VS_DSP^2 USER'S MANUAL

Revision 2.6

March 8, 2001

Revision history:

Rev. 2.6	March 8, 2001	Added long-X decoding
Rev. 2.5	January 5, 2001	Core parameter table and explanation updated
Rev. 2.4	October 16, 2000	Minor corrections to the instruction coding chapter
Rev. 2.3	October 10, 2000	Minor corrections to L-flag references
Rev. 2.2	May 25, 2000	L flag added
Rev. 2.1	October 4, 1999	Instruction coding corrected
Rev. 2.0	July 9, 1999	Revisioned for VS_DSP2
Rev. 1.2	December 8, 1998	Typos corrected
Rev. 1.1	September 7, 1998	ASHL, LSHL removed
Rev. 1.0	April 27, 1998	First release

Contents

Chapter 1

Introduction

1.1 Overview

VS_DSP is a parameterized and extensible DSP core. The different manifestations of the core share a common general architecture and instruction set. The core can be used to build application specific integrated circuits (ASICs) and standard products (ASSPs). The core is available in several CMOS fabrication processes, and can be promptly ported to any normal CMOS process line.

This manual provides introduction to the general architecture, parameters, and extension attachment. The instruction set reference is also included. Separate manuals describe the associated software development tools and development boards.

1.1.1 Functional Units

The basic VS_DSP architecture is shown in Fig. 1.1. The DSP core components are described in detail in Chapters 2 – 4. This section gives an overview of the blocks shown in the architecture diagram.

The following units comprise the DSP core:

- *Datapath* — an arithmetic/logic unit (ALU) and a multiplier. Optionally a barrel shifter, a bit manipulation unit or other special computational units can be added to the core.
- *Data Address Calculation* — Two dedicated address calculation units provide addresses to data memory accesses. They enable two operands to be fetched from the data memory in parallel.
- *Program Control* — The program control fetches the instruction, generates the

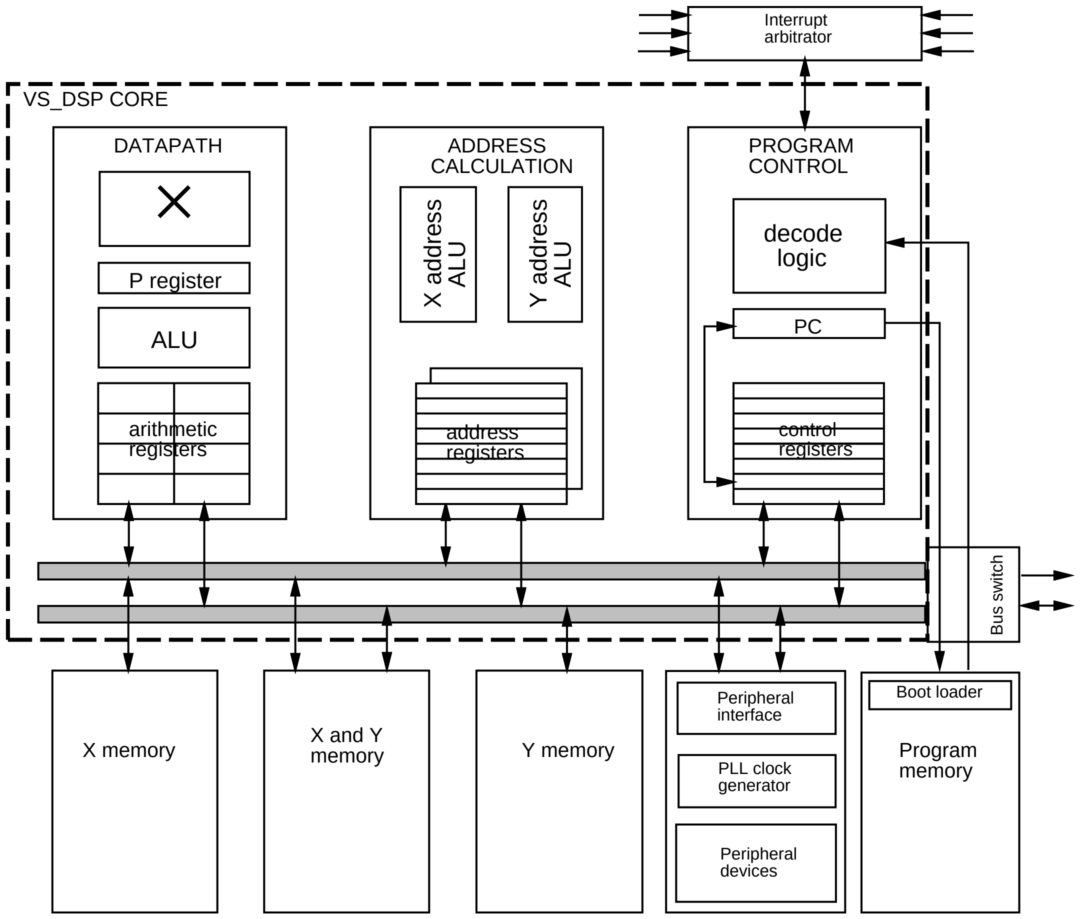


Figure 1.1: VS_DSP General Architecture.

next program address, and decodes the previously fetched instruction. The control may include optional hardware for zero-overhead loop control.

- *Buses* — The blocks are communicating over buses. There are two data buses (X and Y) with the corresponding address buses (XAB and YAB, not shown in the figure). An instruction bus (I) and a corresponding address bus (IAB) are used for code memory accesses. All the buses are available also for off-core use.

The following units may be attached to the core when building system ASICs or ASSPs, but they are not considered to be a part of the core:

- *Memory* — RAM and ROM (or any memory available in the particular fabrication process) can be placed on-chip. The amount of on-chip memory can be tailored to the application, the only practical limits being economical and technical limitations of the fabrication process (the resulting die size). Off-chip memory does not have any implications from the technology used. On the other hand, off-chip accesses typically go through a bus switch.

- *External Bus Switch* — Off-chip accesses can be multiplexed to a single address and data bus to save in the pin-count of the package. The use of flexible wait states enables the use of external memories with different timing characteristics.
- *Peripherals* — Serial and parallel interface ports, timers and also analog interfaces (analog-to-digital and/or digital-to-analog converters) may be attached, subject to technology limitations (analog precision available etc.). The peripherals are mapped to the data memory space of the core, and may be connected to interrupt lines.
- *Interrupt Arbitrator* — The core has a single interrupt line, but multiple interrupt priorities and interrupt nesting are supported by an external interrupt arbitration block. The hardware supports vectored interrupts.
- *Clock Generator* — The operating clocks of the core can be best provided by an on-chip phase-locked loop based clock generator. The control registers of the clock generator can be memory mapped, enabling the core software to control the clock frequency at runtime. This is important especially in systems requiring low power operation.
- *Boot Loader* — A specific piece of program ROM, containing boot-up code for loading software from a host computer or an external non-volatile memory.

1.1.2 Parameters

The most distinguishing feature of VS_DSP core is the use of parameters to tailor the actual implementation of the basic architecture. There are currently two basic implementations of the VS_DSP core, the full-custom version (FC) and the synthesizable VHDL version (VHDL). See Table 1.1 for the parameters, their ranges and their values for the FC and VHDL core implementations.

The parameters can be used to optimize the performance, power consumption and both core and system chip area. Especially the data word length has a major impact on the system ASIC/ASSP area because of its direct relation to the area occupied by data memories.

The parameter values are set in a hardware configuration file. The software tools can adjust their operation according to the parameter values, and the actual hardware is generated by using the very same parameters as input for either layout module generators or synthesis scripts.

1.1.3 Extensions

The basic set of instructions can be extended by adding custom hardware. The hardware can be incorporated as a part of the core, or as a coprocessor. The extension instruction

Table 1.1: VS_DSP Parameters

Parameter (*symbol*)	Range or format	FC	VHDL	Notes
Data word length (*n*)	$8 \cdots 64$ bits	16 bits	16 bits	Applies to data registers and buses
Data address length (*da*)	$8 \cdots 23$ bits	16 bits	16 bits	$da \leq n$
Program address length(*pa*)	$11 \cdots 20$ bits	16 bits	16 bits	$pa \leq n$
Multiplier input width (*m*)	$8 \cdots 64$ bits	16 bits	16 bits	$m \leq n$
Accumulator guard bits (*g*)	$0 \cdots 16$ bits	8 bits	8 bits	$g \leq n$
# of arithmetic registers	$4 \cdots 8$, step 2	8	8	Register length = *n*
# of index registers	8, 16	8	8	Register length = *da*
Modifier-only	binary (0,1)	0	0	Forbids the use of odd registers for addresses
Loop hardware	0, 1, N (N = $2 \cdots 8$)	1	1	Levels of loop hardware
Addressing mode mask	three-bit word ($0 \cdots 7$)	1	7	Enable bits for modulo, bitrev and (reserved) modes
Modemask	*pa* bits word	0x077f	0x7ff	Selects active mode bits/flags
Roundmode	0, 1, 2, 3	0	0	Selects the rounding mode implemented 0 = truncate, 1 = round 2 = convergent 0 3 = convergent 1

mechanisms allow hardware-software trade-offs to be made in the application development. They also have a major impact on the power consumption by enabling a lower clock frequency to be used.

There are hooks for the following types of extensions:

- *New Operation Modes* — The operation of existing instructions can be fine-tuned by generating new operation modes which can be chosen by setting or clearing some of the (reserved) mode bits. In a similar manner, some new flags and conditions can be added.
- *New Pipeline Register Modifications*. The multiply-accumulate operation is pipelined, and it is possible to invent new ways of shifting, masking or modifying the pipeline register content when using it as an ALU (accumulation) operand.
- *New Addressing Modes*. There remains a reserved code in the addressing mode field. More addressing modes can thus be added to the architecture.
- *New Arithmetic-Logic Instructions*. There are reserved operation codes for including new ALU operations using the same register set as the basic core. This is the

way to add, e.g., barrel shifter support or special bit operations to the instruction set.

- *New Conditional Instructions*. E.g., conditional parallel moves can be coded by employing reserved bits in some instructions. This enables, e.g., conditional storage of data or pointer from registers to memory. Also ALU operations can be made conditional, if necessary.
- *Custom Logic Blocks in Parallel with the Core*. There are reserved opcodes for the inclusion of almost full-length extension instructions. A custom logic block (co-processor) can be placed in parallel with the core, and the operation is still controlled directly by the core control. This kind of instruction can also use a different set of data registers. The custom logic can be, e.g., a bit manipulation unit or a divider.
- *Memory-Mapped Co-Processors*. In addition to the tightly coupled custom logic above, the co-operating device can be also memory mapped. This kind of extension does not necessarily require a custom instruction, it can also be controlled simply by storing to and loading from the specific address. The co-processor can be almost anything, e.g., a hardware filter section, some iterative device, interface logic, or a DMA-coprocessor. The main issue in this kind of extension is that it does not need to (or cannot be) controlled by the core software anymore.

We strongly recommend to contact the core vendor before committing to physical design of custom extensions.

1.1.4 Instruction Set

The basic instruction set is common for the different instantiations of VS_DSP core, such that upward compatibility is retained when switching from a simpler version to a more versatile one. This includes all the data word lengths, optional hardware (and modes and instructions) added by the parameters, and also custom extensions. The assembly code can be directly reused, sometimes even the compiled binary code. However, the performance may only be improved by changing the critical parts of the code to use the more advanced features.

In another direction ("downward"), the stripped operations have to be compensated by replacing the extension instructions by corresponding software macros. Special care has to be taken also when decreasing the width of some address or data word length.

On C language level the compatibility is not an issue, since the tools can retarget the code to the correct core version by recompilation.

VS_DSP has a reasonable level of parallelism. The operation is pipelined in three stages (fetch – decode – execute). Within a single processor clock cycle, the core can:

- Generate the next program address
- Fetch an instruction
- Decode the previously fetched instruction
- Perform up to two data moves
- Post-modify up to two data pointers
- Perform a computation on register data

Chapter 6 details the instruction set.

1.2 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:

- *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.
- *Linker* — The Linker links separately assembled modules.
- *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.
- *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.
- *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.
- *C Compiler* — The C Compiler reads ANSI C based source code (interleaved with some optimization constructs) and produces VS_DSP code ready to be assembled.

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.

1.3 Organization of This Manual

The rest of this manual is organized as follows.

- Chapter 2 describes the *datapath* in detail.
- Chapter 3 explains the addressing modes and *data address calculation unit* functionality.
- Chapter 4 describes the *program control* unit.
- Chapter 5 illustrates the *control flow* of the core.
- Chapter 6 is the *instruction set reference*, with the programming model, flags and mode bits, and a detailed description of each instruction in the basic instruction set.
- Chapter 7 describes the *instruction coding* field by field in different instruction types.
- Chapter 8 gives assembly language *software examples*.

Chapter 2

Datapath

2.1 Overview

The VS_DSP datapath architecture is depicted in Fig. 2.1.

The datapath operates with the principle of one cycle per instruction (from register to register). The $2n+g$-bit ALU implements the arithmetic (ABS, ADD, ADDC, SUB, SUBC, MAC, MSU) and logic (AND, ASR, LSL, LSLC, LSR, LSRC, NOT, OR, XOR) instructions. MUL is implemented by the separate multiplier.

The ALU has up to eight n-bit arithmetic registers A0, A1, B0, . . . , D0, D1. Optional guard bit registers A2, . . ., D2 are available. These can be combined to form $2n + g$-bit accumulators A, B, C and D. Note that C and D are optional.

The multiplier is a $m \times m$-bit signed/unsigned integer/fractional saturating/unsaturating multiplier. Multiplier inputs are A0, A1, B0, . . . , D0, D1. Multiplier output goes to a $2n$-bit pipeline register P, which can be used as an ALU operand in ADD and SUB instruction to form a MAC or MSU operation.

The data word length *n* is a parameter, and the multiplier word length *m* is another independent parameter. Two data buses (width = *n*) connect the datapath to off-core memories.

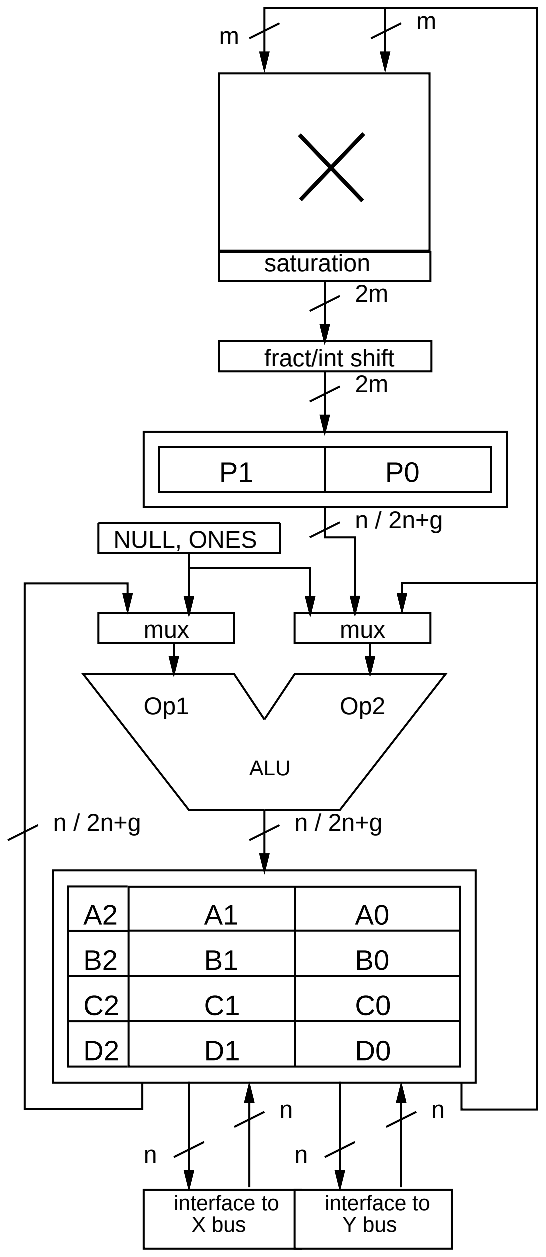


Figure 2.1: VS_DSP datapath.

2.2 Arithmetic

The datapath operates by default on signed (two's complement) numbers. The multiplier has separate modes for integer and fractional multiply, selected by a bit in the mode register (see Chapter 4). The multiplier can also operate on unsigned/signed, signed/unsigned, and unsigned/unsigned operands. The type of the operands is declared in the multiply instruction, with the signed/signed operand pair as the default.

The logical operations simply consider the operands as bit patterns.

There is also a saturation mode for the multiplier and ALU, selected by a bit in the mode register (see Chapter 4). In the saturation mode, the result is interpreted as a signed number, and saturated accordingly.

2.3 Flags and Mode Bits

The processor mode/status register includes the 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
Z	zero flag
N	negative flag
V	overflow flag
E	extension flag
C	carry flag

In the saturation mode, the ALU result in arithmetic operations is saturated to the maximum positive or negative value in case the operation creates an over/underflowing result. The integer/fractional mode controls the shifter after the multiplier to output the result in a correctly aligned format. The rounding mode is implementation dependent.

Arithmetic flags are evaluated after an arithmetic operation.

2.4 ALU

The functions of the ALU and the multiplier are listed below.

Multiplying, adding and subtracting		
`ADD`	Op1, Op2, Result	Add operands
`ADDC`	Op1, Op2, Result	Add operands with carry-in bit
`SUB`	Op1, Op2, Result	Subtract operands (order can be chosen)
`SUBC`	Op1, Op2, Result	Subtract operands with borrow-in (order can be chosen)
`MUL`	Op2, Op2, Result	Multiply operands (multiplier)
`MAC`	Op1, Op2, Result	MAC operation (ALU and multiplier)
`MSU`	Op1, Op2, Result	MSU operation (ALU and multiplier)
Special operations with add/sub		
`SUB`	NULL, Op2, Result	Negate Op2 (two's complement)
`SUB`	Op1, ONES, Result	Increment Op1
`ADD`	Op1, ONES, Result	Decrement Op1
`ADD`	Op1, NULL, Result	Pass Op1 unchanged
`ADD`	NULL, NULL, Result	Clear result register to zero
`ADD`	NULL, ONES, Result	Set result register to –1
`SUB`	NULL, ONES, Result	Set result register to +1
Logical		
`AND`	Op1, Op2, Result	Logical AND of Op1 and Op2
`OR`	Op1, Op2, Result	Logical OR of Op1 and Op2
`XOR`	Op1, Op2, Result	Logical XOR of Op1 and Op2
`NOT`	Op2, Result	Logical Not of Op2 (one's complement)
Shifts		
`ASR`	Op2, Result	Arithmetic 1-bit shift right
`LSR`	Op2, Result	Logical 1-bit shift right
`LSRC`	Op2, Result	Logical 1-bit shift right with carry-in bit
`LSL`	Op2, Result	Logical 1-bit shift left
`LSLC`	Op2, Result	Logical 1-bit shift left with carry-in bit
Miscellaneous		
`ABS`	Op2, Result	Absolute value of Op2 (conditional negate)
`RESP`	Op1, Op2	Restore pipeline register from Op1 and Op2

The ALU can calculate either $2n + g$ or n-bit operations. The selection of operation width is made depending on the operands; if one of the operands is $2n + g$ bits wide, the operation is $2n + g$ bits and the result is stored to a $2n + g$-bit register. If both operands are n bits, the operation and result are also n bits and the result is stored to a n bit register.

The n-bit operands are `A0`, `A1`, `B0`, $\ldots$, `D0`, `D1`. The pseudo-registers `NULL` and `ONES` are also available and contain all zeros and all ones, respectively. `NULL` and `ONES` are considered to be n-bit registers for the purpose of determining the result width (see above).

The $2n + g$-bit operands are `A`, `B`, `C` and `D`. `P` is available as operand2. The register `A` is formed by concatenating `A2:A1:A0`. `A0` is the lsb part. For $2n + g$-bit calculations, also n-bit registers are available as operands. In this case, the register is used as the middle part of the operand. The lsb end is padded with n zeros and the sign is extended to the (optional) guard bits. For example, if register `A0` is used as $2n + g$-bit operand, the operand is `xx:A0:0000` (xx means sign extension bits).

The result of $2n + g$-bit operation is either `A`, `B`, `C` or `D`. The result of n-bit operation is `A0`, `A1`, `B0`, . . . , `D0`, `D1`. The ALU (optionally) produces negative, carry, overflow, zero and extension (guard bits in use) flags.

2.5 Multiplier

The multiplier is a $n \times n$ signed/unsigned integer/fractional saturating/unsaturating multiplier.

Both inputs can be interpreted either as signed or unsigned numbers, to facilitate multiprecision operations. The integer/fractional mode bit controls the 1-bit left shift of the result (fractional mode) when it is written to `P`. In fractional signed$\times$signed multiplication, saturation is optionally (in saturation mode) included so that result of `0x8000` $\times$ `0x8000` is `0x7fffffff` (NOT `0x7ffffffe`!!). The `P` register length is $2n$ 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. The restoring will take place by executing the `RESP` instruction.

2.6 Guard bit registers

Optional guard bit registers behave as an extension of registers `A1`, `B1`, `C1` and `D1`. The following describes `A2`. `B2`, `C2` and `D2` function similarily, but they refer to `B1`, `C1` and `D1`, respectively, instead of `A1`.

Whenever an arithmetic register `A1` is written to, either from data buses or from ALU, `A2` is written to with the sign extension of `A1`. The only exception is when an ALU operates in $2n + g$-bit mode and ALU result is written to `A2`:`A1`:`A0`. In this case the uppermost g bits of ALU result are written to `A2`.

Note that if ALU operates in n-bit mode and `A1` is the result register, the sign extension will be written to `A2`. If `A0` is the result register, `A2` is not written to.

Chapter 3

Data Address Generator

3.1 Architecture Overview

Data Address Generator performs data address calculations and drives data address buses. It contains index registers `I0` $\cdots$ `I7`.

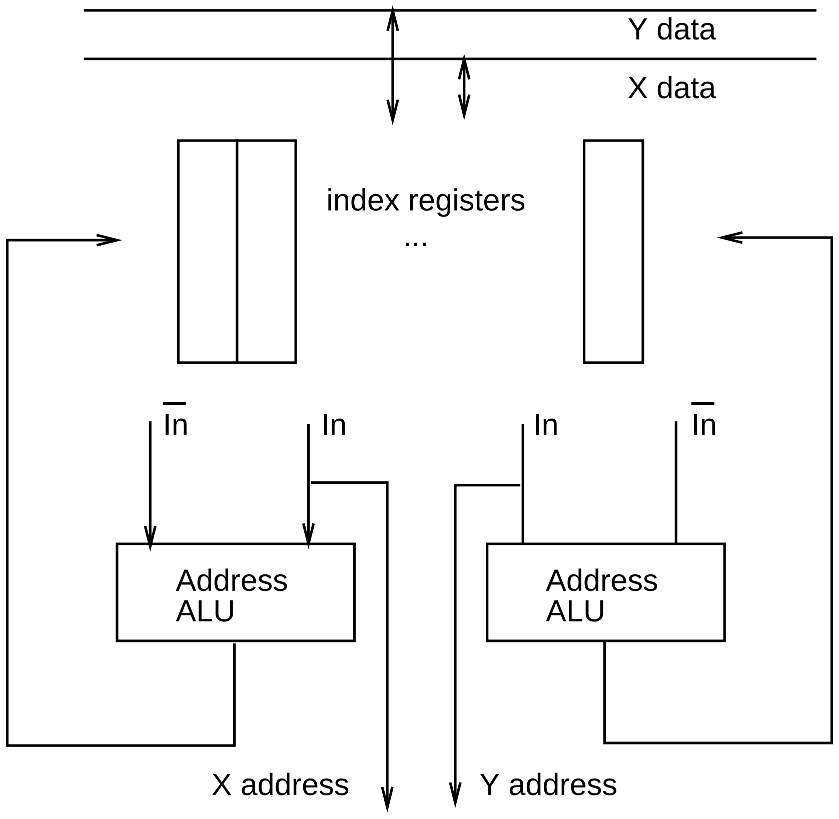


Figure 3.1: Data Address Generator overview

Data Address Generator contains two identical parallel address ALU units and is capable of providing two independent data addresses on each cycle. Two address registers used in addressing can be post-modified.

3.1.1 Index Register File

Index register file contains index registers `I0` $\cdots$ `I7`. Two index registers can be accessed to/from X and Y data bus each cycle.

Index registers are used to form X or Y memory addresses. The registers are accessed in pairs. Each register, designated `In`, has a corresponding register pair, designated $\overline{\texttt{In}}$. $\overline{\texttt{In}}$ is the index register the number of which is generated by inverting the LSB bit of the number of register `In`. For example, if `I3` is `In`, then `I2` is $\overline{\texttt{In}}$.

To form X/Y addresses, `In` is used as the address. $\overline{\texttt{In}}$ then specifies the post-modification address mode, if any. To form long X addresses (2×dataaddress), $\overline{\texttt{In}}$ and `In` are concatenated to form the long address.

Two index registers can be read for X/Y addresses and X/Y Address ALU index inputs, designated `In` in Fig. 3.1. Two index registers can be read for X/Y Address ALU modifier inputs, designated $\overline{\texttt{In}}$ in Fig. 3.1. The $\overline{\texttt{In}}$ is the register pair of `In`.

`In` registers used for X/Y Address ALU index inputs can be updated with address ALU outputs.

3.1.2 Address ALU

Address ALU can calculate three types of updated addresses, which are: linear post-inc/dec, modulo post-inc/dec and bit reversal.

Address ALU contains linear and bit reverse adders for calculating linear and bit reversed addresses. These adders are otherwise identical, but in bit reverse adder carry propagates towards LSB.

Modulo logic is capable of restoring calculated linear addresses to remain within a buffer if modulo addressing is used. The buffer length does not need to be a power of two.

The modulo and bit reverse addressing modes are separately enabled by the *Addressing mode mask* parameter of the core (see section 7.6). It is also possible to extend the addressing modes by an additional mode.

3.1.3 Flags

`MR0` is the processor mode/status register. The bits affected by the address calculation are:

Bit/flag	Meaning
X	index X flag
Y	index Y flag

Index flags are evaluated whenever an index ALU is enabled. If the ALU uses modulo addressing, the flag is set if the modulo wrap-around is done. If the ALU uses addressing mode other than modulo, the flag is the sign bit of the index register used (`In`).

3.2 Post-modification Modes

Addressing has two post modification modes specified in the instruction, post-modification by -7$\cdots$+7 or post-modification by $\overline{\texttt{In}}$. The coding is summarized below.

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

If $\overline{\texttt{In}}$ is used to specify the post modification mode, 3 MSBs of $\overline{\texttt{In}}$ are used to specify the post modification mode as follows:

Binary code	Modification
`000`	`In` = (`In`+`m`) (`m` positive)
`001`	`In` = (reserved)
`010`	`In` = (`In`+2%m) (optional)
`011`	`In` = (`In`−2%m) (optional)
`100`	`In` = (`In`+ + %`m`) (optional)
`101`	`In` = (`In`− − %`m`) (optional)
`110`	`In` = (`In`+`m`) bit reverse (optional)
`111`	`In` = (`In`+`m`) (`m` negative)

3.2.1 Linear Post-increment/decrement

Linear post-inc/dec can be an immediate pppp (-7 $\cdots$ +7) modification or modification `In` + $\overline{\texttt{In}}$.

Post-modifier mod is either immediate pppp (sign extended):

for each i (AW–1 $\cdots$ 3): mod[i] = pppp[3]
for each i (2 $\cdots$ 0): mod[i] = pppp[i]

or modifier specified by $\overline{\texttt{In}}$ (sign extended):

for each i (AW–1 $\cdots$ AW–3): mod[i] = $\overline{\texttt{In}}$[AW–3]
for each i (AW–4 $\cdots$ 0): mod[i] = $\overline{\texttt{In}}$[i]

Updated value for `In` is:

$$\texttt{In}=\texttt{In} + \text{mod}$$

Note that in the case of a negative modifier, $\overline{\texttt{In}}$ should contain the desired modifier *m* in two's complement format.

Example (0x61–0x1f=0x42 using `In` - $\overline{\texttt{In}}$):

	7 … 0	
`In` =	0 1 1 0 0 0 0 1	0x61 (97)
$\overline{\texttt{In}}$ =	1 1 1 │ 0 0 0 0 1	
updated `In` =	0 1 0 0 0 0 1 0	0x42 (66)

3.2.2 Modulo Post-increment/decrement (Optional)

In modulo addressing, calculated addresses are kept within a buffer whose length is M. The lower boundary of the buffer must be an integer multiple of 2^k, where $2^k \geq M$.

To use modulo addressing, `In` must be within the buffer, i.e. AW–k MSB bits of `In` must equal the corresponding bits of the lower boundary. AW–3 LSB bits of $\overline{\texttt{In}}$ should contain the value M–1. AW means the data address width.

To calculate updated address, the following steps are taken:

1) Determine the lower boundary of the buffer

Starting from $\overline{\texttt{In}}$[AW–3], find the uppermost 1 bit in $\overline{\texttt{In}}$. Let k be the bit position of the uppermost 1 bit. The lower boundary is:

for each i (AW$\cdots$ k+1):	lower[i] = `In`[i]
for each i (k $\cdots$ 0):	lower[i] = 0

2) Determine the upper boundary of the buffer

Upper boundary is lower boundary plus the length of the buffer, which is contained in $\overline{\texttt{In}}$. The upper 3 MSBs of $\overline{\texttt{In}}$ are not part of the buffer length.

$$\text{length} = \overline{\texttt{In}}[\text{AW–3} \cdots 0] + 1$$

$$\text{upper} = \text{lower} + \text{length} - 1$$

3) Calculate linear address

Calculate linear modified address depending on MSBs of $\overline{\texttt{In}}$.

$\overline{\mathtt{In}}$[AW$\cdots$ AW–3]	linear address
`010`	lin = `In`$+2$
`011`	lin = `In`-2
`100`	lin = `In`$+1$
`101`	lin = `In`-1

4) Restore to buffer if needed

If linear address is outside the buffer, restore it to buffer. If linear address is already inside the buffer, nothing needs to be done.

If lower $\leq$ lin $\leq$ upper,	`In` = lin
If lin $<$ lower,	`In` = lin + length
If lin $>$ upper,	`In` = lin – length

Example (13-point ring buffer 0x20 $\cdots$ 0x2c, 0x2c+0x2=0x21):

	7 4	3 0	
`In` =	0 0 1 0	1 1 0 0	0x2c (44)
$\overline{\mathtt{In}}$ =	0 1 0	0 1 1 0 0	0x4c (64+12=76)
1) lower =	0 0 1 0	0 0 0 0	0x20
2) upper =	0 0 1 0	1 1 0 0	0x2c
3) linear =	0 0 1 0	1 1 1 0	0x2e
updated `In` =	0 0 1 0	0 0 0 1	0x21 (33)

3.2.3 Bit Reversal (Optional)

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 must be a multiple of 2^k.

To use bit reversal addressing, `In` must be within the buffer, i.e. AW–k MSB bits of `In` must equal the corresponding bits of the lower boundary. 3 MSBs of $\overline{\mathtt{In}}$ should contain `110` to select bit reversal addressing. LSBs of $\overline{\mathtt{In}}$ should contain the value 2^{k-1}.

`In` = `In` + $\overline{\mathtt{In}}$[AW–3 $\cdots$ 0] (propagate carry towards LSB)

Example (16-point (k = 4) FFT in buffer 0x50 $\cdots$ 0x5f):

	7 4	3 0	
`In` =	0 1 0 1	1 1 0 0	0x5c (92)
$\overline{\mathtt{In}}$ =	1 1 0	0 1 0 0 0	0xc8 (192+2^{4-1}=200)
updated `In` =	0 1 0 1	0 0 1 0	0x52 (82)

Chapter 4

Program control

4.1 Architecture Overview

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`.

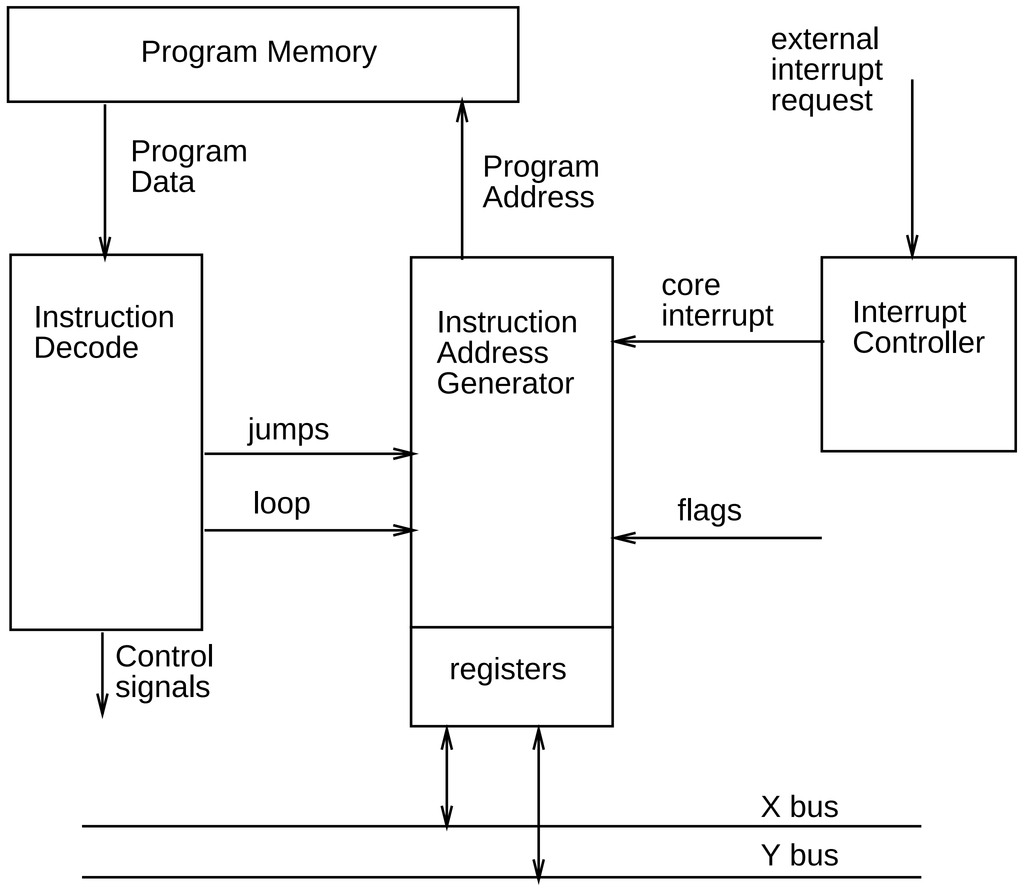


Figure 4.1: Program Control overview

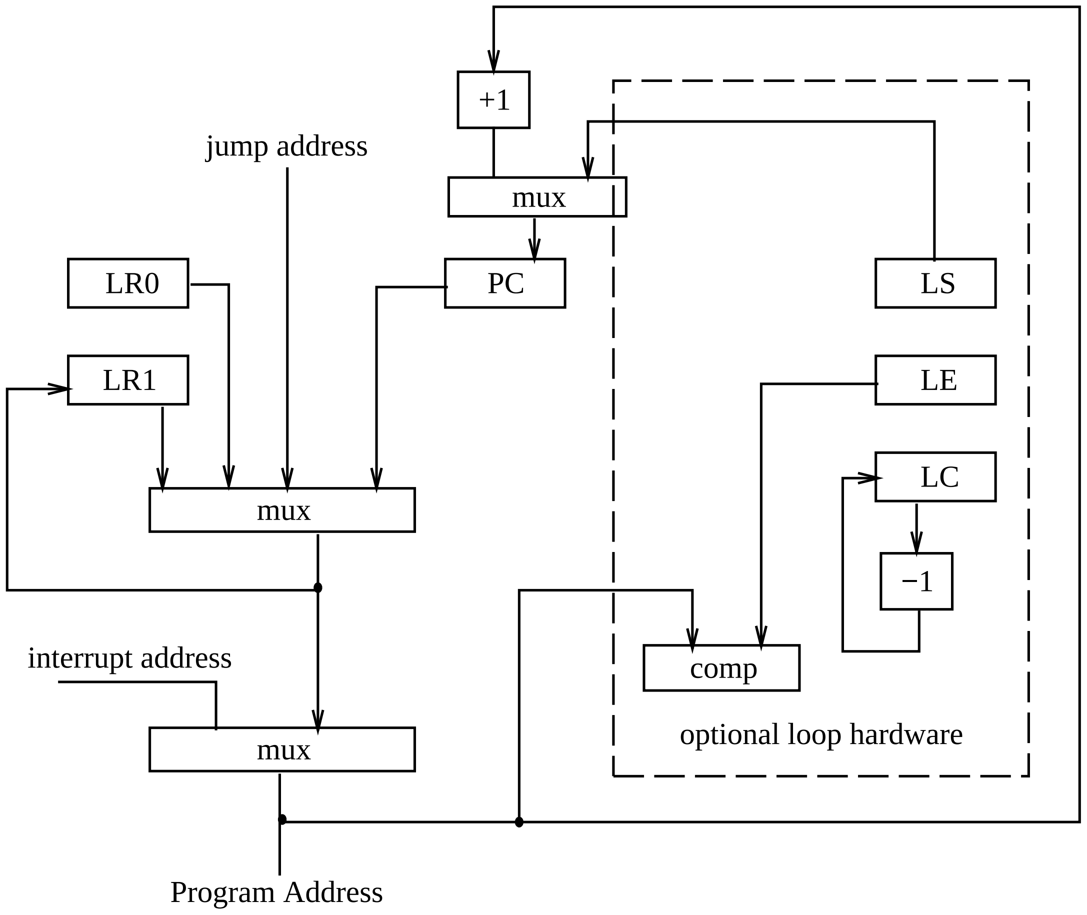


Figure 4.2: Instruction Address Generator overview

Mode register `MR0` holds the mode and flag bits, and `MR1` is used as a temporary mode register while transferring to interrupt service. Optional loop control has three registers, `LS`, `LE` and `LC`. Program counter is not directly accessible.

Program Control unit has three components, which are shown in Fig. 4.1. The components Instruction Decode, Instruction Address Generator and Interrupt Controller are described in the following subsections.

4.1.1 Instruction Decode

Instruction Decode reads instructions from Instruction Data Bus and decodes them.

4.1.2 Instruction Address Generator

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.

Fig. 4.2 shows the overall structure of the Instruction Address Generator. Connections from registers to data buses are not shown.

The fetch address is determined as follows:

- On Interrupt cycle #2, interrupt vector I0 is the fetch address.
- If instruction in execute phase is `Jcc` and the condition is true, jump address is the fetch address.
- If instruction in execute phase is `JRcc` and the condition is true, `LR0` contains the fetch address.
- If instruction in execute phase is `RETI`, `LR1` contains the fetch address.
- In all other cases, `PC` holds the fetch address.

Instruction Address Generator contains the optional loop hardware. Behavior of Instruction Address Generator is further described in Chapter 5.

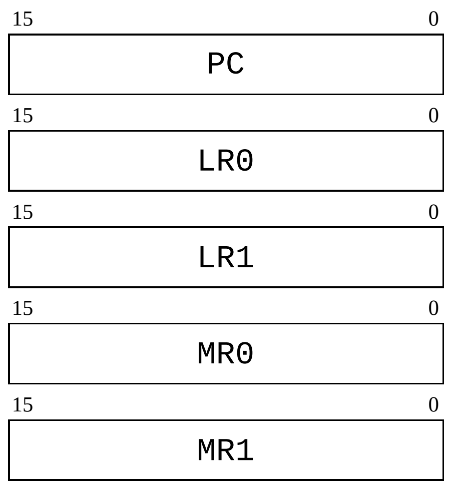
To achieve larger than *pa*-bit instruction address space, two page registers are used. `IPR0` holds the uppermost part of the program address. `IPR0` and `PC` together determine the program address.

4.1.3 Interrupt Control

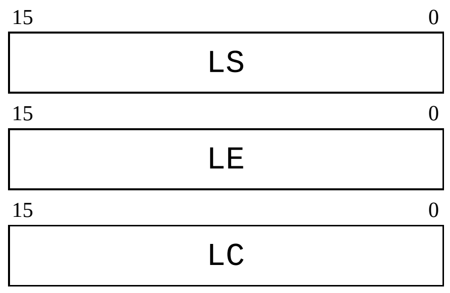
Interrupt Controller processes interrupts. It implements the interrupt state machine described in Fig. 5.7, section 5.3. Interrupt Controller receives external interrupt and drives interrupt fetch signal to Instruction Address Generator. Interrupt Controller makes sure that previous interrupt has been processed before new interrupt request is presented to Instruction Address Generator.

4.2 Programming Model

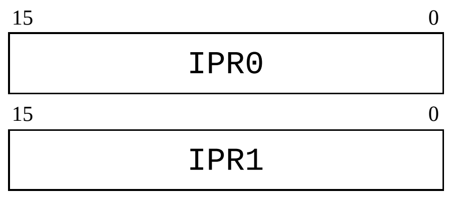
Program control unit has the following registers:



Optional loop registers:



Page registers:



The width of registers is Program address width *pa*. Above it is assumed that *pa* is 16.

4.2.1 `PC`

`PC` is the program counter. It is not directly accessible by the programmer. `PC` is loaded with the program address bus+1 value on all cycles except when new loop round starts. In this case `PC` is loaded with `LS`.

In reset, `PC` is copied to `LR1`.

In instruction fetches, program address bus is driven either from `PC`, `LR0`, `LR1`, decoded instruction jump target address, reset vector address or interrupt vector address.

4.2.2 `LR0`

`LR0` is used in indirect jumps. `JRcc` instruction causes instruction to be fetched from `LR0` address instead of `PC` address, if condition `cc` is true. `LR0` can be used to form subroutines by saving the return address to `LR0` and executing `JRcc` at the end of the subroutine. If nested subroutines are needed, `LR0` must be saved and restored by the calling subroutine.

4.2.3 `LR1`

`LR1` is used in interrupt returns. `RETI` instruction causes instruction to be fetched from `LR1` address instead of `PC` address. `PC` is copied to `LR1` on interrupt cycle #1 and possibly on interrupt cycle #2 (see section 5.3.1 for description of interrupt mechanism).

If nested interrupts are needed, `LR1` must be saved and restored by the interrupt service routine. See section 5.3.2 for the save and restore routines.

4.2.4 `MR0`

`MR0` is the processor mode/status register. The length of the register is *pa*, so the minimum length is 11 and maximum 20 bits. It includes the mode bits and status flags. The bits are (here in the case *pa* = 16):

15							8	7							0
d	d	d	d	d	S	I	R	L	X	Y	Z	N	V	E	C
mode bits								flags							

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

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

The mode bits and flags are described in more detail in section 6.2.

4.2.5 `MR1`

`MR1` is the interrupt register of `MR0`. In interrupts, `MR0` is copied to `MR1` at interrupt cycle #4 when L-flag is set in `MR0`. `MR1` must be saved in the start of the interrupt.

4.2.6 `LS` (optional)

`LS` holds the loop start address. `LOOP` instruction copies instruction fetch address to `LS`. When new loop round starts, `PC` is loaded with `LS` instead of instruction fetch address+1.

4.2.7 `IPR0`

`IPR0` is the instruction page register. It holds the upper *pa* bits of instruction address.

There are limitations on the use of `IPR0`. It can be accessed only as a source operand in `MVX` or `STX` instruction. `IPR0` can be changed by `JRcc` or `JMPI` instruction.

4.2.8 `IPR1`

`IPR1` is the interrupt register of `IPR0`. In interrupts, `IPR0` is copied to `IPR1` at interrupt cycle #2.

There are limitations on the use of `IPR1`. It can be accessed only as a source operand in `MVX` or `STX` instruction. There is no way to write to `IPR1`, except the interrupt mechanism.

4.2.9 `LE` (optional)

`LE` holds the loop end address. `LOOP` instruction loads `LE` with loop end address specified in the `LOOP` instruction. When instruction fetch occurs from `LE` address and L-flag is not set, new loop round starts if $\texttt{LC} \neq 0$.

`LE` is initiated with all ones in system reset.

4.2.10 LC (optional)

LC holds the loop count. LOOP instruction loads LC from specified register. When instruction fetch occurs from LE address, LC is tested for being equal to 0. If $\mathtt{LE} \neq 0$, it is decremented by one, new loop round starts and LS is copied to PC. If $\mathtt{LC} = 0$, nothing special happens and PC is loaded with instruction fetch address+1 as usual.

Chapter 5

Control Flow

The control flow behavior 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 behavior 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 the processor flags and their combinations. The flags that are used in the jump condition evaluation must be unaffected in the cycle before the jump instruction is executed, i.e., the instruction immediately before the jump instruction must not change the jump condition flags. Other flags can be modified.

Fig. 5.1 shows the situation where instruction #2 is a change-of-flow instruction (`J`, `Jcc`, `JRcc`, `CALLcc` or `RETI`). Instruction #3 is in the delay slot and is always executed. When jump instruction executes (cycle 4), program address is driven either from jump target register, `LR0` or `LR1` (jump is taken) or from `PC` (jump is not taken). D2 denotes this address. `PC` is loaded with D2+1 on the next cycle.

Cycle	1	2	3	4	5	6	7	8	9
Fetch	#1	#2	#3	D2	D2+1	· · ·			
Decode		#1	#2	#3	D2	D2+1	· · ·		
Execute			#1	#2	#3	D2	D2+1	· · ·	
PC	#1	#2	#3	#4	D2+1	· · ·			

Figure 5.1: Jump execution

Cycle	1	2	3	4	5	6	7	8	9
Fetch	#1	#2	#3	LS	LS+1	· · ·			
Decode		#1	#2	#3	LS	LS+1	· · ·		
Execute			#1	LOOP	#3	LS	LS+1	· · ·	
PC	#1	#2	#3	LS	LS+1	· · ·			
LS	· · ·				LS				
LE	· · ·				LE				
LC	· · ·				LC				

Figure 5.2: Loop start.

Cycle	1	2	3	4	5	6	7	8	9
Fetch	#1	#2	#3	LS	LS	· · ·			
Decode		#1	#2	#3	LS	LS	· · ·		
Execute			#1	LOOP	#3	LS	LS	· · ·	
PC	#1	#2	#3	LS	LS	· · ·			
LS	· · ·				LS				
LE	· · ·				LE				
LC	· · ·				n–1	n–2	· · ·		

Figure 5.3: Single instruction loop start.

Cycle	1	2	3	4	5	6	7	8	9
Fetch	#1	#2	#3	LS	LE+1	· · ·			
Decode		#1	#2	#3	LS	LE+1	· · ·		
Execute			#1	LOOP	#3	LS	LE+1	· · ·	
PC	#1	#2	#3	LS	LE+1	· · ·			
LS	· · ·				LS				
LE	· · ·				LE				
LC	· · ·				0				

Figure 5.4: Single instruction, single round loop.

Cycle	1	2	3	4	5	6	7	8	9
Fetch	LE–2	LE–1	LE	LS	LS+1	· · ·			
Decode		LE–2	LE–1	LE	LS	LS+1	· · ·		
Execute			LE–2	LE–1	LE	LS	LS+1	· · ·	
PC	LE–2	LE–1	LE	LS	LS+1	· · ·			
LC	n			n–1					

Figure 5.5: Loop end when LC $\neq 0$.

5.2 Loops (Optional)

Loop mechanism is optional. 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. Fig. 5.2 and Fig. 5.3 illustrate start of loop. Loop can also be initiated by setting LS, LE and LC to appropriate values.

When program 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, nothing special happens and the loop ends. Fig. 5.5 and Fig. 5.6 illustrate these loop end situations.

Cycle	1	2	3	4	5	6	7	8	9
Fetch	LE–2	LE–1	LE	LE+1	LE+2	· · ·			
Decode		LE–2	LE–1	LE	LE+1	LE+2	· · ·		
Execute			LE–2	LE–1	LE	LE+1	LE+2	· · ·	
PC	LE–2	LE–1	LE	LE+1	LE+2	· · ·			
LC	0								

Figure 5.6: Loop end when `LC` = 0.

5.3 Interrupts

Interrupts are vectored using a jump table. The external interrupt arbiter supplies an interrupt vector. The vector is an address in the range 0x20. . .0x3f. These addresses 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 end with a `RETI` or a `JRcc`.

If nested interrupts are needed, interrupt program must save `LR1` and restore it before returning from interrupt. Saving and restoring instructions must be the routines specified in section 5.3.2.

5.3.1 Interrupt Mechanism

Fig. 5.7 shows interrupt state machine with 5 interrupt cycles. State transitions are done at the end of each instruction cycle. The actions in each state are described in the right hand side of that state.

On interrupt cycle #1, fetch address is copied to `LR1`. Instruction fetch is done from the normal fetch address.

On interrupt cycle #2, first interrupt instruction (I0) is fetched. `IPR0` is copied to `IPR1`. On this cycle we also decide whether instruction fetched on interrupt cycle #1 will be canceled or not. The L-flag is set in MR0 register.

Instruction fetched on interrupt cycle #1 must be canceled, unless

- instruction fetched on the cycle before interrupt cycle #1 is a change-of-flow instruction, or
- instruction fetch on interrupt cycle #1 occurs from loop end address `LE`.

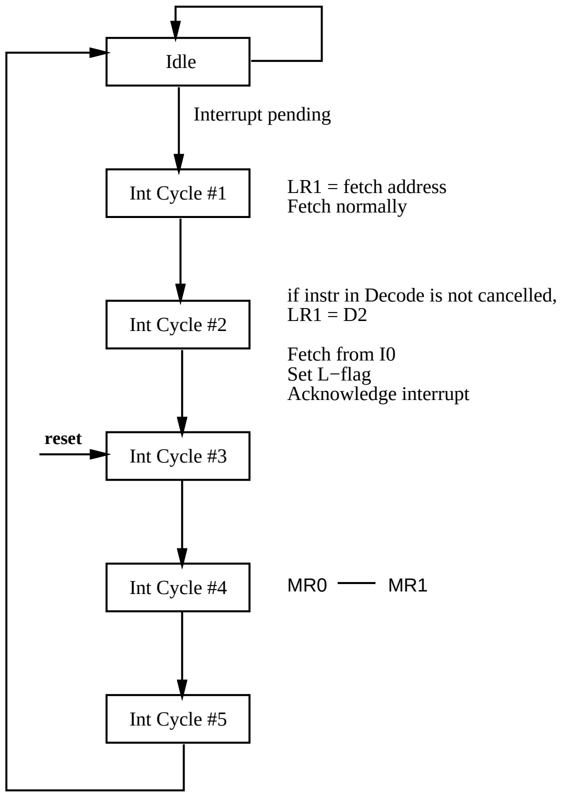


Figure 5.7: Interrupt cycle control flow.

If instruction fetched on interrupt cycle #1 is not canceled, on interrupt cycle #2 `LR1` is loaded with the destination address of instruction that is in execute stage on interrupt cycle #2. Destination address is the address that would be fetched if the interrupt fetch would not have occurred (e.g. jump target address or next linear address).

During interrupt cycle #4 `MR0` is copied into `MR1`.

From interrupts' point of view, change-of-flow instructions are `J`, `Jcc`, `JRcc`, `RETI` and `LOOP`.

Fig. 5.8 shows an example when canceling occurs and Fig. 5.9 when it does not occur. In figure Fig. 5.8, instruction #2 is a change-of-flow instruction, or instruction #3 is fetched from loop end address `LE`. Instruction #3 must be executed before interrupt is serviced. If instruction #2 is a change-of-flow instruction, destination address (denoted by D2) is the jump target address or next linear address if the jump is not taken. If instruction #3 is fetched from loop end address `LE`, D2 is the loop start address `LS` or next linear address if the loop ends. In all cases D2 is copied to `LR1` and will be the interrupt's return address.

If there is the possibility that instruction #3 is a change-of-flow instruction, instruction in its delay slot (#4) would not be fetched because of I0 fetch. To prevent this, instruction #3 must be canceled. Fig. 5.9 illustrates this situation. `LR1` is not updated again on

Cycle	1	2	3	4	5	6	7	8	9
Int Cycle			1	2	3	4	5		
Fetch	#1	#2	#3	I0	I1	I2	I3		
Decode		#1	#2	#3	I0	I1	I2	I3	
Execute			#1	#2	#3	I0	I1	I2	I3
PC	#1	#2	#3	#4	I1	I2	I3	· · ·	
LR1	· · ·			#3	D2				
MR1	· · ·						`MR0`		

Figure 5.8: Interrupt when instruction #3 is not canceled.

Cycle	1	2	3	4	5	6	7	8	9
Int Cycle			1	2	3	4	5		
Fetch	#1	#2	#3	I0	I1	I2	I3		
Decode		#1	#2	`NOP`	I0	I1	I2	I3	
Execute			#1	#2	`NOP`	I0	I1	I2	I3
PC	#1	#2	#3	#4	I1	I2	I3	· · ·	
LR1	· · ·			#3					
MR1	· · ·						`MR0`		

Figure 5.9: Interrupt when instruction #3 is canceled.

interrupt cycle #2, so it will point to address of instruction #3.

In case of nested interrupts, old value of `LR1` is lost when first interrupt instruction (I0) is fetched. In interrupt routine, second instruction (I1) saves `LR1`, so another interrupt fetch can start when I1 has been executed (cycle 8) in Fig. 5.9.

Next I0 can be fetched on cycle #8 at the earliest to be sure `LR1` is saved properly. In interrupts, I0, I1 and I2 are always executed before another interrupt can occur. If I2 is a change-of-flow instruction, also I3 is executed.

Loading of `LR1` must override automatic `LR1` load by interrupt mechanism to guarantee proper interrupt ending (case when instruction #1 above is `LR1` load).

5.3.2 Interrupt Routines

A typical interrupt jump table looks like the following:

```
.org 0x20
```

Rst	active			released					
Int Cycle	3	3	3	3	4	5			
Fetch	no fetch			R0	R1	R2	R3		
Decode	`NOP`				R0	R1	R2	R3	
Execute	`NOP`					R0	R1	R2	R3
PC	R0				R1	R2	R3	· · ·	

Figure 5.10: Reset

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

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

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

The following is a typical 16-bit interrupt routine:

```
_InterruptService:
STX     mr1,(i6); STY i7,(i6)+1
STX     lr1,(i6); STY lr0,(i6)+1

STX     i0,(i6); STY i1,(i6)
...
(actual interrupt functionality)
...
LDX     (i6),i0; LDY (i6)-1,i1

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 chip-specific memory address 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 interrupt routine. Both of the interrupt routines presented here may change the values of X and Y flags, which therefore cannot be used in the actual program in sections where interrupts are enabled.

```
STX     i7,(i6)+1; STY lr0,(i6)
MVX     ipr1,i7
STX     lr1,(i6)+1; STY i7,(i6)
STX     mr1,(i6)+1

STX     i0,(i6); STY i1,(i6)
        ...
(actual interrupt functionality)
        ...
LDX     (i6),i0; LDY (i6)-1,i1

LDX     (i6)-1,mr0
LDC     INT_GLOB_EN,i7
STY     i7,(i7)
LDX     (i6),lr0; LDY (i6)-1,i7  // lr1 ipr1
JR      (i7)
LDX     (i6),i7; LDY (i6)-1,lr0  // i7  lr0
```

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

5.4 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. Interrupt Controller is forced to interrupt cycle #3.

Fig. 5.10 shows reset behavior. R0–R3 denote addresses reset vector – reset vector+3.

Cycle	1	2	3	4	5	6	7	8	9
Int Cycle	idle				1	2	3	4	5
Fetch	#1	#2			#3	I0	I1	I2	I3
Decode	halt	#1			#2	#3	I0	I1	I2
Execute		halt			#1	#2	#3	I0	I1
PC	#1	#2			#3	#4	I1	I2	I3

Figure 5.11: HALT execution

5.5 Halt

In HALT, the processor input clock is held low until an interrupt occurs. The execution pipeline is stopped.

When an interrupt occurs, the processor will execute 3 instructions after HALT instruction before executing the first interrupt instruction. See figure 5.11. In the figure, the execution of the `HALT` instruction takes 3 cycles (cycles #2. . .#4). The interrupt request is received during cycle #4.

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.

Chapter 6

Instruction Set Reference

6.1 Programming Model

The processor programming model is shown in Fig. 6.1. The processor contains arithmetic, address and control registers.

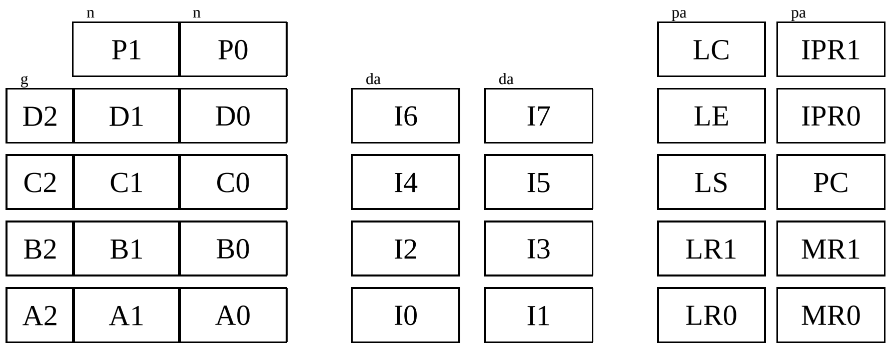


Figure 6.1: Processor programming model

Arithmetic registers are the n-bit registers `A0, A1, B1, ..., D1` and the g-bit guard bit registers `A2, ..., 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 $2n$-bit register. The arithmetic registers can be used either as n-bit registers mentioned above or as $2n + g$-bit registers (`A, B, C, D, P`).

Address registers are the da-bit index registers `I0, I1, ..., I7`. Optionally there may also be index registers `I8, I9, ..., I15`.

Control registers are the program counter `PC`, link registers `LR0, LR1` and mode registers `MR0, MR1`. Optional loop hardware registers are `LS, LE, LC` and page registers `IPR0, IPR1`.

6.2 Flags and Mode Bits

The mode/status register `MR0` bits are (here in the case *pa* = 16):

15							8	7							0
d	d	d	d	d	S	I	R	L	X	Y	Z	N	V	E	C
mode bits								flags							

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

The normal definition of the flags and mode bits is as follows. Exceptions to the flag behavior are listed in the particular instruction description.

6.2.1 Loop (L)

When set, the loop flag disables loop end detection, i.e. loop hardware. The flag is automatically set by the interrupt mechanism to prevent false loop end detections when the 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.

The detailed operation of the loop flag is as follows:

- Interrupt sets the loop flag. The value in MR1 corresponds to L=1.
- 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.

6.2.2 Index X (X)

The flag is set or cleared based on the address ALU output (updated address). If address ALU X uses modulo addressing, the flag is set if the modulo restoring to buffer is done (the calculated linear address is outside the buffer). If restoring is not done (the linear address is inside the buffer), the flag is cleared.

If address ALU X uses other addressing modes (linear or bit reversal), the flag is set if the uppermost bit is set. If the bit is clear, the flag is cleared.

6.2.3 Index Y (Y)

Same as Index X above, but uses address ALU Y instead.

6.2.4 Zero (Z)

If the ALU is operating in the $2n + g$-bit mode and bits $2n + g - 1 \ldots 0$ of the ALU result are all clear, the flag is set.

If the ALU is operating in the n-bit mode and bits $n - 1 \ldots 0$ of the ALU result are all clear, the flag is set.

Otherwise, the flag is cleared.

6.2.5 Negative (N)

If the ALU is operating in the $2n + g$-bit mode and bit $2n + g - 1$ of the ALU result is set, the flag is set.

If the ALU is operating in the n-bit mode and bit $n - 1$ of the ALU result is set, the flag is set.

Otherwise, the flag is cleared.

6.2.6 Overflow (V)

Set if an arithmetic overflow occurs in the ALU result.

6.2.7 Extension (E)

If the ALU is operating in the $2n + g$-bit mode and bits $2n + g - 1 \ldots 2n - 1$ are all the same (either all ones or all zeros), the flag is cleared.

If the ALU is operating in the n-bit mode, the flag is cleared.

Otherwise, the flag is set.

6.2.8 Carry (C)

If a carry is generated in an addition or a borrow is generated in a subtraction, the flag is set. The flag is set also in `LSR` and `LSRC`, if the LSB bit of the operand is logical '1'.

Otherwise, the flag is cleared.

6.2.9 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. Rounding precedes saturation, if both are enabled simultaneously.

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

6.2.10 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.

6.2.11 Rounding (R)

If the rounding mode bit is set, the *2n+g*-bit ALU operations will round the result to *n* bits according to the selected rounding mode. The rounding mode is (by default) chosen by the parameter *roundmode*, or by an extension specific mode bit (to be defined). By default, there is just one rounding mode available in the particular implementation of the core. Rounding precedes saturation, if both are enabled.

The possible rounding modes are truncate, normal round, convergent round to 0 and convergent round to 1. Rounding will clear the *n* bits of the lower half of the result, and adjust the upper half according to the hard-wired rounding mode. In truncation, the upper half is left intact. In normal rounding, the uppermost bit of the lower half is added to the upper part (thus rounding up if the lower part was at least half of the LSB value of the upper part). In convergent rounding the LSB of the upper half also has effect on the rounding operation in case the lower half is exactly half of the LSB (0b1000 $\cdots$). In convergent 0, the upper half LSB is to be added to the upper half in the special case (thus adding 1 if the LSB is one). In convergent to 1, the complement of the upper half LSB is to be added to the upper half in the special case (thus inserting 1 if the LSB is zero). The rounded result will be written in one of the accumulators ($2n + g$ registers).

If the rounding mode bit is clear, the ALU operates normally in the $2n + g$ base.

6.3 List of Instructions

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

Mnemonic	meaning	type	operands	result	S	I	R	L	X	Y	Z	N	V	E	C	db	da
ABS	absolute value	A	Areg	Areg	u	–	u	–	–	–	x	x	x	x	x	–	–
ADD	add	A	2 × Areg	Areg	u	–	u	–	–	–	x	x	x	x	x	–	–
ADDC	add with carry	A	2 × Areg, c	Areg	u	–	u	–	–	–	x	x	x	x	x,u	–	–
AND	logical AND	A	2 × Areg	Areg	–	–	–	–	–	–	x	x	0	x	0	–	–
ASR	1-b arithm. right shift	A	Areg	Areg	–	–	–	–	–	–	x	0	0	x	x	–	–
CALLcc	conditional call	C	addr,cc	LR0	–	–	–	0	u	u	u	u	u	u	u	yes	yes
HALT	wait for an interrupt	C	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
J	jump	C	addr	–	–	–	–	0	–	–	–	–	–	–	–	–	yes
Jcc	conditional jump	C	addr,cc	–	–	–	–	0	u	u	u	u	u	u	u	yes	yes
JMPI	jump, ignore delay slot	C	addr,Ireg	Ireg	–	–	–	–	–	–	–	–	–	–	–	–	yes
JR	jump register	C	0	–	–	–	–	0	–	–	–	–	–	–	–	–	yes
JRcc	conditional JR	C	cc, Ireg	–	–	–	–	0	u	u	u	u	u	u	u	yes	yes
LDC	load constant	M	imm	reg	–	–	–	–	–	–	–	–	–	–	–	–	–
LDX	load on X bus	M	Ireg, mod	reg	–	–	–	–	x	–	–	–	–	–	–	–	–
LDY	load on Y bus	M	Ireg, mod	reg	–	–	–	–	–	x	–	–	–	–	–	–	–
LOOP	start loop	C(O)	reg, addr	Lregs	–	–	–	0	–	–	–	–	–	–	–	–	yes
LSL	1-b log. left shift	A	Areg	Areg	–	–	–	–	–	–	x	x	x	x	x	–	–
LSLC	LSL with carry	A	Areg,c	Areg	–	–	–	–	–	–	x	x	x	x	x	–	–
LSR	1-b log. right shift	A	Areg	Areg	–	–	–	–	–	–	x	0	0	x	x	–	–
LSRC	LSR with carry	A	Areg,c	Areg	–	–	–	–	–	–	x	x	0	x	x	–	–
MAC	multiply-accumulate	A	2 × Areg	Areg,P	u	u	u	–	–	–	x	x	x	x	x	–	–
MSU	multiply-subtract	A	2 × Areg	Areg,P	u	u	u	–	–	–	x	x	x	x	x	–	–
MUL	multiply	A	2 × Areg	P	u	u	–	–	–	–	–	–	–	–	–	–	–
MVX	register move	M	reg	reg	–	–	–	–	–	–	–	–	–	–	–	–	–
MVY	register move	M	reg	reg	–	–	–	–	–	–	–	–	–	–	–	–	–
NOP	no operation	M	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
NOT	logical NOT	A	Areg	Areg	–	–	–	–	–	–	x	x	0	x	0	–	–
OR	logical OR	A	2 × Areg	Areg	–	–	–	–	–	–	x	x	0	x	0	–	–
RESP	restore P	C	2 × Areg	P	–	–	–	–	–	–	–	–	–	–	–	–	–
RETI	return from interr.	C	Ireg	PC	–	–	–	0	–	–	–	–	–	–	–	–	yes
STX	store on X bus	M	Ireg, mod, reg	mem	–	–	–	–	x	–	–	–	–	–	–	–	–
STY	store on Y bus	M	Ireg, mod, reg	mem	–	–	–	–	–	x	–	–	–	–	–	–	–
SUB	subtract	A	2 × Areg	Areg	u	–	u	–	–	–	x	x	x	x	x	–	–
SUBC	SUB with carry	A	2 × Areg, c	Areg	u	–	u	–	–	–	x	x	x	x	x,u	–	–
XOR	logical XOR	A	2 × Areg	Areg	–	–	–	–	–	–	x	x	0	x	0	–	–

Types: A = arithmetic (logic), C = control, M = data move, (O) = optional
Operands and result: reg = register, I = index, mod = modifier, addr = address,
Mode bits and flags: x = sets flag, u = uses bit, 0 = sets flag to 0,
Delay slots: db = delay slot before, da = delay slot after

6.4 Instruction Descriptions

The instruction description includes the `mnemonic` and a `one line description` (name) of the command, the `syntax` and `mathematical expression` of the instruction, 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`. The number of registers is dependent on the core parameters.

Several instructions 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.

ABS Absolute value

$\texttt{ABS}\ Op2, \texttt{A}_n;\quad |Op2| \rightarrow \texttt{A}_n$
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 7.4 (ALU operand), and the result coding in Table 7.3. 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
1111	0000	rrrr	AAA	parallel move

`rrrr` = Op2, AAA = target register.

ADD Addition of two operands

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

The operand coding is shown in Table 7.4 (ALU operand), and the result coding in Table 7.3. `LSL` is constructed with `ADD` Op1, Op1, $\texttt{A}_n$.

Coding:

31 28	27 24	23 20	19 17	16 0
0100	RRRR	rrrr	AAA	parallel move

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

ADDC Addition of two operands with carry

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

The operand coding is shown in Table 7.4 (ALU operand), and the result coding in Table 7.3.

Coding:

31 28	27 24	23 20	19 17	16 0
1000	RRRR	rrrr	AAA	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 7.4 (ALU operand), and the result coding in Table 7.3.

Coding:

31 28	27 24	23 20	19 17	16 0
1011	RRRR	rrrr	AAA	parallel move

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

ASR Arithmetic shift right

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

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

The instruction shifts right by one position. The LSB bit is discarded, and 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
1111	0001	rrrr	AAA	parallel move

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

CALLcc Conditional delayed jump and save return address

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

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 20	19 16	15 12	11 8	7 4	3 0
0010	1001	nnnn	nnnn	nnnn	nnnn	nncc	cccc

`nn...nn` = absolute address, `cccccc` = 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 20	19 16	15 12	11 8	7 4	3 0
0010	1101	dddd	dddd	dddd	dddd	dddd	dddd

`dd` = don't care.

J[1] Delayed jump to absolute address

$$\texttt{J}\ addr;\quad addr \rightarrow \texttt{PC}$$

Flags: `L=0`.

Note the one delay slot associated with this instruction.

Coding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
0010	1000	nnnn	nnnn	nnnn	nnnn	nn00	0000

`nn...nn` = absolute address.

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 6.1 (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 20	19 16	15 12	11 8	7 4	3 0
0010	1000	nnnn	nnnn	nnnn	nnnn	nncc	cccc

`nn...nn` = absolute address, `cccccc` = condition.

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

Table 6.1: 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
`000110`	XS	index X set	X = 1
`000111`	YS	index Y set	Y = 1
`001000`	LT	less than zero	$N \oplus (V \cdot \overline{S})$ = 1
`001001`	LE	less than or equal to zero	$N \oplus (V \cdot \overline{S})$ + 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	X = 0
`010110`	XC	index X clear	X = 0
`010111`	YC	index Y clear	Y = 0
`011000`	GE	greater than or equal to zero	$N \oplus (V \cdot \overline{S})$ = 0
`011001`	GT	greater than zero	$N \oplus (V \cdot \overline{S})$ + Z = 0

JMPI Jump, ignore delay slot, increment index register

`JMPI` $addr, (Op1) + n;\quad addr \rightarrow$ `PC`, $Op1 + n \rightarrow Op1,\ 0 \rightarrow$ `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). The X flag is not updated regardless of the LDX result.

This instruction is used in interrupt vector jump table.

Coding:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
`0010`	`1010`	`nnnn`	`nnnn`	`nnnn`	`nnnn`	`nndm`	`mrrr`

`nn...nn` = absolute address, `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

$$\texttt{JRcc};\;\; \mathit{if\ cond}:\ \texttt{LR0} \rightarrow \texttt{PC}$$

Flags: `L=0`.

JRcc Conditional delayed jump to the address in link register 0

$$\texttt{JRcc}\ (Op1);\;\; \mathit{if\ cond}:\ \texttt{LR0} \rightarrow \texttt{PC},\ Op1 \rightarrow \texttt{IPR0}$$

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 6.1 (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 20	19 16	15 12	11 8	7 4	3 0
0 0 1 0	0 0 0 0	0 d d d	d d d d	d d d d	d d d d	d d c c	c c c c

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

`ddd` = don't care bits, `cccccc` = condition, `rrr` = Op1 (I0. . .I7)

LDC Load constant to a register

$$\texttt{LDC}\ constant, Op1;\;\; constant \rightarrow Op1$$

Flags: `no change`.

The register (Op1) coding is shown in Table 7.9 (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 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
0 0 0 i	i i i i	i i i i	i i i i	i i i i	i i i i	i i R R	R R R R

`RRRRRR` = Op1, `ii...ii` = constant immediate.

LDX Load register from X-memory

`LDX` $(Op1), Op2;\;\; X[Op1] \rightarrow Op2,\; update\; Op1$
Flags: X.

LDY Load register from Y-memory

`LDY` $(Op1), Op2;\;\; Y[Op1] \rightarrow Op2,\; update\; Op1$
Flags: Y.

Coding (double full moves):

31 28	27 14	13 0
0011	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
oooo	dddd	dddd	ddd	0bdFF	FFFF	FFFF	FFFF

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
oooo	dddd	dddd	ddd	1xxxx	xxxx	yyyy	yyyy

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

LDX Load register from X memory with long address

$LDX\;(Op2 : Op3), Op1;\;\; X[Op2 : Op3] \rightarrow Op1$
Flags: X.

STX Store register in X memory with long address

$STX\; Op1, (Op2 : Op3);\;\; Op1 \rightarrow X[Op2 : Op3]$
Flags: X.

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

Coding (parallel move):

31 17	16 10	9 6	5 0
arithmetic opcode	0010100	srrr	RRRRRR

RRRRRR = Op1, rrr = Op2, s = store/load

Table 6.2: Loop count register coding.

Binary code	Register
0000a	A0 . . . A1
0001a	B0 . . . B1
0010a	C0 . . . C1
0011a	D0 . . . D1
01000	LR0
01001	LR1
01010	MR0
01011	MR1
01100	(reserved)
01101	LC (optional)
01110	LS (optional)
01111	LE (optional)
10rrr	I0 ... I7
11rrr	I8 ... I15 (optional)

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 **optional** instruction starts a hardware loop. The instruction carries a register number, as encoded in Table 6.2 (Loop count), 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:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
0010	01nn	nnnn	nnnn	nnnn	nnnn	nndr	rrrr

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

LSL[2] 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
`0100`	`rrrr`	`rrrr`	`AAA`	`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
`1000`	`rrrr`	`rrrr`	`AAA`	`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 7.4 (ALU operand), and the result coding in Table 7.3.

Coding:

31 28	27 24	23 20	19 17	16 0
`1111`	`0010`	`rrrr`	`AAA`	`parallel move`

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

[2]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 7.4 (ALU operand), and the result coding in Table 7.3.

Coding:

31 28	27 24	23 20	19 17	16 0
1111	0011	rrrr	AAA	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
0101	rrrm	mRRR	AAA	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
0111	rrrm	mRRR	AAA	parallel move

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

MUL Multiply

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

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:

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

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

MVX/MVY Register-to-register move

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

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):

31 17	16 12	11 6	5 0
arithmetic opcode	0 0 1 0 0	s s s s s s	d d d d d d

Coding (double move):

31 28	27 24	23 18	17 12	11 6	5 0
0 0 1 0	1 0 1 1	S S S S S S	D D D D D D	s s s s s s	d d d d d d

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

NOP No operation

$$\mathtt{NOP};\quad no\,effect$$
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
1111	0100	dddd	ddd	parallel move

`ddd` = don't care.

NOT[3] Bitwise logic NOT operation

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

The operand (Op2) coding is shown in Table 7.4 (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
1101	1001	rrrr	AAA	parallel move

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

OR Bitwise logic OR operation

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

The operands are encoded as described in Table 7.4 (ALU operand), and the result coding in Table 7.3. The target is one of the registers.

Coding:

31 28	27 24	23 20	19 17	16 0
1100	rrrr	RRRR	AAA	parallel move

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

RESP Restore P register

$$\texttt{RESP}\ Op1, Op2;\ \ Op1 \rightarrow \texttt{P0}\ \ Op2 \rightarrow \texttt{P1}$$

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 2.5. 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

$$\texttt{RETI};\ \ \texttt{LR1} \rightarrow \texttt{PC}$$

Flags: `L=0`.

RETI Delayed return from interrupt

$$\texttt{RETI}\ (Op1);\ \ \texttt{LR1} \rightarrow \texttt{PC},\ Op1 \rightarrow \texttt{IPR0}$$

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 20	19 16	15 12	11 8	7 4	3 0
0 0 1 0	0 0 0 1	0 d d d	d d d d	d d d d	d d d d	d d d d	d d d d

31 28	27 24	23 20	19 16	15 12	11 9	8 6	5 0
0 0 1 0	0 0 0 1	1 d d d	d d d d	d d d d	d d d	r r r	d d d d d d

`ddd` = don't care bits, `nnn` = absolute address, `rrr` = Op1 (I0. . .I7)

STX Store a register in X memory

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

Flags: X.

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

STY Store a register in Y memory

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

Flags: Y.

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 7.4 (ALU operand), and the result coding in Table 7.3.

Coding:

31 28	27 24	23 20	19 17	16 0
0110	RRRR	rrrr	AAA	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 7.4 (ALU operand), and the result coding in Table 7.3.

Coding:

31 28	27 24	23 20	19 17	16 0
1001	RRRR	rrrr	AAA	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 7.4 (ALU operand), and the result coding in Table 7.3. `XOR` has also been used to implement `NOT`.

Coding:

31 28	27 24	23 20	19 17	16 0
1101	RRRR	rrrr	AAA	parallel move

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

6.5 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.

6.5.1 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
```

6.5.2 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 flags can be changed since they do not affect the "carry clear" condition.

```
example1:
  ldx (i0)+1, NULL              /* must not change C flag */
  jcc jump_target
  nop                           /* jump delay slot */

example2:
  not a0,a1                     /* must not change X flag */
  jxs jump_target
  nop                           /* jump delay slot */
```

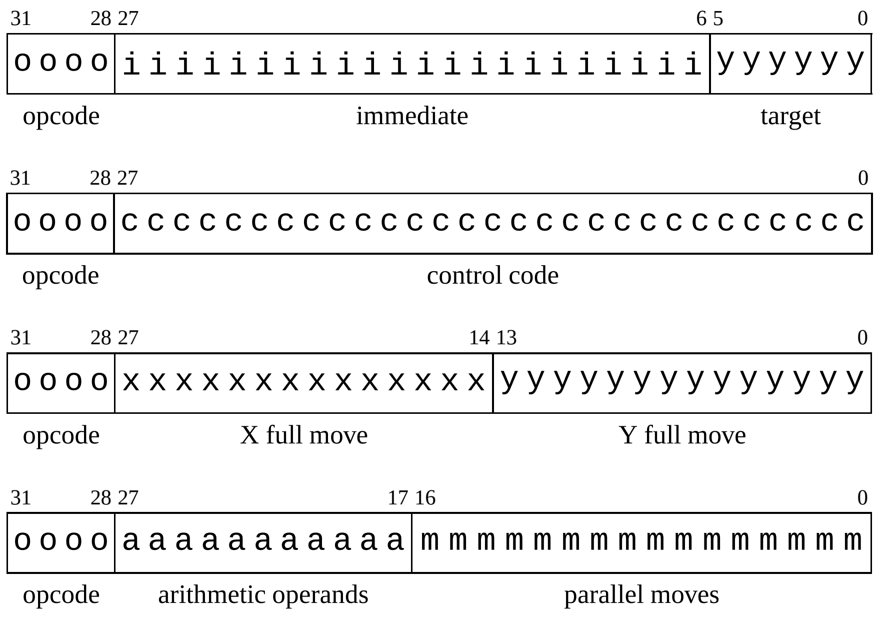
The reason for this restriction is the fact that the jump condition is determined during the decode phase. In a normal (linear) execution, the instruction immediately before the jump does not affect the jump. The situation is different if the jump instruction is canceled due to an interrupt. When execution 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.

Chapter 7

Instruction Coding

7.1 General Instruction Composition

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



7.2 Opcode Field

The encoding of operations is shown in Table 7.1. The control and double move extensions to the opcode are described in the following section.

Table 7.1: 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`	(reserved)	
`1011`	`AND`	yes
`1100`	`OR`	yes
`1101`	`XOR`	yes
`1110`	(reserved)	
`1111`	Single op instructions	yes

7.3 Control Code

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 6.1 (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 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 (optional) 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 register field encoding is given in Table 6.2 (Loop count). 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

Table 7.2: Control Codes.

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)		

for move encoding.

`RESP` is a special instruction to restore the `P` register.

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

Table 7.3: ALU result coding

Binary code	n-bit register	2n+g-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`

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:

27 24	23 20	19 17
`alu op1`	`alu op2`	`alu result`

In `MAC`:

27 25	24 23	22 20	19 17
`mul op1`	`mode`	`mul op2`	`alu result`

Table 7.4 (ALU operand) gives the encoding of Op1 and Op2 of the ALU (fields alu op1 & alu op2). S denotes sign extension.

Table 7.5 (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:

27 24	23 20	19 17
`single opcode`	`alu op2`	`alu result`

In `MUL`:

27 25	24 23	22 20	19 17
`MUL opcode`	`mode`	`mul op2`	`mul op1`

Table 7.4: 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

Table 7.5: Mul operand.

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

Table 7.6: Mul mode.

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

Table 7.7: Single operand ALU instructions.

Binary code	Operation
`0000`	`ABS`
`0001`	`ASR`
`0010`	`LSR`
`0011`	`LSRC`
`0100`	`NOP`
`0101` ... `1101`	(reserved)
`111X`	`MUL`

Table 7.6 (Mul mode) gives the encoding of the mode field.

The result sub-field encoding is shown in Table 7.3.

Table 7.4 (ALU operand) gives the encoding of Op2 of the ALU (field alu op2).

The single-operand opcode encoding is given in Table 7.7.

7.5 Move Encoding

The move instructions are `LDX`, `LDY`, `STX` and `STY`, the X and Y 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. 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:

13 10	9 6	5 0
s r r r	p p p p	R R R R R R

In short moves the move field is as follows:

13 10	9 6	5 0
s r r r	p 0 0 0	0 0 0 R R R

`s` = store/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.

27 14	13 0
s r r r p p p p R R R R R R	s r r r p p p p R R R R R R
X full move	Y full move

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

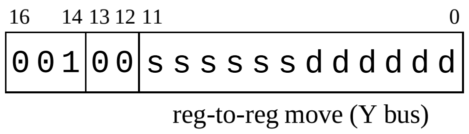
16 14	13 0
0 b 0	s r r r p p p p R R R R R R
	full move

`b` = bus (0=X,1=Y)

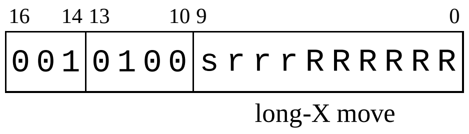
16	15 8	7 0
1	s r r r p R R R	s r r r p R R R
	X short move	Y short move

Table 7.8: Registers in short move.

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



reg-to-reg move (Y bus)



long-X move

The coding of the store/load bit is given in Table 7.10. The `rrr` register is the number of the desired address register. The src/dest register number (`(RRR)RRR`) is given in Table 7.9 (Source and target), and the addressing mode in Table 7.11. 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`. It is recommended to use the odd-numbered registers as $\overline{\texttt{In}}$ and even as `In`. The modifier register set `Mn` (in the basic version aliased to the odd `In`) can be used instead of the $\overline{\texttt{In}}$. If even `In` and even `Mn` are used in the basic version, the code will be transferable to versions with an additional dedicated modifier register set. The post modifications by the $\overline{\texttt{In}}$ (`Mn`) are defined in Table 7.12.

7.6 Addressing Modes

The addressing modes and their availability in short and full formats are summarized in Table 7.13. 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

The modulus `m` is given by the lower end of $\overline{\texttt{In}}$ (word length – 3 bits) in unsigned format such that the third bit from the MSB end of $\overline{\texttt{In}}$ defines whether to add or subtract. In the $\pm$`m` case, the `m` is a (word length – 2 bit) two's complement number, where the

Table 7.9: 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`	`MR1`
`001100`	`NULL` (update index reg & flags)
`001101`	`LC` (optional)
`001110`	`LS` (optional)
`001111`	`LE` (optional)
`010rrr`	`I0` ... `I7`
`011rrr`	`I8` ... `I15` (optional)
`100000`	`A2` (optional)
`100001`	`B2` (optional)
`100010`	`C2` (optional)
`100011`	`D2` (optional)
`100100`	Move NOP (no updates)
`100101` · · · `111101`	reserved
`111110`	`IPR0`
`111111`	`IPR1`

Table 7.10: Load/Store coding.

Binary code	Mode
`0`	load
`1`	store

Table 7.11: Addressing Modes.

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

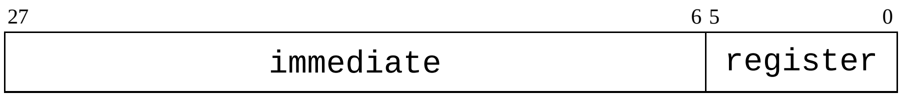
Table 7.12: Modifications by the $\overline{\texttt{In}}$ register.

Binary code	Modification
000	In = (In+m) (m positive)
001	In = (reserved)
010	In = (In+2%m) (optional)
011	In = (In−2%m) (optional)
100	In = (In++%m) (optional)
101	In = (In−−%m) (optional)
110	In = (In+m) bit reverse (optional)
111	In = (In+m) (m negative)

sign is automatically in the three MSB bits of $\overline{\texttt{In}}$. In the basic version only the ±m modifications are implemented.

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 7.9.

Table 7.13: Addressing mode summary.

Mode	full move code	short move code	$\overline{I_n}$	parameter
Linear post-inc/dec				
(I_n)	srrr0000RRRRRR	srrr0RRR	—	—
(I_n)++	srrr0001RRRRRR	N/A	—	—
(I_n)+2	srrr0010RRRRRR	N/A	—	—
(I_n)+3	srrr0011RRRRRR	N/A	—	—
(I_n)+4	srrr0100RRRRRR	N/A	—	—
(I_n)+5	srrr0101RRRRRR	N/A	—	—
(I_n)+6	srrr0110RRRRRR	N/A	—	—
(I_n)+7	srrr0111RRRRRR	N/A	—	—
(I_n)--	srrr1111RRRRRR	N/A	—	—
(I_n)–2	srrr1110RRRRRR	N/A	—	—
(I_n)–3	srrr1101RRRRRR	N/A	—	—
(I_n)–4	srrr1100RRRRRR	N/A	—	—
(I_n)–5	srrr1011RRRRRR	N/A	—	—
(I_n)–6	srrr1010RRRRRR	N/A	—	—
(I_n)–7	srrr1001RRRRRR	N/A	—	—
(I_n)*	Linear post-inc/dec			
(I_n)+m, $m \geq 0$	srrr1000RRRRRR	srrr1RRR	000 mmmm...mmmm	—
(I_n)+m, $m < 0$	srrr1000RRRRRR	srrr1RRR	111 mmmm...mmmm	—
(I_n)*	Modulo post-inc/dec			
(I_n)++%m	srrr1000RRRRRR	srrr1RRR	100 mmmm...mmmm	amm[0]
(I_n)--%m	srrr1000RRRRRR	srrr1RRR	101 mmmm...mmmm	amm[0]
(I_n)+2%m	srrr1000RRRRRR	srrr1RRR	010 mmmm...mmmm	amm[0]
(I_n)–2%m	srrr1000RRRRRR	srrr1RRR	011 mmmm...mmmm	amm[0]
(I_n)*	Bit reversal			
(I_n)+m bit-rev	srrr1000RRRRRR	srrr1RRR	110 mmmm...mmmm	amm[1]
Register as source/destination				
A_n	srrrpppp000RRR	srrrpRRR	—	—
A_n 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 ≥ 1
LS	srrrpppp001110	N/A	—	lc ≥ 1
LE	srrrpppp001111	N/A	—	lc ≥ 1
I_n, n=0· · ·7	srrrpppp010RRR	N/A	—	—
I_n, n=8· · ·15 M_n, n=0· · ·7	srrrpppp011RRR	N/A	—	iregs = 16

Chapter 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.

```
.fract 15

.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     0xa00f,i1    // modulo -1 addressing
                             // (could be 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,firloop   // 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
        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

```
.fract 15

.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     0xa00f,i1     // modulo -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,firloop
        LDC     output,i4
/* 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)
firloop:
        ADDC    a,p,a         // result after this stage in a:b

     /* 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.

```
.fract 15

#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,biquadloop
        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
biquadloop:
        ADD     a,p,a   // result after this biquad to a-reg.

        STX     a1,(i4) // store output
iirend:
.end
```

8.4 Single-Precision Matrix Multiply

C = A $\times$ 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,inloop
        MUL     b0,b1;    LDX (i0)*,b1; LDY (i2)*,b0
inloop:
        MAC     b0,b1,a;  LDX (i0)*,b1; LDY (i2)*,b0
                                  // out+=A[i][k]*B[k][j]

        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
        SUB      a0,ONES,d0     // make result have exp(a)+1
$3:
        ASR      b,b

/* 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
```