VSDSP4 USER'S MANUAL

Revision History			
Rev.	**Date**	**Author**	**Description**
4.2	2008-03-14	PO	Cleaned up version for software developers

© 1998-2008 VLSI Solution Oy, Hermiankatu 8 B, Entrance G, 2nd floor, FIN-33720 Tampere, Finland

Information furnished by VLSI Solution Oy is believed to be accurate and reliable. However, no responsibility is assumed by VLSI Solution Oy for its use.

Specifications are subject to change without notice.

All rights reserved. No part of this manual may be reproduced, in any form or by any means, without permission in writing from the copyright owner.

The descriptions contained herein do not imply the granting of license to make, use, or sell equipment constructed in accordance therewith.

Contents

1 Introduction 1

1.1 VS_DSP Development System 2

1.2 VSDSP^4 compared to VSDSP^2 3

2 Programming Model 4

2.1 Datapath 5

2.2 ALU 6

2.3 Multiplier 6

2.4 Barrel Shifter 6

2.5 Guard Bit Registers 7

2.6 Flags and Mode Bits 8

2.6.1 Saturation (S) 8

2.6.2 Integer (I) 8

2.6.3 Rounding (R) 8

2.6.4 Loop (L) 9

2.6.5 Zero (Z) 9

2.6.6 Negative (N) 9

2.6.7 Overflow (V) 9

2.6.8 Extension (E) 9

2.6.9 Carry (C) 9

3 Data Address Generator 10

3.1 Post-modification Modes 10

3.1.1 Linear Post-Increment/Decrement 11

3.1.2 Modulo Post-Increment/Decrement 11

3.1.3 Bit Reversal 12

4 Program control **13**

4.1 PC . . . 13

4.2 LR0 . . . 13

4.3 LR1 . . . 14

4.4 MR0 . . . 14

4.5 IPR0, IPR1 . . . 14

4.6 LS, LE, LC . . . 15

5 Control Flow **16**

5.1 Jumps . . . 16

5.2 Loops . . . 16

5.3 System Reset . . . 17

5.4 Interrupts . . . 17

5.4.1 Interrupt Routines . . . 18

5.5 Halt . . . 19

6 Instruction Set Reference **20**

6.1 List of Instructions . . . 20

6.2 Instruction Descriptions . . . 21

6.3 Instruction Sequence Restrictions . . . 39

6.3.1 Loop Register Restrictions . . . 39

6.3.2 Conditional Jump Restrictions . . . 40

7 Instruction Coding **41**

7.1 General Instruction Composition . . . 41

7.2 Opcode Field . . . 41

7.3 Control Instructions . . . 41

7.4 Arithmetic Operands . . . 44

7.5 Move Encoding 47
7.6 Addressing Modes 48
7.7 Constant Loading 50

8 Software Examples 52

8.1 Single-Precision FIR Transversal Filter 52
8.2 Double-Precision FIR Transversal Filter 53
8.3 Cascaded Biquad IIR Filter 55
8.4 Single-Precision Matrix Multiply 56
8.5 Floating-Point Multiplication and Addition 57

List of Figures

1 VS_DSP General Architecture. 1

2 Processor programming model 4

3 VS_DSP datapath. 5

List of Tables

1 Jump conditions. 26
2 Operation Codes 42
3 Control Instructions 42
4 ALU operand encoding. 44
5 ALU result coding 44
6 Mul operand. 45
7 Mul mode. 45
8 Single operand ALU instructions. 46
9 Registers in short move. 48
10 Registers in full move. 49
11 Load/Store coding. 49
12 Addressing Modes. 49
13 Modifications by the $\overline{\texttt{In}}$ register. 50
14 Addressing mode summary. 51

1 Introduction

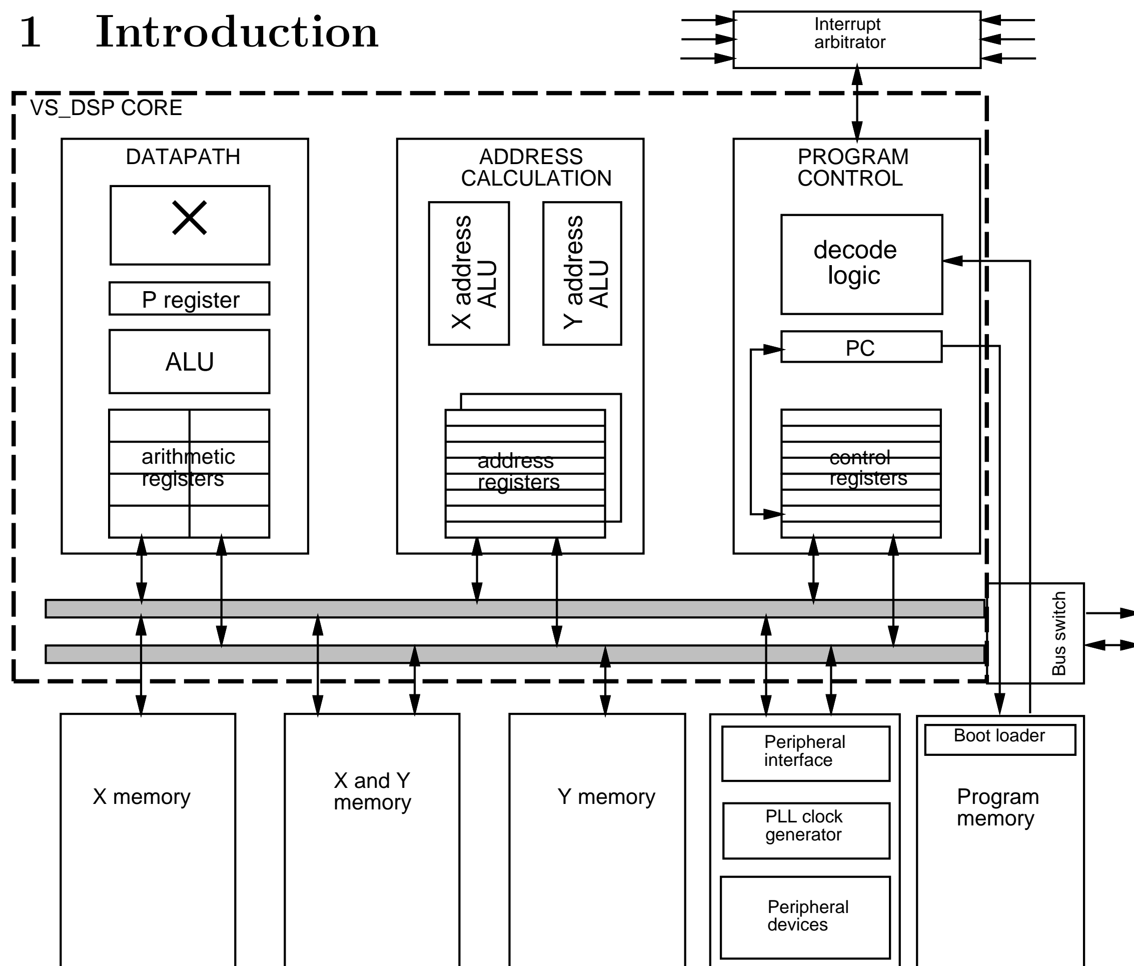


Figure 1: VS_DSP General Architecture.

VSDSP^4 consists of these units:

- *Datapath* — an arithmetic/logic unit (ALU) and a multiplier unit. VSDSP^4 also contains a barrel shifter.
- *Data Address Calculation* — Two dedicated address calculation units provide addresses for simultaneous operations on X and Y memory buses.
- *Program Control* — Instruction fetch, instruction address generation, and instruction decode. The program control also includes harware loop control.
- *Buses* – Internal buses transfer data between different units and memories.

There are also other subsystems that are not part of the core.

- *Memory* — Internal RAM and ROM.
- *Peripherals* — Memory-mapped peripherals, such as interrupt arbiter, serial port, GPIO, timers, DA and/or AD converters.
- *External Bus Switch* — Some chips have external memory buses.
- *Clock Generator* — A phase-locked loop (PLL) can generate core clock.

1.1 VS_DSP Development System

VS_DSP is supported by a comprehensive set of software and hardware for core evaluation and application system development. The VS_DSP Evaluation Kit consists of the VS_DSP Software Development Toolkit (VSKIT) and the Development Board.

VSKIT includes:

- VSA *Assembler* — The Assembler assembles the source code and data modules, and enables, e.g., macros and include files to be used. The Assembler adapts to the parameter values given in Configuration Files.
- VSLINK *Linker* — The Linker links separately assembled modules.
- VSAR *Archiver* — The Archiver enables a function library to be built by the user.
- *Configuration Files* — The Configuration Files describe the system. There is a configuration file to declare the parameter values of the core, and another file for allocating memory and mapping peripherals to the memory space.
- VSSIM / VSS *Instruction Set Simulator* — The Instruction Set Simulator (ISS) reads lod- or coff-format object files generated by the Linker and performs an interactive, instruction-level simulation. The ISS uses the Configuration Files to create a correct model of the core and its surroundings. The features include disassembly, breakpoints, memory and register watch, profiling, dumping and undumping of the state (save and resume), file i/o, and generation of test vectors to be used for hardware verification.
- VSEMU / VS3EMU *Emulator User Interface* — The Emulator User Interface looks like the ISS, but it connects to the Development Board for program execution instead of using the simulator engine.
- VCC *C Compiler* — The C Compiler reads ANSI C based source code (interleaved with some optimization constructs) and produces VS_DSP code.

All software included in the VSKIT is documented in a separate manual called "VS_DSP Software Tools User's Manual". For further information, please refer to that manual.

1.2 VSDSP^4 compared to VSDSP^2

VSDSP^4 has some improvements over the VSDSP^2 core.

- X and Y flags removed
- MR1 removed (interrrupts can save MR0 without changing flags)
- ASHL − Single-cycle arithmetic multi-bit shift
- EXP − Count leading bits
- SAT − Saturate 40-bit ALU register to 32 bits
- RND − Round and saturate 40-bit ALU register to 16 bits
  rounding mode bit: 1 = convergent 0, 0 = round towards 0
- Modulo addressing allows address modified by
  - $-128 \ldots +127$ when buffer size is a multiple of 64 upto 4096 words
  - $-64 \ldots +63$ when buffer size is $1 \ldots 64$

  Supports the old ±1 modulo mode, but not the old ±2 mode.
- Bit-reverse addressing can count backward as well as forward
- STI and LDI instructions write and read internal instruction RAM
  → IRAM does not need to be mapped to X or Y data spaces
- Instruction address bus timing now similar to XAB and YAB timing
  → Identical timing requirements for all memories

2 Programming Model

The processor programming model is shown in Fig. 2. The processor contains arithmetic, address and control registers.

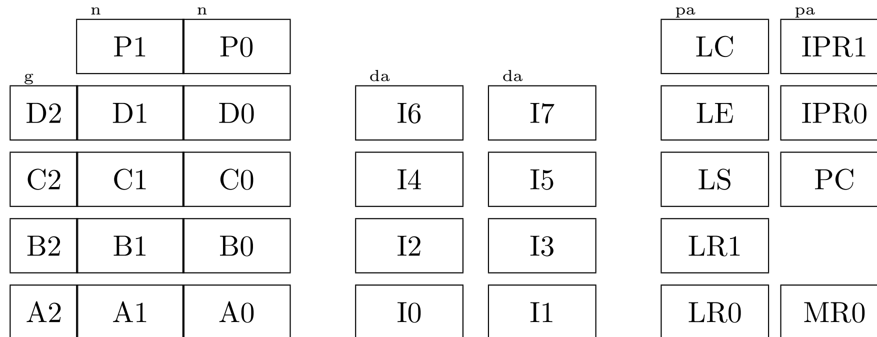


Figure 2: Processor programming model

Arithmetic registers are the 16-bit registers `A0, A1, B0, B1, C0, C1, D0, D1` and the 8-bit guard bit registers `A2, B2, C2, D2`. The multiplier pipeline register `P0, P1` is also shown. There is no guard bit register for `P` because a single multiplication result always fits into 32-bit register. The arithmetic registers can be used either as 16-bit registers mentioned above or as 40-bit registers (`A, B, C, D, P`).

Address registers are the 16-bit index registers `I0, I1, ..., I7`.

Control registers are the program counter `PC`, link registers `LR0, LR1` and mode register `MR0`. Loop hardware registers are `LS, LE, LC`, and page registers `IPR0, IPR1`.

2.1 Datapath

This picture shows the VSDSP datapath. The ALU has eight 16-bit arithmetic registers A0, A1, B0, . . . , D0, D1 and four 8-bit guard bit registers A2, . . ., D2. These can be combined to form 40-bit accumulators A, B, C and D. Calculation can be performed in 40-bit or 16-bit mode. The width depends on the operands. If one of the operands is 40 bits wide, the operation is performed in 40 bits, otherwise in 16 bits.

The multiplier unit is a 16 × 16-bit signed/unsigned integer/fractional saturating/unsaturating multiplier. Multiplier inputs can be A0, A1, B0, B1, C0, C1, D0, D1. The result goes to a 32-bit register P, which can be used as the second ALU operand in 40-bit arithmetic and is also used with MAC or MSU.

The 16/40-bit ALU implements the arithmetic and logic instructions. The ALU produces negative, carry, overflow, zero, and extension flags. There is also a 16/40-bit barrel shifter.

Two internal data buses connect the datapath registers to other registers and memories.

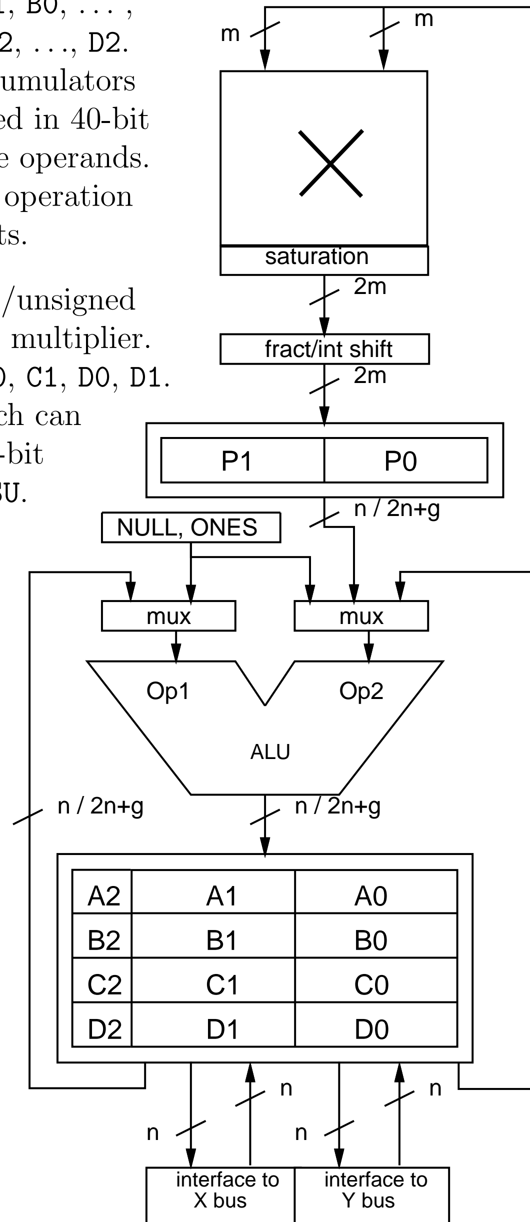


Figure 3: VS_DSP datapath.

2.2 ALU

The ALU can calculate either 40-bit or 16-bit operations. The width depends on the operands; if one of the operands is 40 bits wide, the operation is 40 bits and the result is stored to a 40-bit register. If both operands are 16 bits, the operation and result are also 16 bits and the result is stored to a 16 bit register. Exceptions to these rules are `EXP`, `ASHL` and `RND`. The result of `EXP` and `RND` is always 16-bit wide, and Op2 of `ASHL` is always an 16-bit register.

The 16-bit operands are `A0`, `A1`, `B0`, `B1`, `C0`, `C1`, `D0`, `D1`. Pseudo-registers `NULL` and `ONES` are also available and contain all zeros and all ones, respectively. `NULL` and `ONES` are considered to be 16-bit registers for the purpose of determining the operation width.

The 40-bit operands are `A`, `B`, `C` and `D`. `P` is only available as operand2. The register `A` is formed by concatenating `A2:A1:A0`. `A0` is the lsb part. For 40-bit calculations, also 16-bit registers are available as the other operand. In this case, the register is used as the middle part of the operand. The lsb end is padded with 16 zeros and the sign is extended to the guard bits. For example, if register `A0` is used with an 40-bit operand, the operand is `xx:A0:0000` (xx means sign extension bits).

The result register of 40-bit operation must be one of `A`, `B`, `C`, or `D`. The result register of a 16-bit operation is one of the 16-bit registers `A0`, . . . , `D1`.

2.3 Multiplier

The multiplier is a 16×16 signed/unsigned integer/fractional saturating/unsaturating multiplier.

Both inputs can be interpreted either as signed or unsigned numbers, to facilitate multi-precision operations. The integer/fractional mode bit controls the 1-bit left shift of the result (fractional mode). In fractional signed×signed multiplication, saturation is optionally (in saturation mode) included so that the result of `0x8000` × `0x8000` is `0x7fffffff`. The `P` register length is 32 bits.

The `P` register can be saved by executing `ADD NULL, P, An`. The high and low parts will reside in the high and low parts of the target accumulator, respectively. `P` can be restored by executing `RESP`.

2.4 Barrel Shifter

The barrel shifter can operate in both 40-bit and 16-bit mode. In 40-bit mode it can shift $0 \ldots 39$ bits logically left when operand2 is positive, or up to 39 bits arithmetically right when operand2 is negative. The result is undefined if the value of the operand2 register is out of range $-39 \ldots 39$.

In 16-bit mode Operand2 must be in range $-15 \ldots 15$.

The last bit shifted out is copied to the carry flag. When shifting left, the overflow flag is set if the msb bit is changed during shifting. When overflow happens in the saturation mode, overflow flag is set and result is saturated.

2.5 Guard Bit Registers

Guard bit registers behave as an extension of registers `A1`, `B1`, `C1`, and `D1`.

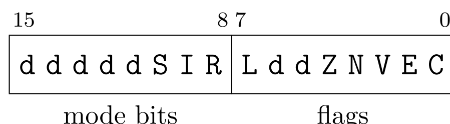
Whenever the arithmetic register `A1` is written to as a 16-bit register, either from a data bus or from ALU, the value is sign-extended to `A2`. Writes to `B1`, `C1`, and `D1` behave in the same way.

This does not happen when ALU operates in 40-bit mode and the result is written to `A`.

If you restore 40-bit values, remember to write to the guard bit register last, otherwise a write to `A1/B1/C1/D1` will sign-extend over the desired value. This is usually an issue only in interrupt handlers.

2.6 Flags and Mode Bits

The processor mode register includes mode bits and status flags. The bits affecting or being affected by the datapath are:



Bit/flag	Meaning
S	saturation mode
I	integer(1)/fractional(0) mult. mode
R	rounding mode
L	loop flag
Z	zero flag
N	negative flag
V	overflow flag
E	extension flag
C	carry flag

2.6.1 Saturation (S)

If the saturation mode bit is set, the ALU and multiplier operations will saturate the result in case of an over/underflow. The overflow flag will be set, but its interpretation is that saturation has taken place in the ALU.

If the mode bit is clear, the ALU and multiplier will not saturate their outputs, and the overflow flag will have its normal meaning.

2.6.2 Integer (I)

If the integer mode bit is set, the multiplier result is interpreted as an integer and thus no re-alignment is needed.

Otherwise, the multiplier result is assumed to be a fractional number with two leading sign bits, which will be re-aligned by a single left-shift before storing in the `P` register. Normally, a zero will be fed into the LSB. In saturation to the largest positive value, the LSB will be set to one.

2.6.3 Rounding (R)

If the rounding mode bit is set, `RND` will round using convergent 0 rounding, otherwise `RND` will always round towards 0.

2.6.4 Loop (L)

Loop flag is needed with 32-bit code space. The loop flag is set by the interrupt mechanism to disable loop end detection. This prevents false loop end detections when an interrupt causes the execution to transfer to zero page from another page. Normally, there is no need for the user to set or clear the loop flag.

- Interrupt sets the loop flag.
- MR0 load can set or clear the loop flag.
- JR, RETI, J, CALL, and LOOP instructions clear the loop flag.
- JMPI does not affect the loop flag.

2.6.5 Zero (Z)

If the ALU is operating in the 40-bit mode and bits $39 \ldots 0$ of the ALU result are all clear, the flag is set. If the ALU is operating in the 16-bit mode and bits $15 \ldots 0$ of the ALU result are all clear, the flag is set. Otherwise, the flag is cleared.

2.6.6 Negative (N)

If the ALU is operating in the 40-bit mode and bit 39 of the ALU result is set, the flag is set. If the ALU is operating in the 16-bit mode and bit 15 of the ALU result is set, the flag is set. Otherwise, the flag is cleared.

2.6.7 Overflow (V)

Set if an arithmetic overflow occurs in the ALU result. Otherwise cleared.

2.6.8 Extension (E)

If the ALU is operating in the 40-bit mode and bits $39 \ldots 31$ are all the same (either all ones or all zeros), the flag is cleared. Otherwise, the flag is set. If the ALU is operating in the 16-bit mode, the flag is cleared.

2.6.9 Carry (C)

If a carry is generated in an addition or a borrow is not generated in a subtraction, the flag is set. The flag is set also in `ASR`, `LSR` and `LSRC`, if the LSB bit of the operand is logical '1'.

Otherwise, the flag is cleared.

3 Data Address Generator

The data address generator uses index registers I0 $\cdots$ I7 to generate X and Y data bus addresses each cycle.

Each register In has a corresponding register pair $\overline{\texttt{In}}$. You get $\overline{\texttt{In}}$ by inverting the LSB bit of the number of register In. For example, the pair of I3 is I2, and the pair of I2 is I3.

Any In can be used as a X or Y data bus address. If needed, $\overline{\texttt{In}}$ specifies a post-modification for In. 32-bit X addresses are formed by concatenating $\overline{\texttt{In}}$ and In, but these are only useful with chips that have external data buses.

3.1 Post-modification Modes

There are two post modification modes specified in the instruction: post-modification by $-7 \ldots +7$ or post-modification by $\overline{\texttt{In}}$.

- `ldx (i0),a0` – load a0, no post-modification
- `ldx (i0)+6,a0` – load a0, post-modification by +6
- `ldx (i0)-7,a0` – load a0, post-modification by -7
- `ldx (i0)*,a0` – load a0, post-modification by $\overline{\texttt{I0}}$, i.e. I1

The modification by $\overline{\texttt{In}}$ (i.e. using *) uses the most significant bits of $\overline{\texttt{In}}$ to specify the post modification mode: linear post-modification, modulo post-modification and bit reverse.

In(15:13)	Mask	Modification
000	0x0000	In = (In+m) (m positive)
001	0x2000	In = [(In+m(12 : 6)) % (m(5 : 0) + 1)]
01x	0x4000	In = [(In+m(13 : 6)) % (m(5 : 0) × 64 + 64)]
100	0x8000	In = [(In+1) % (m + 1)]
101	0xa000	In = [(In−1) % (m + 1)]
110	0xc000	In = (In+m) bit reverse
111	0xe000	In = (In+m) (m negative)

When modulo addressing is used, modulo logic keeps the address within a circular buffer. The buffer length does not need to be a power of two, but the starting address of the buffer must be aligned to the nearest larger or equal power of two.

The bit-reverse modification is useful for FFT and DFT implementations.

3.1.1 Linear Post-Increment/Decrement

Linear post-modification can be an immediate -7 $\cdots$ +7 modification or modification by $\overline{\mathtt{In}}$. In the case of a negative modifier, $\overline{\mathtt{In}}$ contains the value in two's complement format.

- `ldx (i0)+5,a0` – load a0, post-modification by +5
- `ldc -10,i1`
  `ldx (i0)*,null` – no load, post-modification by -10
- `ldc 8191,i0`
  `ldy (i1)*,a0` – load a0, post-modification by 8191

3.1.2 Modulo Post-Increment/Decrement

In modulo modification the modified address is kept inside the circular buffer. This requires that the buffer start address is aligned to a power-of-two boundary according to the buffer size.

There are four different modulo modes. The most used ones are the +1 and -1 updates (masks 0x8000 and 0xa000). The lower bits of $\overline{\mathtt{In}}$ give the size of the modulo buffer minus one.

- `ldc 0x8000+BUFSIZE-1,i1`
  `ldx (i0)*,null` – no load, post-modification by +1 modulo BUFSIZE
- `ldc 0xa000+BUFSIZE-1,i1`
  `ldx (i0)*,null` – no load, post-modification by -1 modulo BUFSIZE

The other modulo modes can modify the address by larger steps than 1, but they have restrictions on what the buffer size can be. If the buffer size is 1..64 the modification can be -64..63. If the buffer size is a multiple of 64 (from 64 to 4096), the modification can be -128..127.

- `ldc 0x2000+((STEP&0x3f)<<6)+((BUFSIZE-1)&0x3f),i1`
  `ldx (i0)*,null` – post-modification by STEP modulo BUFSIZE
- `ldc 0x4000+((STEP&0x7f)<<6)+((BUFSIZE/64-1)&0x3f),i1`
  `ldx (i0)*,null` – post-modification by STEP modulo BUFSIZE

3.1.3 Bit Reversal

In bit reversal addressing, calculated addresses are kept within a buffer length 2^k and when calculating the updated address, carry is propagated towards the LSB. The lower boundary of the buffer is a multiple of 2^k. The boundary is decided by finding the highest 1-bit in $\overline{\texttt{In}}(12:0)$.

3 MSBs of $\overline{\texttt{In}}$ should contain `110` to select bit reversal addressing. LSBs of $\overline{\texttt{In}}$ should contain the reversed adder value, normally 2^{k-1}.

$\texttt{In} = \texttt{In} + \overline{\texttt{In}}[12 \cdots 0]$ (propagate carry towards LSB)

Example (64-point (k = 6) FFT in buffer 0x3000 ··· 0x303f), getting the next entry after 0x3030:

	15 ... 8	7 ... 0	
In =	0 0 1 1 0 0 0 0	0 0 1 1 0 0 0 0	0x3030
$\overline{\texttt{In}}$ =	1 1 0	0 0 0 0 0 0 0 1 0 0 0 0 0	0xc020
updated In =	0 0 1 1 0 0 0 0	0 0 0 0 1 0 0 0	0x3008

The previous example shows the normal usage, although other values than power of two are possible. The next example shows how to go backwards instead of forwards by setting $\overline{\texttt{In}}(12:0)$ to $2^k - 1$ instead of 2^{k-1}.

Example (64-point (k = 6) FFT in buffer 0x3000 ··· 0x303f), getting the previous entry before 0x3030:

	15 ... 8	7 ... 0	
In =	0 0 1 1 0 0 0 0	0 0 1 1 0 0 0 0	0x3030
$\overline{\texttt{In}}$ =	1 1 0	0 0 0 0 0 0 0 1 1 1 1 1 1	0xc03f
updated In =	0 0 1 1 0 0 0 0	0 0 0 1 0 0 0 0	0x3010

4 Program control

Program control unit (pcu) performs instruction fetch and decode, control flow changes and interrupt fetching. In addition to the program counter PC, program control unit has two link registers which are used for indirect jumps, LR0 and LR1.

Mode register MR0 holds the mode and flag bits. Loop control has three registers, LS, LE and LC. Program counter is not directly accessible.

Instruction Address Generator contains all pcu registers. Instruction Address Generator drives Instruction Address Bus from PC, LR0, LR1, interrupt address or from instruction jump address.

To achieve 32-bit instruction address space (large-code), two page registers are used. IPR0 holds the uppermost part of the instruction address. IPR0 and PC together determine the instruction address. IPR0 is copied to IPR1 during interrupts.

Interrupt Controller processes interrupts. It implements the interrupt state machine. Interrupt Controller receives external interrupt and drives interrupt fetch signal to Instruction Address Generator. Interrupt Controller makes sure that previous interrupt has been processed before new interrupt request is presented to Instruction Address Generator.

4.1 PC

PC is the program counter. It is not directly accessible by the programmer. PC is loaded with the fetch address+1 value on all cycles except when new loop round starts. In this case PC is loaded with LS. PC is kept at the old value if the instruction data and address buses are used by LDI or STI.

In interrupts, PC is copied to LR1.

In instruction fetches, instruction address bus (IAB) is driven either from PC, LR0, LR1, decoded instruction jump target address, reset vector address, interrupt vector address, or calculated address for LDI or STI.

4.2 LR0

LR0 is used in indirect jumps. JRcc causes instruction to be fetched from LR0 address instead of PC address, if condition cc is true. LR0 is used to save the return address for subroutine calls, so executing JRcc at the end of the subroutine returns to the caller. If nested subroutines are needed, the previous LR0 must be saved and restored by the caller.

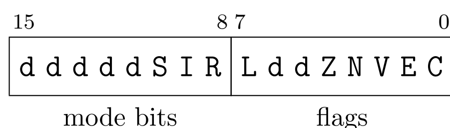
4.3 LR1

LR1 is used in interrupt returns. RETI causes instruction to be fetched from LR1 address instead of PC address. PC is copied to LR1 on interrupts.

If nested interrupts are needed, LR1 must be saved and restored by the interrupt service routine. See section 5.4.1 for the save and restore routines.

4.4 MR0

MR0 is the processor mode / status flag register.



Bit/flag	Meaning
S	saturation mode
I	integer(1)/fractional(0) mult. mode
R	rounding mode
L	loop flag
Z	zero flag
N	negative flag
V	overflow flag
E	extension flag
C	carry flag

In the end of an interrupt, MR0 is restored from the stack. Thus explicit moves override the evaluation of flags.

The mode bits and flags are described in more detail in section 2.6.

4.5 IPR0, IPR1

IPR0 is the instruction page register and is used to implement 32-bit code address space. It holds the upper 16 bits of instruction address. IPR0 can be changed by JRcc or JMPI instruction.

In interrupts IPR0 is copied to IPR1 at interrupt cycle #2.

4.6 LS, LE, LC

LS holds the loop start address. LE holds the loop end address. LC holds the loop count.

LOOP instruction copies instruction fetch address to LS, loads LE with loop end address specified in the LOOP instruction, and copies LC from the specified register.

When instruction fetch occurs from LE address and the L-flag is not set, LC is tested. If LC $\neq 0$, it is decremented by one, new loop round starts by copying LS to PC. If LC $= 0$, fetch continues from the next address.

LE is initiated with all ones in system reset.

5 Control Flow

The control flow behaviour follows the three-stage pipelining of the processor operation. The change-of-flow instructions are all delayed, with one delay slot following the instruction. There can not be another change-of-flow instruction in the delay slot. In this sense, also LOOP is considered as a change-of-flow instruction, in addition to J, Jcc, JRcc, CALLcc and RETI.

The JMPI instruction is also a change-of-flow instruction and has the same kind of timing behaviour as other change-of-flow instructions, but the instruction in the delay slot is canceled (executed as NOP), and can therefore be a change-of-flow instruction. This feature is mostly used in the interrupt vector table.

5.1 Jumps

Jump conditions are taken from the flags in MR0. The flags that are part of the condition must be unaltered in the preceding instruction. Other flags can be modified.

5.2 Loops

The loop mechanism has three registers which are loop start register LS, loop end register LE and loop count register LC.

Change-of-flow instructions can not be at loop end address or immediately before that.

LOOP instruction starts a hardware loop. LOOP instruction has one delay slot, i.e., loop start address is LOOP+2. This results from the fact that instruction at LOOP+1 (delay slot) is fetched before loop registers are updated by LOOP instruction. Loop can also be initiated by setting LS, LE and LC to appropriate values.

When the instruction fetch address equals LE, the value of LC is checked. If LC is not equal to zero, it is decremented by 1 and PC is loaded with LS. If LC is equal to zero, executing continues linearly from the next instruction.

5.3 System Reset

System reset forces the processor to a known reset state. After reset is released, the processor starts executing instructions from reset address onwards.

All registers except `LE` and `PC` are zeroed on reset. `LE` is set to all ones. `PC` is set to reset vector (normally 0x4000).

5.4 Interrupts

Interrupts are vectored using a jump table. The external interrupt peripheral supplies an interrupt vector to core. The vector is an address in the range 0x20...0x3f. These addresses must hold a jump table with `JMPI` instructions which jump to the start of the appropriate interrupt routine.

In interrupts `LR1` is used to save the return address. When main program is interrupted, return address is automatically copied to `LR1`. Interrupts normally end with a `RETI` (jump to `LR1`) or a `JRcc`(jump to `LR0`).

When generating an interrupt request, the interrupt peripheral automatically disables further interrupts by increasing its interrupt disable count register. If nested interrupts are required, the interrupt handler must save `LR1` before enabling further interrupts.

Note that if you call C-compiled routines from the interrupt handler, you must also save `P` and the guard bit registers.

5.4.1 Interrupt Routines

A typical interrupt jump table looks like the following:

```
  .org 0x20
  JMPI int_routine0,(SP)+1
  JMPI int_routine1,(SP)+1
  JMPI int_routine2,(SP)+1
  ...
```

Here, the `JMPI` instructions also increases the stack pointer.

The start of the interrupt handler must save the processor state before enabling interrupts in the interrupt controller. The end of the handler restores the processor state. Depending whether only 16-bit or both 16- and 32-bit code model will be used in the program, a different kind of a saving and restoring is used.

The following is a typical 16-bit (small-code space) interrupt routine:

```
_int_routine0:
  STX mr0,(i6) ; STY i7,(i6)+1
  STX lr1,(i6) ; STY lr0,(i6)
  ...
  (actual interrupt functionality)
  ...
  LDC INT_GLOB_EN,i7
  LDX (i6),lr1 ; LDY (i6)-1,lr0
  LDX (i6),mr0
  RETI
  STX i7,(i7) ; LDY (i6)-1,i7
```

When an interrupt is taken, the interrupt controller automatically disables all interrupts. Writing to the memory-mapped register INT_GLOB_EN enables the interrupts.

The interrupts must be disabled during the `RETI` instruction execution, and they will therefore be enabled in its delay slot. The `RETI` will also clear the L-flag, and the restoring of `MR0` must therefore come before it, if the flag is not cleared by the user.

The following is a typical 32-bit (large-code) interrupt routine.

```
STX i7,(i6)+1 ; STY lr0,(i6)
STX ipr1,(i6)+1 ; STY lr1,(i6)
STX mr0,(i6)
...
(actual interrupt functionality)
...
LDX (i6)-1,mr0
LDC INT_GLOB_EN,i7
STY i7,(i7)
LDX (i6)-1,i7 ; LDY (i6),lr0
JR (i7)       // clears the loop flag
LDX (i6)-1,i7 ; LDY (i6),lr0
```

`I7` and `LR0` must be restored in the delay slot of the JR instruction, because JR uses them both.

5.5 Halt

In HALT, the processor waits until an interrupt occurs. The execution pipeline is stopped.

When an interrupt occurs, the processor executes 3 instructions after the HALT instruction before executing the first interrupt instruction.

If the interrupt state machine is not in the idle state when HALT goes to execution, HALT instruction has no effect and is executed like a NOP.

6 Instruction Set Reference

6.1 List of Instructions

The following table lists all basic and optional instructions. The operands of each instruction, mode bits affecting the operation and the flags affected are also shown.

Mnemonic	meaning	operands	result	S	I	R	L	Z	N	V	E	C
ABS	absolute value	Areg	Areg	u	–	–	–	x	x	x	x	x
ADD	add	2×Areg	Areg	u	–	–	–	x	x	x	x	x
ADDC	add with carry	2×Areg,c	Areg	u	–	–	–	x	x	x	x	x,u
AND	logical AND	2×Areg	Areg	–	–	–	–	x	x	0	x	0
ASHL	n-b arithmetic shift	2×Areg	Areg	u	–	–	–	x	x	x	x	x
ASR	1-b arithmetic right shift	Areg	Areg	–	–	–	–	x	0	0	x	x
CALLcc	conditional call	addr,cc	PC, LR0	–	–	–	0	u	u	u	u	u
EXP	count leading bits	Areg	Areg	–	–	–	–	x	0	0	0	0
HALT	wait for an interrupt	–	–	–	–	–	–	–	–	–	–	–
Jcc	conditional jump	addr,cc	PC	–	–	–	0	u	u	u	u	u
JMPI	jump, ignore delay slot	addr,In	PC, In	–	–	–	–	–	–	–	–	–
JRcc	conditional jump with LR0	LR0, cc, In	PC	–	–	–	0	u	u	u	u	u
LDC	load constant	imm	reg	–	–	–	–	–	–	–	–	–
LDX	load on X bus	In, $\overline{In}$	reg	–	–	–	–	–	–	–	–	–
LDY	load on Y bus	In, $\overline{In}$	reg	–	–	–	–	–	–	–	–	–
LDI	load on I bus	In, $\overline{In}$	Areg	–	–	–	–	–	–	–	–	–
LOOP	start loop	reg, addr	Lregs	–	–	–	0	–	–	–	–	–
LSL	1-b logical left shift	Areg	Areg	–	–	–	–	x	x	x	x	x
LSLC	LSL with carry	Areg,c	Areg	–	–	–	–	x	x	x	x	x
LSR	1-b logical right shift	Areg	Areg	–	–	–	–	x	0	0	x	x
LSRC	LSR with carry	Areg,c	Areg	–	–	–	–	x	x	0	x	x
MAC	multiply-accumulate	2×Areg	Areg,P	u	u	–	–	x	x	x	x	x
MSU	multiply-subtract	2×Areg	Areg,P	u	u	–	–	x	x	x	x	x
MUL	multiply	2×Areg	P	u	u	–	–	–	–	–	–	–
MVX	register move	reg	reg	–	–	–	–	–	–	–	–	–
MVY	register move	reg	reg	–	–	–	–	–	–	–	–	–
NOP	no operation	–	–	–	–	–	–	–	–	–	–	–
NOT	logical NOT	Areg	Areg	–	–	–	–	x	x	0	x	0
OR	logical OR	2×Areg	Areg	–	–	–	–	x	x	0	x	0
RESP	restore P	2×Areg	P	–	–	–	–	–	–	–	–	–
RETI	jump with LR1	LR1, In	PC	–	–	–	0	–	–	–	–	–
RND	round to 16 bits	Areg	Areg	–	–	u	–	x	x	x	0	0
SAT	saturate to 32 bits	Areg	Areg	–	–	–	–	x	x	x	0	0
STX	store on X bus	In, $\overline{In}$, reg	mem	–	–	–	–	–	–	–	–	–
STY	store on Y bus	In, $\overline{In}$, reg	mem	–	–	–	–	–	–	–	–	–
STI	store on I bus	In, $\overline{In}$, Areg	mem	–	–	–	–	–	–	–	–	–
SUB	subtract	2×Areg	Areg	u	–	–	–	x	x	x	x	x
SUBC	SUB with carry	2×Areg,c	Areg	u	–	–	–	x	x	x	x	x,u
XOR	logical XOR	2×Areg	Areg	–	–	–	–	x	x	0	x	0

Operands and result: reg = register, In = index, $\overline{In}$ = modifier, addr = address, cc = condition code, c = carry in, imm = immediate data, Lregs = loop registers, P = multiplier result, PC = program counter, mem = memory location
Mode bits and flags: x = sets flag, u = uses bit, 0 = sets flag to 0,

6.2 Instruction Descriptions

The instruction description includes the **mnemonic** and a **one line description** of the operation, the **syntax** and **mathematical expression** of the operation, comments on the use and other **specific information**, and finally the **coding** of the instruction. The operand fields or other further refinements are given in accompanying **tables**.

Several operations can be executed in parallel when they are using different fields of the instruction word, e.g., ALU operations and two parallel moves with indirect addressing are possible, see instruction composition in chapter 7. In assembler the parallel operations are separated by a semicolon. The following lists the main rules.

One instruction can contain:

- Any single operation
  ```
  LDC 1234,i0
  J label
  ```
- ALU operation and any load or store
  ```
  sub a0,a1,b0 ; ldx (i1)-4,i0
  ```
- ALU operation and any register move
  ```
  add a1,null,a0 ; mv a2,a1
  ```
- Two register moves (there are some register bank restrictions)
  ```
  mv a0,i0 ; mv a1,i1
  ```
- One X and one Y load or store
  ```
  ldx (i6)-1,a0 ; ldy (i6),a1
  ldx (i0)+7,a0 ; sty a0,(i2)+1
  ```
- ALU operation and one restricted X and one restricted Y load or store
  ```
  mac a0,a1,b ; ldx (i0)*,a0 ; ldy (i2)*,a1
  ```
  In restricted (short) load/store one can only use the * modification or no modification, and the data register must be an ALU register.

ABS Absolute value

ABS $Op2, \mathtt{A}_n$; $|Op2| \rightarrow \mathtt{A}_n$
Flags: Z,N,V,E,C.

The operand is conditionally negated (two's complement operation) and placed in the target register. The coding of Op2 is given in Table 4 (ALU operand), and the result coding in Table 5. The absolute value of the minimum integer (fraction -1.0) is the maximum integer in the saturation mode.

Coding:

31 28	27 24	23 20	19 17	16 0
1 1 1 1	0 0 0 0	r r r r	A A A	parallel move

rrrr = Op2, AAA = target register.

ADD Addition of two operands

ADD $Op1, Op2, \mathtt{A}_n$; $Op1 + Op2 \rightarrow \mathtt{A}_n$
Flags: Z,N,V,E,C.

The operand coding is shown in Table 4 (ALU operand), and the result coding in Table 5. LSL is constructed with ADD Op1, Op1, $\mathtt{A}_n$.

Coding:

31 28	27 24	23 20	19 17	16 0
0 1 0 0	R R R R	r r r r	A A A	parallel move

RRRR = Op1, rrrr = Op2, AAA = target register.

ADDC Addition of two operands with carry

ADDC $Op1, Op2, \mathtt{A}_n$; $Op1 + Op2 + C \rightarrow \mathtt{A}_n$
Flags: Z,N,V,E,C.

The operand coding is shown in Table 4 (ALU operand), and the result coding in Table 5. LSLC is constructed with ADDC Op1, Op1, $\mathtt{A}_n$.

Coding:

31 28	27 24	23 20	19 17	16 0
1 0 0 0	R R R R	r r r r	A A A	parallel move

RRRR = Op1, rrrr = Op2, AAA = target register.

AND   Bitwise AND of two operands

$$\texttt{AND}\ Op1, Op2, \texttt{A}_n; \quad for\ each\ i : Op1[i] \cdot Op2[i] \rightarrow \texttt{A}_n[i]$$

Flags: `Z,N,V=0,E,C=0`.

The operand coding is found in Table 4 (ALU operand), and the result coding in Table 5.

Coding:

31–28	27–24	23–20	19–17	16–0
1 0 1 1	R R R R	r r r r	A A A	parallel move

`RRRR` = Op1, `rrrr` = Op2, `AAA` = target register.

ASHL   Arithmetic multi-bit shift

$$ASHL\ Op1, Op2, \texttt{A}_n; \quad if\ Op2 > 0 : Op1 << Op2 \rightarrow \texttt{A}_n : else\ Op1 >> |Op2| \rightarrow \texttt{A}_n$$

Flags: `Z,N,V,E,C`.

When Op2 is positive then the source is shifted left Op2 bits. Bits shifted out of position 40 are lost, but for the last bit is copied to the carry flag. Zeros are supplied to the vacated positions on the right.

When Op2 is negative then the source is shifted right abs(Op2) bits. Bits shifted out of position 0 are lost, but the last bit is copied to the carry flag. Copies of the MSB are supplied to the vacated positions on the left (arithmetic shift).

If a zero shift count is specified, the carry bit is cleared. Overflow flag is set if MSB is changed any time during the shift operation. This can only happen when shifting left.

Note: if the number of shifts exceeds the range of $-40 \ldots 40$ (or $-16 \ldots 16$ for 16-bit source/result) then the result is undefined.
Note2: Op2 is always 16-bit register.

The operand coding is found in Table 4 (ALU operand), and the result coding in Table 5.

Coding:

31–28	27–24	23–20	19–17	16–0
1 0 1 0	R R R R	r r r r	A A A	parallel move

`RRRR` = Op1, `rrrr` = Op2, `AAA` = target register.

ASR Arithmetic shift right

$$ASR\ Op2, \mathtt{A}_n;\quad for\ each\ i > 0:\ Op2[i] \rightarrow \mathtt{A}_n[i-1],\ Op2[msb] \rightarrow \mathtt{A}_n[msb]$$

Flags: `Z,N,V,E,C=op2(0)`.

The instruction shifts right by one position. The LSB bit is discarded, and MSB of the source registers is fed into the MSB bit of the result.

Coding:

31–28	27–24	23–20	19–17	16–0
1 1 1 1	0 0 0 1	r r r r	A A A	parallel move

`rrrr` = Op2, `AAA` = target register.

EXP Count leading bits

$$EXP\ Op2, \mathtt{A}_n$$

Flags: `Z,N=0,V=0,E=0,C=0`.

Count leading zeros or ones according to MSB of the source. The result is a unsigned integer in whose range of possible values are from 0 to $2n + g$. If Op2 is 0 then result is 0.

Note: Result is always written to 16-bit register.
Note2: This instruction can be used in conjunction with `ASHL` instruction, to specify the shift amount needed for normalization.

The operand coding is found in Table 4 (ALU operand), and the result coding in Table 5.

Coding:

31–28	27–24	23–20	19–17	16–0
1 1 1 1	0 1 0 1	r r r r	A A A	parallel move

`rrrr` = Op2, `AAA` = target register.

CALLcc Conditional delayed jump and save return address

$$CALL\ addr;\quad \mathtt{PC} \rightarrow \mathtt{LR0},\ if\ cond:\ addr \rightarrow \mathtt{PC}$$

Flags: `L=0`.

Identical to normal jump instruction, but `PC` is saved to `LR0`. This instruction replaces the sequence J addr, LDC @+1,LR0 which is used in subroutine calls.

Note the one delay slot associated to this instruction. The address which is saved to `LR0` is the CALL instruction address + 2. The instruction in the delay slot is always executed regardless of the condition.

Coding:

31 28	27 24	23 22	21 6	5 0
0 0 1 0	1 0 0 1	-	absolute address	condition

HALT Halt the processor and wait for an interrupt

HALT
Flags: `no change`.

The processor is halted to a low-power state. Normal execution is resumed when an interrupt occurs.

Coding:

31 28	27 24	23 0
0 0 1 0	1 1 0 1	-

Jcc Conditional delayed jump to absolute address

$\texttt{Jcc}\ addr;\quad if\ cond:\ addr \rightarrow \texttt{PC},\ else:\ \texttt{PC}+1 \rightarrow \texttt{PC}$
Flags: `L=0`.

Flags and their combinations can be used as jump conditions, as shown in Table 1 (Jump conditions). The instruction immediately before the `Jcc` must not change the flags that are used in the jump condition. Other flags can be changed. Note the one delay slot associated to this instruction.

Coding:

31 28	27 24	23 22	21 6	5 0
0 0 1 0	1 0 0 0	-	absolute address	condition

Table 1: Jump conditions.

Binary code	Abbrev	Name	definition
000000		always	
000001	CS	carry set	C = 1
000010	ES	extension set	E = 1
000011	VS	overflow	V = 1
000100	NS	negative	N = 1
000101	ZS	zero	Z = 1
001000	LT	less than zero	$N \bigoplus (V \cdot \overline{S}) = 1$
001001	LE	less than or equal to zero	$N \bigoplus (V \cdot \overline{S}) + Z = 1$
010001	CC	carry clear	C = 0
010010	EC	extension clear	E = 0
010011	VC	not overflow	V = 0
010100	NC	not negative	N = 0
010101	ZC	not zero	Z = 0
011000	GE	greater than or equal to zero	$N \bigoplus (V \cdot \overline{S}) = 0$
011001	GT	greater than zero	$N \bigoplus (V \cdot \overline{S}) + Z = 0$

JMPI Jump, ignore delay slot, increment index register

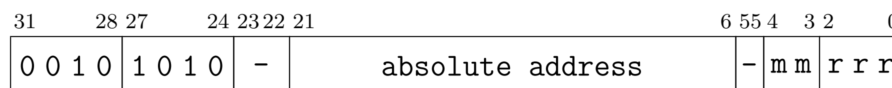
$$\texttt{JMPI}\ addr, (Op1) + n;\quad addr \rightarrow \texttt{PC},\ Op1 + n \rightarrow Op1,\ 0 \rightarrow \texttt{IPR0}$$

Flags: `no change`.

Identical to normal jump instruction, but ignores the instruction in the delay slot (a NOP is executed instead) and jumps to zero page. Also, the index register specified is optionally modified (identical to LDX (Op1)+n,NULL).

This instruction is used in interrupt vector jump table. *Do not use this instruction in normal code if interrupts are enabled.*

Coding:



`rrr` = address register, `dd` = don't care,
`mm` = address mode (00 = no update, 01 = +1, 11 = -1).

JRcc — Conditional delayed jump to the address in link register 0

`JRcc`; $if\ cond:$ `LR0` → `PC`
Flags: `L=0`.

JRcc — Conditional delayed jump to the address in link register 0

`JRcc` ($Op1$); $if\ cond:$ `LR0` → `PC`, $Op1$ → `IPR0`
Flags: `L=0`.

The `JRcc` instruction can be used for returns from subroutines, as well as for other jumps with run-time calculated addresses. The return addresses are typically loaded by an `LDC` instruction. Flags and their combinations can be used as jump conditions, as shown in Table 1 (Jump conditions). The instruction immediately before the `JRcc` must not change the flags that are used in the jump condition. Other flags can be changed. Unconditional return can be done with the "always" condition. Note the one delay slot associated to this instruction.

Coding:

31–28	27–24	23	22–6	5–0
0 0 1 0	0 0 0 0	0	-	condition

31–28	27–24	23	22–9	8–6	5–0
0 0 1 0	0 0 0 0	1	-	r r r	condition

`cccccc` = condition, `rrr` = Op1 (I0...I7)

LDC — Load constant to a register

`LDC` $constant, Op1$; $constant \rightarrow Op1$
Flags: `no change`.

The register (Op1) coding is shown in Table 10 (Target full move). The assembler understands numbers in different bases (e.g., hexadecimal, decimal, binary), while the immediate is finally coded in binary format. A single constant load can be done in an instruction, and no parallel arithmetic can be used. The constant is LSB-aligned and sign extended if needed.

Coding:

31–29	28–22	21–6	5–0
0 0 0	-	constant	R R R R R R

`RRRRRR` = Op1

LDX Load register from X-memory

$$\texttt{LDX}\ (Op1), Op2;\quad X[Op1] \rightarrow Op2,\ update\ Op1$$
Flags: `no change`.

LDY Load register from Y-memory

$$\texttt{LDY}\ (Op1), Op2;\quad Y[Op1] \rightarrow Op2,\ update\ Op1$$
Flags: `no change`.

Coding (double full moves):

31 28	27 14	13 0
0 0 1 1	X full move	Y full move

Coding (parallel full move):

31 28	27 24	23 20	19 17	16 12	11 8	7 4	3 0
o o o o	d d d d	d d d d	d d d	0 b 0 F F	F F F F	F F F F	F F F F

`oooo` = opcode allowing parallel moves, `dddd` = don't care
`b` = bus X/Y (0/1), `FFFFF` = full move bits of X/Y

Coding (parallel short moves):

31 28	27 24	23 20	19 17	16 12	11 8	7 4	3 0
o o o o	d d d d	d d d d	d d d	1 x x x x	x x x x	y y y y	y y y y

`xxxx` = short move bits of X, `yyyy` = short move bits of Y.

LDX Load register from X memory with long address

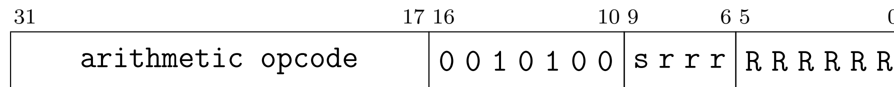
$$LDX\ (Op2 : Op3), Op1;\quad X[Op2 : Op3] \rightarrow Op1$$
Flags: `no change`.

STX Store register in X memory with long address

$$STX\ Op1, (Op2 : Op3);\quad Op1 \rightarrow X[Op2 : Op3]$$
Flags: `no change`.

Load or store a register from or to X memory. This instruction uses two index registers to generate a long (2×dataaddress) memory address. When Op2 is `In`, Op3 is the corresponding modifier register $\overline{\texttt{In}}$.

Coding (parallel move):



31–17	16–10	9–6	5–0
arithmetic opcode	0 0 1 0 1 0 0	s r r r	R R R R R R

RRRRRR = Op1, rrr = Op2, s = 1-store/0-load

LDI

Load register from I memory

$$LDI\ (Op2), Op1;\quad I[Op2] \rightarrow Op1,\ update\ Op2$$

Flags: no change.

STI

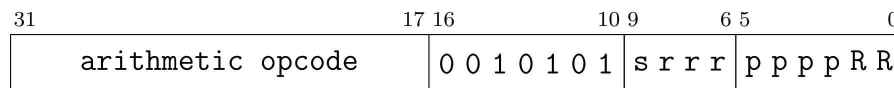
Store register to I memory

$$STI\ Op1, (Op2);\quad Op1 \rightarrow I[Op2],\ update\ Op2$$

Flags: no change.

Transfer data between I memory and registers. During the access the instruction data and address buses are not available for instruction fetches. The instruction is forced to NOP, PC update and LE compare are supressed. Op1 is A, B, C, or D, Op2 is In. The next instruction can not be a change-of-flow instruction.

Coding (parallel move):



31–17	16–10	9–6	5–0
arithmetic opcode	0 0 1 0 1 0 1	s r r r	p p p p R R

RR = Op1, rrr = Op2, s = 1-store/0-load, pppp = post-modification -7..7 or $\overline{\texttt{In}}$

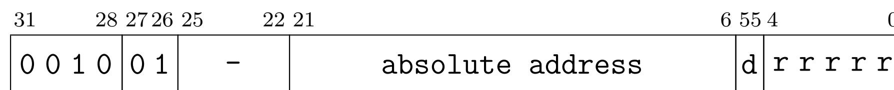
LOOP Start a hardware loop, delayed

$$\texttt{LOOP}\ Op1, addr; \quad Op1 \rightarrow \texttt{LC}, \quad addr \rightarrow \texttt{LE}, \quad \texttt{PC} + 2 \rightarrow \texttt{LS}$$

Flags: `L=0`.

This instruction starts a hardware loop. The instruction carries a register number, and an absolute loop end address which can be calculated by the assembler. The `LE` indicates the address of the last instruction within the loop body. The loop start is implicitly the second instruction from the `LOOP` instruction. See section 5.2 for details. Note the one delay slot associated to this instruction.

Coding:



`rrrrr` = Op1 (loop count), `nn...nn` = absolute loop end address.
`d` = don't care bit.

LSL[1] Logical shift left

$$\texttt{LSL}\ Op2, \texttt{A}_n;\quad for\ each\ i < bits-1:\ Op2[i] \rightarrow \texttt{A}_n[i+1],\ 0 \rightarrow \texttt{A}_n[0]$$

Flags: `Z,N,V,E,C=op2(bits-1)`.

The instruction shifts left by one position. This instruction is implemented in hardware as `ADD` Op2, Op2, $\texttt{A}_n$. **Note!** `P` is not available as an operand for this instruction.

Coding:

31 28	27 24	23 20	19 17	16 0
0 1 0 0	r r r r	r r r r	A A A	parallel move

`rrrr` = Op2, `AAA` = target register.

LSLC[2] Logical shift left with carry

$$\texttt{LSLC}\ Op2, \texttt{A}_n;\quad for\ each\ i < bits-1:\ Op2[i] \rightarrow \texttt{A}_n[i+1],\ C \rightarrow \texttt{A}_n[0]$$

Flags: `Z,N,V,E,C=op2(bits-1)`.

The instruction shifts left by one position. This instruction is implemented in hardware as `ADDC` Op2, Op2, $\texttt{A}_n$. **Note!** `P` is not available as an operand for this instruction.

Coding:

31 28	27 24	23 20	19 17	16 0
1 0 0 0	r r r r	r r r r	A A A	parallel move

`rrrr` = Op2, `AAA` = target register.

LSR Logical shift right

$$\texttt{LSR}\ Op2, \texttt{A}_n;\quad for\ each\ i > 0:\ Op2[i] \rightarrow \texttt{A}_n[i-1],\ 0 \rightarrow \texttt{A}_n[msb]$$

Flags: `Z,N,V,E,C=op2(0)`.

The instruction shifts right by one position. The LSB bit is discarded, and zero is fed into the MSB bit. The operand (Op2) is encoded as described in Table 4 (ALU operand), and the result coding in Table 5.

Coding:

31 28	27 24	23 20	19 17	16 0
1 1 1 1	0 0 1 0	r r r r	A A A	parallel move

`rrrr` = Op2, `AAA` = target register.

[1] This instruction is implemented as a single instruction software macro.

LSRC Logical shift right with carry

$$\texttt{LSRC}\ Op2, \texttt{A}_n; \quad for\ each\ i > 0: \ Op2[i] \rightarrow \texttt{A}_n[i-1],\ C \rightarrow \texttt{A}_n[msb]$$

Flags: `Z,N,V,E,C=op2(0)`.

The instruction shifts right by one position. The LSB bit is fed to carry, and carry is fed into the MSB bit. The operand (Op2) is encoded as described in Table 4 (ALU operand), and the result coding in Table 5.

Coding:

31–28	27–24	23–20	19–17	16–0
1 1 1 1	0 0 1 1	r r r r	A A A	parallel move

`rrrr` = Op2, `AAA` = target register.

MAC Multiply-accumulate

$$\texttt{MAC}\ Op1, Op2, \texttt{A}_n; \quad \texttt{A}_n + \texttt{P} \rightarrow \texttt{A}_n, Op1 \times Op2 \rightarrow \texttt{P}$$

Flags: `Z,N,V,E,C`.

The instruction performs one multiplication and adds the result of the previous multiplication (`P`) to a register. The multiplication operands are considered signed or unsigned (see `MUL`), multiplication mode and possible saturation are controlled by the appropriate mode bits.

Coding:

31–28	27–24	23–20	19–17	16–0
0 1 0 1	r r r m	m R R R	A A A	parallel move

`rrr` = Op1, `RRR` = Op2, `AAA` = target register, `mm` = data format.

MSU Multiply-subtract

$$\texttt{MSU}\ Op1, Op2, \texttt{A}_n; \quad \texttt{A}_n - \texttt{P} \rightarrow \texttt{A}_n, Op1 \times Op2 \rightarrow \texttt{P}$$

Flags: `Z,N,V,E,C`.

The instruction performs one multiplication and subtracts the result of the previous multiplication (`P`) from a register. The multiplication operands are considered signed or unsigned (see `MUL`).

Coding:

31–28	27–24	23–20	19–17	16–0
0 1 1 1	r r r m	m R R R	A A A	parallel move

`rrr` = Op1, `RRR` = Op2, `AAA` = target register, `mm` = data format.

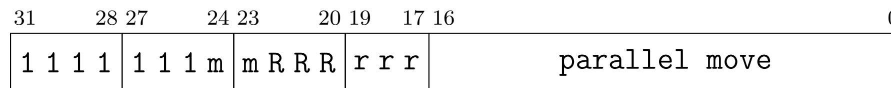
MUL Multiply

$$\texttt{MUL}\ Op1, Op2; \quad Op1 \times Op2 \rightarrow \texttt{P}$$

Flags: `no change`.

Performs one multiplication. The operands can be signed or unsigned, multiplication mode and possible saturation are controlled by the appropriate mode bits. There are different mnemonics for different format operands. The data format can be Op1 signed/Op2 signed (`MULSS`), Op1 unsigned/Op2 signed (`MULUS`), Op1 signed/Op2 unsigned (`MULSU`) or Op1 unsigned/Op2 unsigned (`MULUU`). The format `SS` is the default, and `MULSS` can thus be written as plain `MUL`.

Coding:



`rrr` = op1, `RRR` = op2, `mm` = data format.

MVX/MVY Register-to-register move

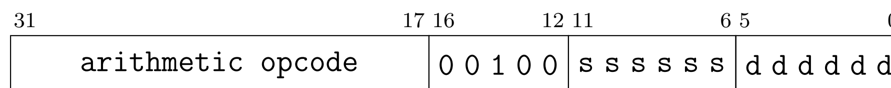
$$MVX\ Op1, Op2; \quad Op1 \rightarrow Op2$$

Flags: `no change`.

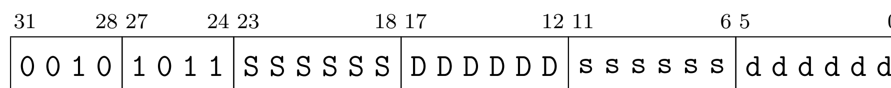
Moves a register to another register using X or Y data bus. In parallel MVX, any register can be used as a source or target. The source is read on X bus, switched to Y bus and written from Y bus.

In double MVX/MVY, two moves can be performed with a single instruction. The source and destination registers must be from different execution units (ALU, DAG, PCU).

Coding (parallel move):



Coding (double move):



`n` = reserved, `ssssss` = Y source, `dddddd` = Y tar get,
`SSSSSS` = X source , `DDDDDD` = X target.

NOP No operation

$$\texttt{NOP};\quad no\ effect$$
Flags: `no change`.

A parallel move `NOP` is a load operation to `NOP` register. A total `NOP` is `LDC` to `NOP`.

Coding:

31 … 28	27 … 24	23 … 20	19 … 17	16 … 0
1 1 1 1	0 1 0 0	d d d d	d d d	parallel move

`ddd` = don't care.

NOT[3] Bitwise logic NOT operation

$$\texttt{NOT}\ Op2, \texttt{A}_n;\quad for\ each\ i:\ \overline{Op2[i]} \rightarrow \texttt{A}_n[i]$$
Flags: `Z,N,V=0,E,C=0`.

The operand (Op2) coding is shown in Table 4 (ALU operand), the target can be one of the registers. In hardware this is equal to an `XOR` with register `ONES`.

Coding:

31 … 28	27 … 24	23 … 20	19 … 17	16 … 0
1 1 0 1	1 0 0 1	r r r r	A A A	parallel move

`rrrr` = Op2, `AAA` = target register.

OR Bitwise logic OR operation

$$\texttt{OR}\ Op1, Op2, \texttt{A}_n;\quad for\ each\ i:\ Op1[i] + Op2[i] \rightarrow \texttt{A}_n[i]$$
Flags: `Z,N,V=0,E,C=0`.

The operands are encoded as described in Table 4 (ALU operand), and the result coding in Table 5. The target is one of the registers.

Coding:

31 … 28	27 … 24	23 … 20	19 … 17	16 … 0
1 1 0 0	r r r r	R R R R	A A A	parallel move

`rrrr` = Op1, `RRRR` = Op2, `AAA` = target register.

RESP Restore P register

RESP $Op1, Op2$; $Op1 \rightarrow$ P0 $Op2 \rightarrow$ P1
Flags: no change.

This instruction restores the P contents from two arithmetic registers. The saving of the P shall be done as described in section 2.3. The operands are encoded as multiplication operands.

Coding:

31–28	27–24	23–20	19–16	15–12	11–8	7–4	3–0
0 0 1 0	0 0 1 0	d R R R	r r r d	d d d d	d d d d	d d d d	d d d d

rrr = Op1, RRR = Op2, ddd = don't care bits.

RETI Delayed return from interrupt

RETI; LR1 $\rightarrow$ PC
Flags: L=0.

RETI Delayed return from interrupt

RETI ($Op1$); LR1 $\rightarrow$ PC, $Op1 \rightarrow$ IPR0
Flags: L=0.

The RETI instruction is used for returns from interrupts, similarly as JRcc is used for returns from subroutines. For description of interrupt mechanism and the correct use of RETI, see chapter 5.

Coding:

31–28	27–24	23	22–0
0 0 1 0	0 0 0 1	0	-

31–28	27–24	23	22–9	8–6	5–0
0 0 1 0	0 0 0 1	1	-	r r r	-

rrr = Op1 (I0...I7)

RND Round and saturate a 40-bit ALU register to 32 bits

RND $Op2$, A_n
Flags: Z,N,V,E=0,C=0.

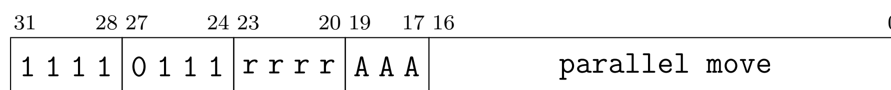
Round long ALU register to top 24 bits. If mode bit R is set, uses convergent 0 rounding (round exact x.5 values towards even numbers), otherwise round towards 0. After the number is rounded, it is saturated to the lowest 16 bits of the intermediary 24-bit result.

The result is a signed integer.

Note: Result is always written to 16-bit register.

The operand coding is found in Table 4 (ALU operand), and the result coding in Table 5.

Coding:



31–28	27–24	23–20	19–17	16–0
1 1 1 1	0 1 1 1	r r r r	A A A	parallel move

rrrr = Op2, AAA = target register.

SAT Saturate 40-bit ALU register to 32 bits

SAT $Op2$, A_n
Flags: Z,N,V,E=0,C=0.

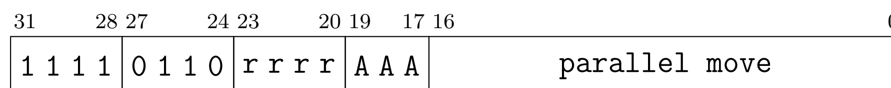
Saturate 40-bit register to 32-bit range. This is different from saturation mode set in MR0 register, which saturates ALU results to 40-bit range.

The overflow flag is set if Op2 was out of 32-bit range and saturation was made.

Note: Saturation mode bit in MR0 register does not affect this instruction.

The operand coding is shown in Table 4 (ALU operand), and the result coding in Table 5.

Coding:



31–28	27–24	23–20	19–17	16–0
1 1 1 1	0 1 1 0	r r r r	A A A	parallel move

rrrr = Op2, AAA = target register.

STX Store a register in X memory

$$\texttt{STX}\ Op1, (Op2);\quad Op1 \rightarrow X[Op2],\ \textit{update}\ Op2$$

Flags: `no change`.

See `LDX` for the general load/store capability description and the encoding of the move fields.

STY Store a register in Y memory

$$\texttt{STY}\ Op1, (Op2);\quad Op1 \rightarrow Y[Op2],\ \textit{update}\ Op2$$

Flags: `no change`.

See `LDX` for the general load/store capability description and the encoding of the move fields.

SUB Subtraction of two operands

$$\texttt{SUB}\ Op1, Op2, \texttt{A}_n;\quad Op1 - Op2 \rightarrow \texttt{A}_n$$

Flags: `Z,N,V,E,C`.

The operand coding is shown in Table 4 (ALU operand), and the result coding in Table 5.

Coding:

31–28	27–24	23–20	19–17	16–0
0 1 1 0	R R R R	r r r r	A A A	parallel move

`RRRR` = Op1, `rrrr` = Op2, `AAA` = target register.

SUBC

Subtraction of two operands with carry

$$\texttt{SUBC}\ Op1, Op2, \texttt{A}_n;\quad Op1 - Op2 - C \rightarrow \texttt{A}_n$$

Flags: `Z,N,V,E,C`.

The operand coding is shown in Table 4 (ALU operand), and the result coding in Table 5.

Coding:

31–28	27–24	23–20	19–17	16–0
1 0 0 1	R R R R	r r r r	A A A	parallel move

`RRRR` = Op1, `rrrr` = Op2, `AAA` = target register.

XOR

Bitwise logic XOR operation

$$\texttt{XOR}\ Op1, Op2, \texttt{A}_n;\quad for\ each\ i:\ Op1[i] \bigoplus Op2[i] \rightarrow \texttt{A}_n[i]$$

Flags: `Z,N,V=0,E,C=0`.

The operand coding of Op1 and Op2 is shown in Table 4 (ALU operand), and the result coding in Table 5. `XOR` has also been used to implement `NOT`.

Coding:

31–28	27–24	23–20	19–17	16–0
1 1 0 1	R R R R	r r r r	A A A	parallel move

`RRRR` = Op1, `rrrr` = Op2, `AAA` = target register.

6.3 Instruction Sequence Restrictions

There are certain sequences of instructions which, due to the pipelined execution, would produce undetermined results. These sequences are either flagged as errors by the software tools or masked off by the hardware.

6.3.1 Loop Register Restrictions

When either the `LE`, `LC` or `LS` register is loaded from memory with a `LDX` or `LDY` instruction, the loop end comparison is not done.

This means that loop registers can not be loaded by instruction whose address is $\texttt{LE}-2$. If this is done, further loop rounds are ignored and the execution continues linearly.

The `LDC` instruction does not have this restriction and the loop hardware uses the value loaded with an `LDC` if it is needed on the same cycle. Also, the `LOOP` instruction does not have the restriction so single instruction loops are allowed.

```
illegal_example:
  ldc loop_end1,le
  ldx (i0),lc                   /* le comparison not done */
  nop
loop_end1:
  nop

legal_example:
  ldc 2,lc
  ldc loop_start,ls
  ldc loop_end2,le              /* le comparison is done */
  nop
loop_end2:
  nop
```

6.3.2 Conditional Jump Restrictions

The instruction immediately before the jump instruction (`JRcc` or `Jcc`) must not change the flags that affect the jump condition.

For example, if the jump is a `JCC` (jump if carry clear) the instruction immediately before must not change the C flag. In practice, this means that instruction must not be an ALU instruction. X and Y memory accesses can be made since they do not affect the "carry clear" condition.

```
example:
  ldx (i0)+1, NULL             /* must not change C flag */
  jcc jump_target
  nop                          /* jump delay slot */
```

The reason for this restriction is the fact that the jump condition is determined during the decode phase. In a normal (linear) execution, the instruction immediately before the jump does not affect the jump. The situation is different if the jump instruction is canceled due to an interrupt. When execution returns from the interrupt to the normal execution flow, the instruction immediately before the jump has been executed. The jump condition is determined again, this time with different flags.

7 Instruction Coding

7.1 General Instruction Composition

The instruction is composed of a 4-bit opcode and additional fields as described below.

31 28	27 6	5 0
o o o o	i i i i i i i i i i i i i i i i i i i i i i	y y y y y y
opcode	immediate	target

31 28	27 0
o o o o	c c c c c c c c c c c c c c c c c c c c c c c c c c c c
opcode	control instruction

31 28	27 14	13 0
o o o o	x x x x x x x x x x x x x x	y y y y y y y y y y y y y y
opcode	X full move	Y full move

31 28	27 17	16 0
o o o o	a a a a a a a a a a a	m m m m m m m m m m m m m m m m m
opcode	arithmetic operands	parallel moves

7.2 Opcode Field

The encoding of operations is shown in Table 2. The control and double move extensions to the opcode are described in the following section.

7.3 Control Instructions

The absolute address in jump instructions is at most 20 bits. The conditional jumps `Jcc` are taken when the condition given in the instruction is true. See Table 1 (Jump condition) for the condition field coding. The flag and mode bits can be masked by the implementation parameter Modemask, see Chapter 4.

Return (`JRcc`) and return from interrupt (`RETI`) use the link registers to restore the `PC`. The linking (return address storage) is done by a constant load instruction to the link register `LR0` (the link register should be saved beforehand in case of a subroutine already being executed). The return address is calculated at compilation/linking time, not run-time. This allows also jumps by loading the link register

Table 2: Operation Codes

Binary code	Operation	Parallel
000X	LDC	none
0010	Control	none
0011	Double moves	none
0100	ADD	yes
0101	MAC	yes
0110	SUB	yes
0111	MSU	yes
1000	ADDC	yes
1001	SUBC	yes
1010	ASHL	yes
1011	AND	yes
1100	OR	yes
1101	XOR	yes
1110	(reserved)	
1111	Single op instructions	yes

Table 3: Control Instructions

Binary code	Operation	Sub-fields	Additional fields
0000dddddddd	JRcc		condition
0001dddddddd	RETI		
0010dxxxyyyd	RESP	x = op2, y = op1	
01nnnnnnnnnn	LOOP	n = loop end msb	loop end lsb, register (loop count)
1000nnnnnnnn	Jcc	n = address msb	address lsb, condition
1001nnnnnnnn	CALLcc	n = address msb	address lsb, condition
1010nnnnnnnn	JMPI	n = address msb	address lsb, index reg
1011nnnnnnnn	MVX/MVY		move fields
1101nnnnnnnn	HALT		
111000000000 ... 111111111111	(reserved)		

and then executing the JRcc instruction. The linking can be done also in the delay slot. The LR1 loading takes place automatically when interrupt processing is started.

In the loop instruction there is a register number containing the loop count. All registers except the double-size accumulators can be used. The loop end address is given as an immediate (at most 20 bits) value. The loop start address will be loaded automatically from the PC. The loop registers (LC, LS, LE) should not be loaded within the two instructions preceding a loop end to avoid implementation-dependent ambiguities in the loop behavior.

In the full size moves, the load/store operations can use all the addressing modes and all registers. These moves do not allow any control operations in parallel. See section 7.5 for move encoding.

RESP is a special instruction to restore the P register.

The rest of the control instructions are reserved for future extensions.

Table 4: ALU operand encoding.

Binary code	register	composition
0000	A0	S:A0:0000
0001	A1	S:A1:0000
0010	B0	S:B0:0000
0011	B1	S:B1:0000
0100	C0	S:C0:0000
0101	C1	S:C1:0000
0110	D0	S:D0:0000
0111	D1	S:D1:0000
1000	NULL	0:0000:0000
1001	ONES	F:FFFF:FFFF
1010	(reserved)	(reserved)
1011	P	S:P1:P0
1100	A	A2:A1:A0
1101	B	B2:B1:B0
1110	C	C2:C1:C0
1111	D	D2:D1:D0

7.4 Arithmetic Operands

The operands of two-operand arithmetic and logic instructions (ADD, SUB, AND, OR, XOR) are encoded in the second field of these instructions. The field is composed as follows:

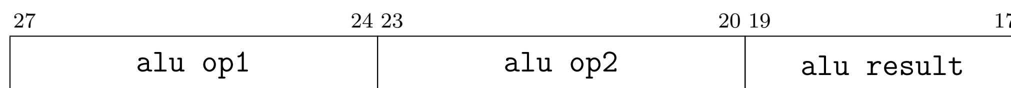


Table 4 (ALU operand) gives the encoding of Op1 and Op2 of the ALU (fields alu

Table 5: ALU result coding

Binary code	16-bit register	40-bit register
000	A0	(reserved)
001	A1	A
010	B0	(reserved)
011	B1	B
100	C0	(reserved)
101	C1	C
110	D0	(reserved)
111	D1	D

Table 6: Mul operand.

Binary code	register
000	A0
001	A1
010	B0
011	B1
100	C0
101	C1
110	D0
111	D1

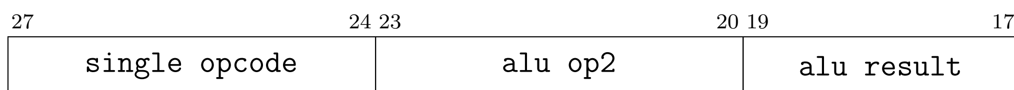
Table 7: Mul mode.

Binary code	op1	op2
00	signed	signed
01	signed	unsigned
10	unsigned	signed
11	unsigned	unsigned

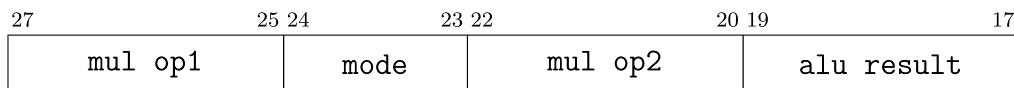
op1 & alu op2). S denotes sign extension.

Table 6 (Mul operand) gives the encoding of fields mac op1 and mac op2.

The opcode of single-operand arithmetic and logic instructions (`ABS`, `LSR` and `MUL`) is encoded in the first operand field. The encoding is:



In `MAC`:



In `MUL`:

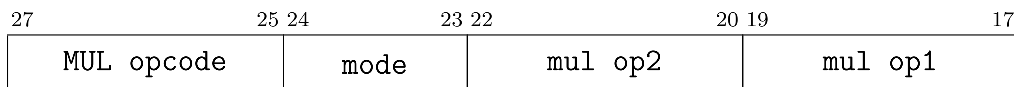


Table 7 (Mul mode) gives the encoding of the mode field.

The result field encoding is shown in Table 5.

Table 4 (ALU operand) gives the encoding of Op2 of the ALU (field alu op2).

Table 8: Single operand ALU instructions.

Binary code	Operation
0000	ABS
0001	ASR
0010	LSR
0011	LSRC
0100	NOP
0101	EXP
0110	SAT
0111	RND
1000 ... 1101	(reserved)
111X	MUL

The single-operand opcode encoding is given in Table 8.

7.5 Move Encoding

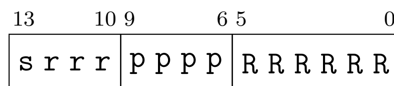
The move instructions are `LDX`, `LDY`, `LDI`, `STX`, `STY`, and `STI`, the X, Y, and I denoting the desired data bus to be used. There can be a maximum of two moves (loads or stores) in parallel, one operating on the X bus and the other on Y bus. Constant loading is described separately in section 7.7.

There are two kinds of moves: full moves and short moves.

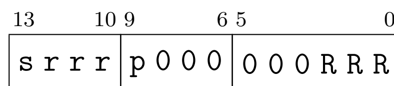
The short moves use a restricted set of registers and restricted addressing modes. The full moves have all registers and all addressing modes available.

The parallel moves can be done together with arithmetic operations, and can either be one full or two short moves. Long-X and I-bus moves are only available as parallel moves. Double full move instruction has two full moves, but can not be executed in parallel with other instructions.

The full move field is always the following 14-bit control field:

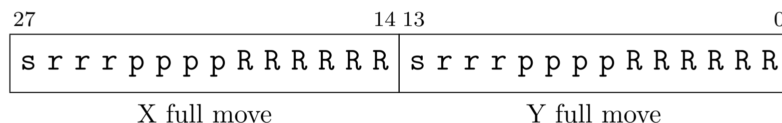


In short moves the move field is as follows:



`s` = 1-store/0-load, `r` = address register, `p` = post modification mode,
`R` = move source/destination register.

In the double full move the 14-bit fields come directly after the instruction.



Parallel move can be either one full move, two short moves, register-to-register move, long-X move, or I-bus move. The coding of parallel moves is:

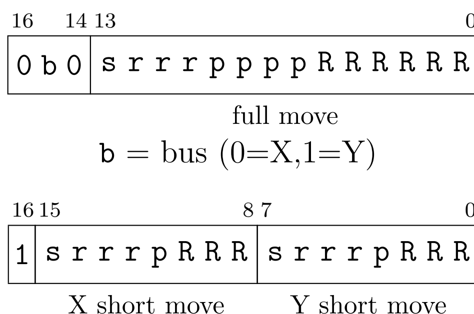


Table 9: Registers in short move.

Binary code	Register
00a	A0 ... A1
01a	B0 ... B1
10a	C0 ... C1
11a	D0 ... D1

16 14	13 12	11 0
0 0 1	0 0	s s s s s s d d d d d d

reg-to-reg move (Y bus)

16 14	13 10	9 0
0 0 1	0 1 0 0	s r r r R R R R R R

long-X move

16 14	13 10	9 0
0 0 1	0 1 0 1	s r r r p p p p R R

I-bus move

The coding of the store/load bit is given in Table 11. The `rrr` register is the number of the desired address register. The src/dest register number (`(RRR)RRR`) is given in Table 10 (Source and target), and the addressing mode in Table 12. See also section 7.6 for further description of the addressing modes available. The post modification `pppp` is a four-bit two's complement number (-7 ... +7), which is added to the address register. The code -8 is for the additional address post modification modes found in $\overline{\texttt{In}}$.

The $\overline{\texttt{In}}$ is the index register the number of which is generated by inverting the LSB bit of the number of register `In`. The post modifications by the $\overline{\texttt{In}}$ are defined in Table 13.

7.6 Addressing Modes

The addressing modes and their availability in short and full formats are summarized in Table 14. The addressing modes available in the implementation are controlled by the parameter *Addressing mode mask*, which has enable bits for the modulo, bit-reversal and (reserved) addressing modes in the following manner:

(reserved)	bitrev	modulo

For the details of how the modulus mode works, see Chapter 3.1.2.

Table 10: Registers in full move.

Binary code	Register
00000a	A0 ... A1
00001a	B0 ... B1
00010a	C0 ... C1
00011a	D0 ... D1
001000	LR0
001001	LR1
001010	MR0
001011	(reserved)
001100	NULL (update index reg & flags)
001101	LC
001110	LS
001111	LE (optional)
010rrr	I0 ... I7
100000	A2
100001	B2
100010	C2
100011	D2
100100	Move NOP (no updates)
100101 ... 111101	reserved
111110	IPR0
111111	IPR1

Table 11: Load/Store coding.

Binary code	Mode
0	load
1	store

Table 12: Addressing Modes.

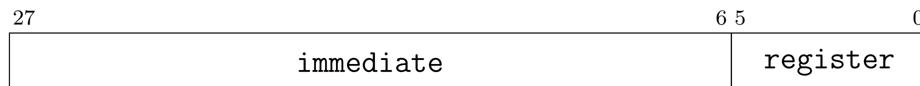
Binary code	Mode
rrrpppp	indirect [In] with post modify by pppp (-7...+7)
rrr1000	indirect [In] with post modification specified in $\overline{\texttt{In}}$

Table 13: Modifications by the $\overline{\texttt{In}}$ register.

Binary code	Modification
000	In = (In+m) (m positive)
001	In = [(In+m(12 : 6))%(m(5 : 0) + 1)]
01	In = [(In+m(13 : 6))%(m(5 : 0) × 64 + 64)]
100	In = [(In+1)%(m + 1)]
101	In = [(In−1)%(m + 1)]
110	In = (In+m) bit reverse
111	In = (In+m) (m negative)

7.7 Constant Loading

The additional fields in the constant load instruction LDC look like:



The immediates are assumed signed and will be sign extended if the register is wider than the immediate. In case there are more bits in the immediate than in the register to be loaded, the LSB part is taken. The register number is encoded as in the full addressing load/stores, shown in Table 10.

Table 14: Addressing mode summary.

Mode	full move code	short move code	$\overline{I_n}$	parameter
Linear post-inc/dec				
(In)	srrr0000RRRRRR	srrr0RRR	—	—
(In)+1	srrr0001RRRRRR	N/A	—	—
(In)+2	srrr0010RRRRRR	N/A	—	—
(In)+3	srrr0011RRRRRR	N/A	—	—
(In)+4	srrr0100RRRRRR	N/A	—	—
(In)+5	srrr0101RRRRRR	N/A	—	—
(In)+6	srrr0110RRRRRR	N/A	—	—
(In)+7	srrr0111RRRRRR	N/A	—	—
(In)–1	srrr1111RRRRRR	N/A	—	—
(In)–2	srrr1110RRRRRR	N/A	—	—
(In)–3	srrr1101RRRRRR	N/A	—	—
(In)–4	srrr1100RRRRRR	N/A	—	—
(In)–5	srrr1011RRRRRR	N/A	—	—
(In)–6	srrr1010RRRRRR	N/A	—	—
(In)–7	srrr1001RRRRRR	N/A	—	—
(In)*	Linear post-inc/dec			
(In)+m, $m \geq 0$	srrr1000RRRRRR	srrr1RRR	000 mmmm...mmmm	—
(In)+m, $m < 0$	srrr1000RRRRRR	srrr1RRR	111 mmmm...mmmm	—
(In)*	Modulo post-inc/dec			
(In)+n%m	srrr1000RRRRRR	srrr1RRR	001 nnnn...mmmm	amm[0]
(In)+n%m×64	srrr1000RRRRRR	srrr1RRR	01n nnnn...mmmm	amm[0]
(In)+1%m	srrr1000RRRRRR	srrr1RRR	100 mmmm...mmmm	amm[0]
(In)–1%m	srrr1000RRRRRR	srrr1RRR	101 mmmm...mmmm	amm[0]
(In)*	Bit reversal			
(In)+m bit-rev	srrr1000RRRRRR	srrr1RRR	110 mmmm...mmmm	amm[1]
Register as source/destination				
An	srrrpppp000RRR	srrrpRRR	—	—
An ext	srrrpppp1000RR	N/A	—	$g > 0$
LR0, LR1	srrrpppp00100R	N/A	—	—
MR0, MR1	srrrpppp00101R	N/A	—	—
NULL	srrrpppp001100	N/A	—	—
NOP	srrrpppp100100	N/A	—	—
LC	srrrpppp001101	N/A	—	$lc \geq 1$
LS	srrrpppp001110	N/A	—	$lc \geq 1$
LE	srrrpppp001111	N/A	—	$lc \geq 1$
In, n=0···7	srrrpppp010RRR	N/A	—	—

8 Software Examples

8.1 Single-Precision FIR Transversal Filter

This code implements an single-precision single-sample direct-form (transverse) 16-stage FIR filter. The input and the coefficients are 16 bits wide, the intermediate results being 32 bits.

```
        .sect data_x, XData
delay:
        .zero 15        // x[-15]...x[-1] (delay line) at startup
input:
        .uword 0x1234        // x[0] at startup
output:
        .zero 1

        .sect data_y, YData
coef:
        .zero 16

        .sect code, Single_precision_FIR
fir:
        LDC     0x400,mr0    // fractional & saturation mode
        LDC     input,i0     // point to the newest sample
        LDC     -1,i1        // linear -1
        LDC     coef,i2
        LDC     1,i3         // post-increment by 1 addressing
        LDC     output,i4    // pointer to output buffer
        AND     a,NULL,a; LDX (i0)*,b1; LDY (i2)*,b0
                // clear a-reg., load first sample/coef.-pair
        LDC     15,ls // loop count, number of loops minus one
                      // use otherwise unused ls-register
        LOOP    ls,firloopend-1   // start looping
        MUL     b1,b0;   LDX (i0)*,b1; LDY (i2)*,b0
                // perform first multiply, load next pair
firloop:
        MAC     b1,b0,a; LDX (i0)*,b1; LDY (i2)*,b0
                // use pipelined MAC to implement FIR
firloopend:
        STX     a1,(i4)      // store result
endfir:
        .end
```

8.2 Double-Precision FIR Transversal Filter

This code implements an double-precision single-sample FIR filter. The input and the filter coefficients are 32 bits wide, the intermediate results 64 bits.

Algorithm:
$(A \times 2^{16} + B) \times (C \times 2^{16} + D) = AC \times 2^{32} + AD \times 2^{16} + BC \times 2^{16} + BD$
In this example, AC is first added to a-reg, then BD to b-reg. and after that BC to a1:b0 and finally AD to a1:b0

```
        .sect data_x, XData
input:
input_hi:
        .uword 0x9234,0x6666,0x7654
        .zero 14
output:
output_hi:
        .zero 16
coef:
coef_hi:
        .uword 0x8001,0xffff,0x5656
        .zero 14

        .sect data_y, YData
input_lo:
        .uword 0x5678,0x4444,0x9f01
        .zero 14
output_lo:
        .zero 16
coef_lo:
        .uword 0xffff,0xeeee,0xaeae
        .zero 14
```

```
        .sect code,Double_precision_FIR
fir:
        /* Double precision single-sample FIR */
        LDC     0x200,mr0
        LDC     input,i0
        LDC     -1,i1      // Linear -1 addressing
//      LDC     1,i1
        LDC     coef,i2
        LDC     1,i3
        AND     a,NULL,a       // intermediate results in a:b
        AND     b,NULL,b       // set result to zero
        LDC     15,ls          // 16 stages
        LOOP    ls,firloopend-1
        LDC     output,i4

firloop:
/* Next sample from delay line -> c, next coefficients -> d */
        LDX     (i0),c1; LDY (i0),c0
        LDX     (i2),d1; LDY (i2)*,d0
/* 32x32-bit MAC with 64-bit result */
        MULUU   c0,d0
        ADD     b,p,b
        MULSS   c1,d1
        ADDC    a,p,a
        MULUS   c0,d1
        ADD     NULL,p,c
        ADD     c0,b1,b1; LDX (i0)*,c0
        LDC     1,d1
        MULSS   d1,c1          // sign extend BC(31..16)
        ADDC    a,p,a
        MULSU   c0,d0
        ADD     NULL,p,c
        ADD     c0,b1,b1
        MULSS   d1,c1          // sign extend AD(31..16)
        ADDC    a,p,a          // result after this stage in a:b

firloopend:
     /* scale result to Q31 and store */
        LSL     b,b
        LSLC    a,a
        STX     a1,(i4); STY a0,(i4)+1 // store output
endfir:
        NOP
        .end
```

8.3 Cascaded Biquad IIR Filter

This code implements a single-sample IIR filter as a cascade of second-order biquad sections. The number of sections in this example is 8.

```
#define BIQUADS 8

        .sect data_x, XData
dly:       // delay line, z(-2)'s
        .uword 0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88 // BIQUADS
coef:      // coefficients, a11, b11, a12,...
        .uword 0x100,0x200,0x300,0x400// 2*BIQUADS
        .uword 0x500,0x600,0x700,0x800
        .uword 0x1100,0x1200,0x1300,0x1400
        .uword 0x1500,0x1600,0x1700,0x1800
input:
        .uword 0x1234
output:
        .zero 1

        .sect data_y, YData
dly_1:     // delay line, z(-1)'s
        .uword 0x111,0x222,0x333,0x444 // BIQUADS
        .uword 0x555,0x666,0x777,0x888
coef_1:    //  coefficients, a21, b21, a22,...
        .uword 0x2100,0x2200,0x2300,0x2400 // 2*BIQUADS
        .uword 0x2500,0x2600,0x2700,0x2800
        .uword 0x3100,0x3200,0x3300,0x3400
        .uword 0x3500,0x3600,0x3700,0x3800

        .sect code,Biquad_IIR
iir:
        LDC     0x400,mr0
        LDC     input,i0
        AND     a0,NULL,a0; LDX (i0),a1       // input -> a
        LDC     dly,i0
        LDC     coef,i2
        LDC     1,i3
        LDC     BIQUADS-1,ls
        LOOP    ls,biquadloopend-1
        LDC     output,i4
        LDX     (i2),b0; LDY (i0),b1   // a11 -> b0, z(-1) -> b1
        MUL     b0,b1;   LDX (i0),b0;  LDY (i2)*,c0
                                       // z(-2) -> b0, a21 -> c0
        MAC     b0,c0,a; LDX (i2),c0;  LDY (i2)*,c1
                                       // b11 -> c0, b21 -> c1
        MAC     c0,b1,a; STX b1,(i0)   // z(-2) = z(-1)
        MAC     c1,b0,a; STY a1,(i0)+1 // z(-1) = t
        ADD     a,p,a   // result after this biquad to a-reg.
biquadloopend:
        STX     a1,(i4) // store output
iirend:
        .end
```

8.4 Single-Precision Matrix Multiply

C = A × B matrix multiplication, matrix dimensions: A[5][4], B[4][3], C[5][3].

Note: to test with integers, use mode 0x600 instead of 0x400 and store a0 (or the whole a-reg.) instead of a1.

```
.fract 15

/* Matrices' dimensions */
#define RA 5
#define CA 4
#define RB CA
#define CB 3
#define RC RA
#define CC CB

.sect data_x,XData
matrixA:
.uword 1,2,3,4
.uword 5,6,7,8
.uword 9,1,2,3
.uword 4,5,6,7
.uword 8,9,1,2

.sect data_y,YData
matrixB:
.uword 12,13,14
.uword 15,16,17
.uword 18,19,20
.uword 21,22,23
matrixC:
.zero 15

.sect code,Matrix_Multiply
mult:
        LDC     0x400,mr0    // saturation & fractional mode
        LDC     matrixA,i0
        LDC     1,i1
        LDC     matrixB,i2
        LDC     CB,i3
        LDC     matrixC,i4
        LDC     CA-1,c0      // loop counter for one output value
        LDC     RC,d0        // loop counter for rows
nextrow:
        LDC     CC,d1        // loop counter for columns
nextcolumn:
        AND     a,NULL,a; LDX (i0)*,b1; LDY (i2)*,b0
                             // out=0 -> a
        LOOP    c0,inloopend-1
        MUL     b0,b1;    LDX (i0)*,b1; LDY (i2)*,b0
        MAC     b0,b1,a;  LDX (i0)*,b1; LDY (i2)*,b0
                                  // out+=A[i][k]*B[k][j]
inloopend:

        LDC     -(CA+2),i1        // modify addresses before
        LDC     1-CB*(CA+2),i3    // the next round (next column)
        LDX     (i2)*,NULL; STY a1,(i4)+1 // store C[i][j]
        ADD     d1,ONES,d1; LDX (i0)*,NULL
        LDC     1,i1              // restore modifiers
        JZC     nextcolumn
        LDC     CB,i3

        LDC     CA,i1             // modify addresses before
        LDC     -CB,i3            // the next round (next row)
        ADD     d0,ONES,d0; LDX (i0)*,NULL
        LDX     (i2)*,NULL
        LDC     1,i1              // restore modifiers
        JZC     nextrow
        LDC     CB,i3

endmult:
        .end
```

8.5 Floating-Point Multiplication and Addition

Single-precision, i.e., a0 exponent (16 bits signed), a1 mantissa 1.15 format (Q15) (from -1.0 to 0.9999999...9).

f_mul multiplies a and b and puts result in c, f_add is the addition routine (c = a + b) and f_sub is the subtraction (c = a - b).

```
.fract 15

// Maximum difference in exponents
// If the difference is greater, no calculation is done
// and larger number is returned
#define _F_MAX_EXP_DIFF 16
// Stack pointer index register
#define SP              i6




.sect code,Floating_point
// Fractional mode must be set, saturation mode must be unset
// e.g. LDC 0x0000,mr0
// a * b -> c
f_mul:
        MULSS    a1,b1
        ADD      NULL,p,c       // truncate mode

        J        f_norm_res
        ADD      a0,b0,d0
/* a + b -> c */
f_add:
        SUB      a0,b0,d0; LDX (i6)+1,NULL
                                // make room to stack
        LDC      _F_MAX_EXP_DIFF,d1
        JGE      $1             // exp(a) >= exp(b)

        ADD      a,NULL,c       // swap a,b
        ADD      b,NULL,a
        ADD      c,NULL,b
        SUB      a0,b0,d0

/* exp(a) >= exp(b) */
$1:
/* check the difference in exponents, save loop hw status */
        SUB      d0,d1,d1; STX lc,(i6)+1
        STX      ls,(i6);  STY le,(i6)

        JGE      $2        // a is much bigger than b, return a
        AND      b0,NULL,b0 // zero lsp

/* shift a & b right 1 times to avoid overflow in add later */
/* loop shifts b 1 extra times */

/* shift b until it has the same exponent */
        LOOP     d0,$3-1
        SUB      a0,ONES,d0     // make result have exp(a)+1
        ASR      b,b
$3:

/* shift a 1 time, restore loop hw */
        AND      a0,NULL,a0; LDY (i6),le   // zero lsp
        ASR      a,a;        LDX (i6)-1,ls

/* a & b now have the same exp */
        J        f_norm_res
        ADD      a,b,c;      LDX (i6)-1,lc // do the add

/* return a */
$2:
        J       f_norm
        ADD     a,NULL,c

/* a - b -> c */
f_sub:
        J       f_add           // calculate a + (-b)
        SUB     NULL,b1,b1      // negate b1

/* Subroutines called by f_add, f_sub and f_mul */
// f_norm_res
// d0 exp
// c1:c0 mantissa
// norm(c) -> c
```

```
f_norm_res:
        ADD     c,NULL,c        // test mantissa for zero
        NOP
        JZC     $1              // result is not zero
        NOP
        JR
        AND     c0, NULL, c0    // force exp to zero
$1:
        ADD     c1,c1,d1        // shift left for xor
        XOR     c1,d1,d1

        NOP
        JNS     $2              // normalized, exit
        ADD     d0,ONES,d0

        J       $1
        ADD     c,c,c           // shift left

/* exit, first adjust c0 by 1 */
$2:
        JR
        SUB     d0,ONES,c0      // adjust back

f_norm:
        ADD     c0,NULL,d0
        J       f_norm_res
        AND     c0,NULL,c0

        .end
```