x \in [0..1]$ then $\mathbf{arx} \leftarrow \mathbf{arx} + \mathbf{stepx}$ else $\mathbf{arx} \leftarrow \mathbf{arx} + 1$

The (`arx++%`) addressing mode will: if $\mathbf{arx} = \mathbf{topx}$ then $\mathbf{arx} \leftarrow \mathbf{botx}$ else $\mathbf{arx} \leftarrow \mathbf{arx} + \mathbf{stepx}$

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

No flags affected

Comment

Example

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

Instruction 6.19: CONV_{xx} — 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	78

Table 7.1: iterative instructions

REPEAT — repeat instructions

Syntax

```
1  repeat      K,L
   K-pc-1 ∈ [0..127], L ∈ [$0001..$0FFF]
```

Operation

```
1  loopn ← L, loopb ← pc + 1, loope ← L + pc
   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	80
call	call to subroutine	81
ret	return from subroutine	82
nop	no operation, only used for time shimming	83

Table 8.1: flow control instructions

JUMP — jump to address

Syntax

```

1  jump[.cdt]    [dsx] ra
2  jump[.cdt]    [dsx] K
   a $\in$ [0..31], x $\in$ [0..3], K $\in$ [$0000..$FFFF]

```

Operation

```

1  if cdt is true: after x cycles: pc $\leftarrow$ (ra)
2  if cdt is true: after x cycles: pc $\leftarrow$ K

```

Description

The `dsx` 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      [dsx] ra
2  call      [dsx] K
   a $\in$ [0..31], x $\in$ [0..3], K $\in$ [$0000..$FFFF]
```

Operation

Pre operation:

DM1[sp] \leftarrow pc+1+x, sp \leftarrow sp+1

1 after x cycles: pc \leftarrow (ra)

2 after x cycles: pc \leftarrow K

Description

The dsx 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          dsx  
   x $\in$ [0..3]
```

Operation

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

Description

The **dsx** 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

NOP — no operation

Syntax

1 `nop`

Operation

1 $pc \leftarrow pc + 1$

Description

No operation

Flags

No flags affected

Comment

Example

```
    nop                ;simple enough
```

Instruction 8.4: 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	86
incn	increase a register by one with no flag change	87
dec	decrease a register by one	88
decn	decrease a register by one with no flag change	89
push	push register to stack	90
pop	pop stack to register	91

Table 9.1: alias instructions

INC — increase register

Syntax

```
1  inc      rd
```

Operation

```
1  rd ← rd + 1
```

Description

Add one to a general register

Flags

AV	AC	AN	AZ
U	U	U	U

Comment

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

Example

```
inc      r2
```

Instruction 9.1: INC — increase register

INCN — increase register with no flag change

Syntax

```
1  incn      rd
```

Operation

```
1  rd ← rd + 1
```

Description

Add one to a general register

Flags

No flags affected

Comment

This instruction is implemented as `addn rd,rd,1`

Example

```
incn    r7
```

Instruction 9.2: INCN — increase register with no flag change

DEC — decrease register

Syntax

```
1  dec      rd
```

Operation

```
1  rd ← rd - 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      rd
```

Operation

```
1  rd ← rd - 1
```

Description

Subtract one from a general register

Flags

No flags affected

Comment

This instruction is implemented as `addn rd,rd,-1`

Example

```
decn      r17
```

Instruction 9.4: DECN — decrease register with no flag change

PUSH — push register to stack

Syntax

```
1  push      ra
   a $\in$ [0..31]
```

Operation

```
1  DM1[sp] $\leftarrow$ ra, increment sp
```

Description

Copy contents of source register to top of stack

Flags

No flags affected

Comment

This instruction is implemented as `st1 (sp++),ra`

Example

```
push    r30
```

Instruction 9.5: PUSH — push register to stack

POP — pop stack to register

Syntax

```
1  pop          rd  
   d $\in$ [0..31]
```

Operation

```
1  decrement sp, rd $\leftarrow$ DM1[sp]
```

Description

Copy contents from top of stack to destination register

Flags

No flags affected

Comment

This instruction is implemented as `ld1 rd, (--sp)`

Example

```
pop          r7
```

Instruction 9.6: POP — pop stack to register

List of instructions

2.1	MOVE — move	14
2.2	SET — set register to constant	15
2.3	LD0 — load register from DM0	16
2.4	LD1 — load register from DM1	17
2.5	ST0 — store register to DM0	18
2.6	ST1 — store register to DM1	19
2.7	DBLLD — double load	20
2.8	DBLST — double store	21
2.9	IN — read IO port to register	22
2.10	OUT — write register to IO port	23
3.1	ADD — addition	26
3.2	ADDN — addition with no flag change	27
3.3	ADDC — addition with carry	28
3.4	ADDS — addition with saturation	29
3.5	SUB — subtraction	30
3.6	SUBN — subtraction with no flag change	31
3.7	SUBC — subtraction with carry	32
3.8	SUBS — subtraction with saturation	33
3.9	CMP — compare	34
3.10	MAX — return maximum value	35
3.11	MIN — return minimum value	36
3.12	ABS — return absolute value	37
4.1	AND — logic and	40
4.2	ANDN — logic and with no flag change	41
4.3	OR — logic or	42
4.4	ORN — logic or with no flag change	43
4.5	XOR — logic Xor	44
4.6	XORN — logic xor with no flag change	45
5.1	ASR — arithmetic shift right	48
5.2	ASL — arithmetic shift left	49
5.3	LSR — logic shift right	50
5.4	LSL — logic shift left	51
5.5	ROR — rotate right	52
5.6	ROL — rotate left	53
5.7	RCR — rotate right through carry	54

5.8	RCL — rotate left through carry	55
6.1	ADDL — long addition	58
6.2	SUBL — long subtraction	59
6.3	SUBLST0 — long subtraction and store register to DM0	60
6.4	SUBLST1 — long subtraction and store register to DM1	61
6.5	CMPL — long compare	62
6.6	ABSL — long absolute	63
6.7	NEGL — long negation	64
6.8	MOVEL — long move	65
6.9	CLR — clear accumulator	66
6.10	POSTOP — post accumulator operation	67
6.11	MUL xx — multiply	68
6.12	MULLD0 — multiply and load register from DM0	69
6.13	MULLD1 — multiply and load register from DM1	70
6.14	MULDBLLD — multiply and double load	71
6.15	MAC xx — multiply and accumulate	72
6.16	MACLD0 — multiply, accumulate and load register from DM0	73
6.17	MACLD1 — multiply, accumulate and load register from DM1	74
6.18	MDM xx — multiply and diminish	75
6.19	CONV xx — convolution	76
7.1	REPEAT — repeat instructions	78
8.1	JUMP — jump to address	80
8.2	CALL — call to subroutine	81
8.3	RET — return from subroutine	82
8.4	NOP — no operation	83
9.1	INC — increase register	86
9.2	INCN — increase register with no flag change	87
9.3	DEC — decrease register	88
9.4	DECN — decrease register with no flag change	89
9.5	PUSH — push register to stack	90
9.6	POP — pop stack to register	91