

ST20-C1 Core Instruction Set Reference Manual



72-TRN-274-01

July 1997

Contents

1	Introduction	4
1.1	ST20-C1 features	4
1.2	Manual structure	5
2	Notation	6
2.1	Instruction listings	6
2.2	Instruction definitions	8
2.3	Operators used in the definitions	11
2.4	Data structures and constants	13
3	Architecture	16
3.1	Values	16
3.2	Memory	18
3.3	Registers	20
3.4	Instruction encoding	24
4	Using ST20-C1 instructions	30
4.1	Manipulating the evaluation stack	30
4.2	Loading and storing	31
4.3	Expression evaluation	33
4.4	Arithmetic	35
4.5	Forming addresses	41
4.6	Comparisons and jumps	43
4.7	Evaluation of boolean expressions	45
4.8	Bitwise logic and bit operations	47
4.9	Shifting and byte swapping	48
4.10	Function and procedure calls	48
4.11	Peripherals and I/O	52
4.12	Status register	55
5	Multiply accumulate	56
5.1	Data formats	56
5.2	mac and umac	56
5.3	Short multiply accumulate loop	56
5.4	Biquad IIR filter	58
5.5	Data vectors	59
5.6	Scaling	59
5.7	Data formats	65

6	Exceptions	70
6.1	Exception levels	71
6.2	Exception vector table.	72
6.3	Exception control block and the saved state.	73
6.4	Initial exception handler state	74
6.5	Restrictions on exception handlers	75
6.6	Interrupts.	75
6.7	Traps.	76
6.8	Setting up the exception handler	76
7	Multi-tasking	78
7.1	Processes	78
7.2	Descheduled processes	79
7.3	Queues	80
7.4	Timeslicing	81
7.5	Inactive processes	82
7.6	Descheduled process state.	82
7.7	Initializing multi-tasking	83
7.8	Scheduling kernels	84
7.9	Semaphores	84
7.10	Sleep.	85
8	Instruction Set Reference	87
	Appendices	168
A	Constants and data structures.	169
B	Instruction set summary.	174
C	Compiling for the ST20-C1	178
D	Glossary	187

1 Introduction

This manual provides a summary and reference to the ST20 architecture and instruction set for the ST20-C1 core.

ST20 is a technology for building successful embedded VLSI designs. ST20 devices comprise a collection of VLSI macro-cells connected through a high-performance on-chip bus. This architecture allows the easy construction of both general purpose (e.g. ST20-MC1 micro-controller) and application specific devices (e.g. ST20-TPx digital set top box family).

The ST20 macro-cell library includes CPU micro-cores, on-chip memories and a wide range of digital and analogue I/O devices. SGS-THOMSON offers a range of ST20 CPU micro-cores, allowing the best cost vs. performance trade-off to be achieved in each application area. This manual describes the ST20-C1 CPU micro-core.

ST20 devices are available from SGS-THOMSON and licensed second source vendors.

1.1 ST20-C1 features

The ST20-C1 has the following features:

- It is implemented as a 2-way superscalar, 3-stage pipeline, with an internal 16-word register cache. This architecture can sustain 4 instructions in progress, with a maximum of 2 instructions completing per cycle.
- It uses a variable length instruction coding scheme based on 8-bit units which gives excellent static and dynamic code size. Instructions take between 1 and 8 units to code, with an average of 1.25 units (10 bits) per instruction.
- It provides flexible prioritized vectored interrupt capabilities. The worst case interrupt latency is 0.5 microseconds (at 33 Mhz operating frequency).
- It provides extensive instruction level support for 16-bit digital signal processing (DSP) algorithms.
- It is particularly suitable for low power and battery-powered applications, with low core operating power, and sophisticated power management facilities.
- It provides extensive real-time debugging capability through the optional ST20 diagnostic controller unit (DCU) macro-cell, which supports fully non-intrusive breakpoints, watchpoints and code tracing.
- It has a flexible and powerful built-in hardware scheduler. This is a light-weight *real-time operating system* (RTOS) directly implemented in the microcode of the ST20-C1 processor. The hardware scheduler can be customized and provides support for software schedulers.
- It provides a built-in user-programmable 32-bit input/output register providing system control and communication capability directly from the CPU.

1.2 Manual structure

The manual is divided into the following chapters:

- 1 This introduction chapter, which explains the structure of the book;
- 2 A notation chapter (Chapter 2) which explains the layout and notation conventions used in the instruction definitions and elsewhere;
- 3 An architecture chapter (Chapter 3), which explains the structure of the ST20-C1 core, the registers, memory addressing, the format of the instructions and the exception handling and process models;
- 4 Four chapters on using the instructions and how the instructions can be used to achieve certain useful outcomes: Chapter 4 on the general instructions; Chapter 5 on multiply-accumulate; Chapter 6 on interrupts and traps; and Chapter 7 on processes and support for multi-tasking.
- 5 An alphabetical listing of the instructions, one to a page (Chapter 8). Descriptions and formal definitions are presented in a standard format with the instruction mnemonic and full name of the instruction at the top of the page. The notation used is explained in detail in Chapter 2.

In addition there are appendices listing constants and structures, covering issues related to compiling for a ST20-C1 core and listing the instruction set plus a glossary for ST20-C1 terminology.

2 Notation

This chapter describes the notation used throughout this manual, including the meaning of the instruction listings and the meanings and values of constants.

2.1 Instruction listings

The instructions are listed in alphabetical order, one to a page. Descriptions are presented in a standard format with the instruction mnemonic and full name of the instruction at the top of the page, followed by these categories of information:

- **Code:** the instruction code;
- **Description:** a brief summary of the purpose and behavior of the instruction;
- **Definition:** a more formal and complete description of the instruction, using the notation described below in section 2.2;
- **Status Register:** a list of errors and other changes to the Status Register which can occur;
- **Comments:** a list of other important features of the instruction;
- **See also:** cross references are provided to other instructions with related functions.

These categories are explained in more detail below, using the *and* instruction as an example.

2.1.1 Instruction name

The header at the top of each page shows the instruction mnemonic and, on the right, the full name of the instruction. For primary instructions the mnemonic is followed by 'n' to indicate the operand to the instruction; the same notation is used in the description to show how the operand is used. An explanation of the primary and secondary instruction formats is given in section 3.4.

2.1.2 Code

The code of the instruction is the value that would appear in memory to represent the instruction.

For secondary instructions the instruction 'operation code' is shown as the memory code — the actual bytes, including any prefixes, which are stored in memory. The value is given as a sequence of bytes in hexadecimal, decoded left to right. The codes are stored in memory in 'little-endian' format, with the first byte at the lowest address.

For example, the entry for the *and* instruction is:

Code: F9

This means that the hexadecimal byte value F9 would appear in memory for an *and*.

For primary instructions the code stored in memory is determined partly by the value of the operand to the instruction. In this case the op-code is shown as 'Function x ' where x is the function code in the last byte of the instruction. For example, *adc* (*add constant*) is shown as

Code: Function 8

This means that *adc 1* would appear in memory as the hexadecimal byte value 81. For an operand n in the range 0 to 15, *adc n* would appear in memory as $8n$.

2.1.3 Description

The description section provides an indication of the purpose of the instruction as well as a summary of the behavior. This may include details of the use of registers, whose initial values may be used as parameters and into which results may be stored.

For example, the *and* instruction contains the following description:

Description: Bitwise AND of **Areg** and **Breg**.

2.1.4 Definition

The definition section provides a formal description of the behavior of the instruction. The behavior is defined in terms of its effect on the state of the processor, i.e. the changes in the values in registers and memory before and after the instruction has executed.

The effects of the instruction on registers, etc. are given as statements of the following form:

$\text{register}' \leftarrow \text{expression involving registers, etc.}$

$\text{memory_location}' \leftarrow \text{expression involving registers, etc.}$

Primed names (e.g. **Areg'**) represent values after instruction execution, while names without primes represent values when the instruction execution starts. For example, **Areg** represents the value in **Areg** before the execution of the instruction while **Areg'** represents the value in **Areg** afterwards. So the example above states that after the instruction has been executed the register or memory location on the left hand side holds the value of the expression on the right hand side.

Only the changed registers and memory locations are given on the left hand side of the statements. If the new value of a register or memory location is not given then the value is unchanged by the instruction.

The description is written with the main function of the instruction stated first. For example the main function of the *add* instruction is to put the sum of **Areg** and **Breg** into **Areg**). This is followed by the other effects of the instruction, such as rotating the stack. There is no temporal ordering implied by the order in which the statements are written.

2.2 Instruction definitions

For example, the *and* instruction contains the following description:

Definition:

$$\text{Areg}' \leftarrow \text{Breg} \wedge \text{Areg}$$
$$\text{Breg}' \leftarrow \text{Creg}$$
$$\text{Creg}' \leftarrow \text{Areg}$$

This says that the integer stack is rotated and **Areg** is assigned the bitwise AND of the values that were initially in **Breg** and **Areg**. After the instruction has executed **Breg** contains the value that was originally in **Creg**, and **Creg** has the value that was in **Areg**.

The notation is described more fully in section 2.2.

2.1.5 Status Register

This section of the instruction definitions lists any changes to bits of the **Status** register which can occur. The **Status** register is described in more detail in section 3.3.2.

2.1.6 Comments

This section is used for listing other information about the instructions that may be of interest. This includes an indication of the type of the instruction:

“Primary instruction” — indicates one of the 13 functions which may be directly encoded with an operand in a single byte instruction.

“Secondary instruction” — indicates an instruction which is encoded using *opr*.

An explanation of the primary and secondary instruction formats is given in section 3.4.

The **Comments** section also describes any situations where the operation of the instruction is undefined or invalid and any limits to the parameter values.

For example, the only comment listed for the *and* instruction is:

Comments:

Secondary instruction.

This says that *and* is a secondary instruction.

2.2 Instruction definitions

The following sections give a full description of the notation used in the formal definition section of the instruction descriptions.

2.2.1 The process state

The process state consists of the registers (**Areg**, **Breg**, **Creg**, **lptr**, **Tdesc**, **Wptr**, and **Status**), and the contents of memory. A description of the meanings and uses of the registers and special memory locations and data structures is given in section 3.3.

2.2.2 General

The instruction descriptions are not intended to describe the way the instructions are implemented, but only their effect on the state of the processor. So, for example, the result of *mul* is shown in terms of an intermediate result calculated to infinite precision, although no such intermediate result is used in the implementation.

Comments (in *italics*) are used to both clarify the description and to describe actions or values that cannot easily be represented by the notation used here; e.g. *take timeslice trap*. Some of these actions and values are described in more detail in other chapters.

An ellipsis is used to show a range of values; e.g. '*i* = 0..31' means that *i* has values from 0 to 31, inclusive.

Subscripts are used to indicate particular bits in a word; e.g. *Areg_i* for bit *i* of **Areg**, and *Areg_{0..7}* for the least significant byte of **Areg**. Note that bit 0 is the least significant bit in a word, and bit 31 is the most significant bit.

Except for **lptr**, certain reserved words of memory, and taking exceptions or switching processes, if the description does not mention the state of a register or memory location after the instruction, then the value will not be changed by the instruction.

lptr is assigned the address of the next instruction in the code *before* the instruction execution starts. The **lptr** is included in the description only when there are additional effects of the instruction (e.g. in the *jump* instruction). In these cases the address of the next instruction is indicated by the comment '*next instruction*'.

2.2.3 Undefined values

Some instructions in some circumstances leave the contents of a register or memory location in an undefined state. This means that the value of the location may be changed by the instruction, but the new value cannot be easily defined, or is not a meaningful result of the instruction. For example, when division by zero is attempted, **Breg** and **Creg** become undefined, i.e. they do not contain any meaningful data. An undefined value is represented by the name *undefined*.

The values of registers which become undefined as a result of executing an instruction are implementation dependent and are not guaranteed to be the same on different members or revisions of the ST20 family of processors.

2.2.4 Data types

The instruction set includes operations on three sizes of data: 8, 16 and 32-bit objects. 8-bit and 16-bit data can represent signed or unsigned integers and 32-bit data can

2.2 Instruction definitions

represent addresses, signed or unsigned integers. Generally the arithmetic is signed. In some cases it is clear from the context (e.g. from the operators used) whether a particular object represents a signed or unsigned number. A subscripted label is added (e.g. $Areg_{\text{unsigned}}$) to clarify where necessary.

2.2.5 Representing memory

The memory is represented by arrays of each data type. These are indexed by a value representing a byte address. Access to the three data types is represented in the instruction descriptions in the following way:

$\text{byte}[address]$ references a byte in memory at the given address

$\text{sixteen}[address]$ references a 16-bit half word in memory

$\text{word}[address]$ references a 32-bit word in memory

For all of these, the state of the machine referenced is that *before* the instruction if the function is used without a prime (e.g. $\text{word}[address]$), and that *after* the instruction if the function is used with a prime (e.g. $\text{word}'[address]$).

For example, writing a value given by an expression, $expr$, to the word in memory at address $addr$ is represented by:

$$\text{word}'[addr] \leftarrow expr$$

and reading a word from a memory location is achieved by:

$$\text{register}' \leftarrow \text{word}[addr]$$

Writing to memory in any of these ways will update the contents of memory, and these updates will be consistently visible to the other representations of the memory. For example, writing a byte at address 0 will modify the least significant byte of the word at address 0.

Data alignment

Generally, word and half word data items have restrictions on their alignment in memory. Byte values can be accessed at any byte address, i.e. they are byte aligned. 16-bit objects can only be accessed at even byte addresses, i.e. the least significant bit of the address must be 0. 32-bit objects must be word aligned, i.e. the 2 least significant bits of the address must be zero.

Address calculation

An address identifies a particular byte in memory. Addresses are frequently calculated from a base address and an offset. For different instructions the offset may be given in units of bytes or words depending on the data type being accessed. In order to calculate the address of the data, a word offset must be converted to a byte offset before being added to the base address. This is done by multiplying the offset by the number of bytes per word, i.e. 4.

As there are many accesses to memory at word offsets, a shorthand notation is used to represent the calculation of a word address. The notation $\text{register} @ x$ is used to

represent an address which is offset by x words ($4x$ bytes) from the address in *register*. For example, in the specification of *load non-local* there is:

$$\text{Areg}' \leftarrow \text{word}[\text{Areg} @ n]$$

Here, **Areg** is loaded with the contents of the word that is n words from the address pointed to by **Areg**, i.e. the word at address **Areg** + $4n$.

In all cases, if the given base address has the correct alignment then any offset used will also give a correctly aligned address.

2.3 Operators used in the definitions

A full list of the operators used in the instruction definitions is given in Table 2.1. Unless otherwise stated, all arithmetic is signed.

Symbol	Meaning
Unchecked (modulo) integer arithmetic	
$+$ $-$ \times $/$ rem	Signed integer add, subtract, multiply, divide and remainder. If the computation overflows the result of the operation is truncated to the word length. If a divide or remainder by zero occurs the result of the operation is undefined. No errors are signalled. The operator ' $-$ ' is also used as a monadic operator.
Signed comparison operators	
$<$ $>$ \leq \geq $=$ \neq	Comparisons of signed integer values: 'less than', 'greater than', 'less than or equal', 'greater than or equal', 'equal' and 'not equal'.
Bitwise operators	
\sim \wedge \vee \otimes $>>$ $<<$ $>>\text{arith}$	'Not', 'and', 'or', 'exclusive or', logical left and right shift and arithmetic right shift operations on bits in words.
Boolean operators	
not and or	Boolean combination in conditionals.

Table 2.1 Operators used in the instruction descriptions

Modulo operators

Arithmetic is done using *modulo* arithmetic — i.e. there is no checking for errors and, if the calculation overflows, the result 'wraps around' the range of values representable in the word length of the processor — e.g. adding 1 to the address at the top of the

2.3 Operators used in the definitions

address map produces the address of the byte at the bottom of the address map. These operators are represented by the symbols '+', '-', etc.

Error conditions

Any errors that can occur in instructions which are defined in terms of the modulo operators are indicated explicitly in the instruction description. For example the *add* instruction indicates the cases that can cause overflow or underflow, independently of the actual addition:

```
if (sum > MostPos)
{
    Areg'          ← sum - 2BitsPerWord
    Status'_underflow ← clear
    Status'_overflow  ← set
}
else if (sum < MostNeg)
{
    Areg'          ← sum + 2BitsPerWord
    Status'_underflow ← set
    Status'_overflow  ← clear
}
else
{
    Areg'          ← sum
    Status'_underflow ← clear
    Status'_overflow  ← clear
}
...
```

2.3.1 Functions

Type conversions

The following notation is used to indicate the type cast of x to a 16-bit integer:

int16 (x)

If x is too large or too small to fit into a 16-bit integer then the result of the instruction is undefined.

Double word splitting

Where a calculation is performed using a 48-bit or 64-bit value, the value may be split into two words. The function *low_word* returns the least significant word and the function *high_word* returns the most significant word.

2.3.2 Conditions to instructions

In many cases, the action of an instruction depends on the current state of the processor. In these cases the conditions are shown by an **if** clause; this can take one of the following forms:

- **if** *condition*
 statement
- **if** *condition*
 statement
 else
 statement
- **if** *condition*
 statement
 else if *condition*
 statement
 else
 statement

These conditions can be nested. Braces, {}, are used to group statements which are dependent on a condition. For example, the *cj* (*conditional jump*) instruction contains the following lines:

```

if (Areg = 0)
    lptr' ← next instruction + n
else
{
    lptr' ← next instruction

    Areg' ← Breg
    Breg' ← Creg
    Creg' ← Areg
}

```

This says that if the value in **Areg** is zero, then the jump is taken (the instruction operand, *n*, is added to the instruction pointer), otherwise the stack is popped and execution continues with the next instruction.

2.4 Data structures and constants

A number of data structures have been defined in this manual. Each comprises a number of data slots that are referenced by name in the text and the instruction descriptions.

These data structures are listed in the tables in Appendix A. Each table gives the name of each slot in the structure and the word offsets from the base address of the structure. A slot in a data structure is identified using the offset notation described in section 2.2.5:

word[*base_address* @ *word_offset*]

2.4 Data structures and constants

For example, the back pointer of a semaphore structure at address `sem` would be:

`word[sem @ s.Back]`

In addition, several constants are used to identify fixed values for the ST20-C1 processor. All the constants are listed in Appendix A.

Product identity value

This is the value returned by the *Idprodid* instruction. For specific product ids in the ST20 family refer to SGS-THOMSON.

3 Architecture

This chapter describes the general architectural features of the ST20-C1 core which are relevant to more than one instruction or group of instructions. Interrupts and traps are described in Chapter 6 and support for multi-tasking is described in Chapter 7. Other features which are related to specific tasks are described in Chapter 4. A full list of constants and data structures is given in Appendix A.

The ST20-C1 instruction set covers:

- control flow
- arithmetic and logical operations
- bit field manipulations
- shifting and byte-swapping
- register manipulations
- memory access with various addressing modes and data sizes
- task scheduling
- direct input/output

3.1 Values

The ST20-C1 core supports data objects of different sizes, either signed or unsigned. The sizes directly supported are bytes (8-bit), half words (16-bit), words (32-bit) and multiple words (64-bit, 96-bit etc.). Bytes, half-words and words may be loaded and stored. Arithmetic operations are provided for signed words and multiple words. A half word is called a *sixteen* in the instruction names.

The most negative integer (0x80000000) is known as *MostNeg* and the most positive (0x7FFFFFFF) as *MostPos*.

Boolean objects, taking one of the values *true* or *false*, are also used by some instructions. *False* is represented by the value 0 and *true* has the value 1. Section 4.7 describes how other values may be implemented for language compilation.

Several data structures are defined in this manual. Each comprises a number of data words (sometimes called *slots*) that are referenced by name in the text and the instruction descriptions and addressed as offsets from the base of the data structure. A full list of these data structures and other constants is given in Appendix A.

3.1.1 Ordering of information

The ST20 is *little-endian* - i.e. less significant data is always held in lower addresses. This applies to bits in bytes, bytes in words and words in memory. Hence, in a word of data representing an integer, one byte is more significant than another if its byte selector is larger.

Figure 3.1 shows the ordering of bytes in words for the ST20.

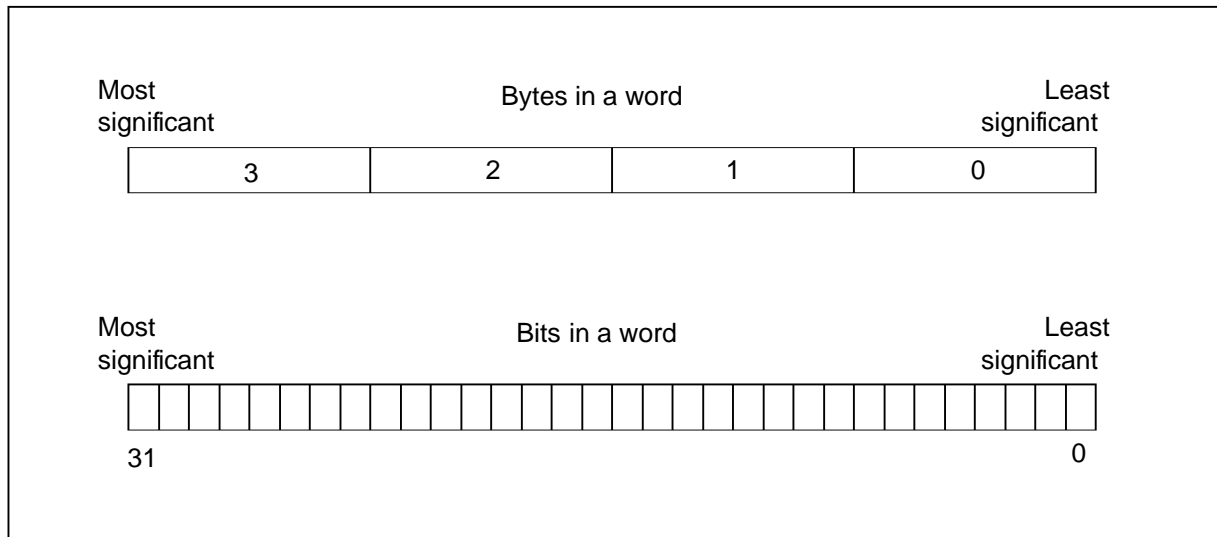


Figure 3.1 Bytes and bits in words

For example, the most significant bit of a word is bit 31, and the most significant byte is byte 3, consisting of bits 24 to 31. This ordering is compatible with Intel processors, but not Motorola or SPARC.

For compatibility with other devices, a *swap32* instruction is provided to reverse the order of bytes within a word.

3.1.2 Signed integers and sign extension

A signed object is stored in twos-complement format. A signed value may be represented by an object of any size. Most commonly a signed integer is represented by a single word, but as explained, it may be stored, for example, in a 64-bit object, a 16-bit object, or an 8-bit object. In each of these formats, all the bits within the object contain useful information.

The length of the object that stores a signed value can be increased, so that the object size is increased without changing the value that is represented. This operation is known as *sign extension*. All the extra bits that are allocated for the larger object, are meaningful to the value of the signed integer; they must therefore be set to the appropriate value. The value for all these extra bits is the same as the value of the most significant bit - i.e. the sign bit - of the smaller object. The ST20-C1 provides instructions that sign extend byte and half-word objects to words.

The example shown in Figure 3.2 shows how the value -10 is stored in a 32-bit register, either as an 8-bit object or as a 32-bit object. In this case, bits 31 to 8 are meaningful for the 32-bit object but not for the 8-bit object. These bits are set to 1 in the 32-bit object.

3.2 Memory

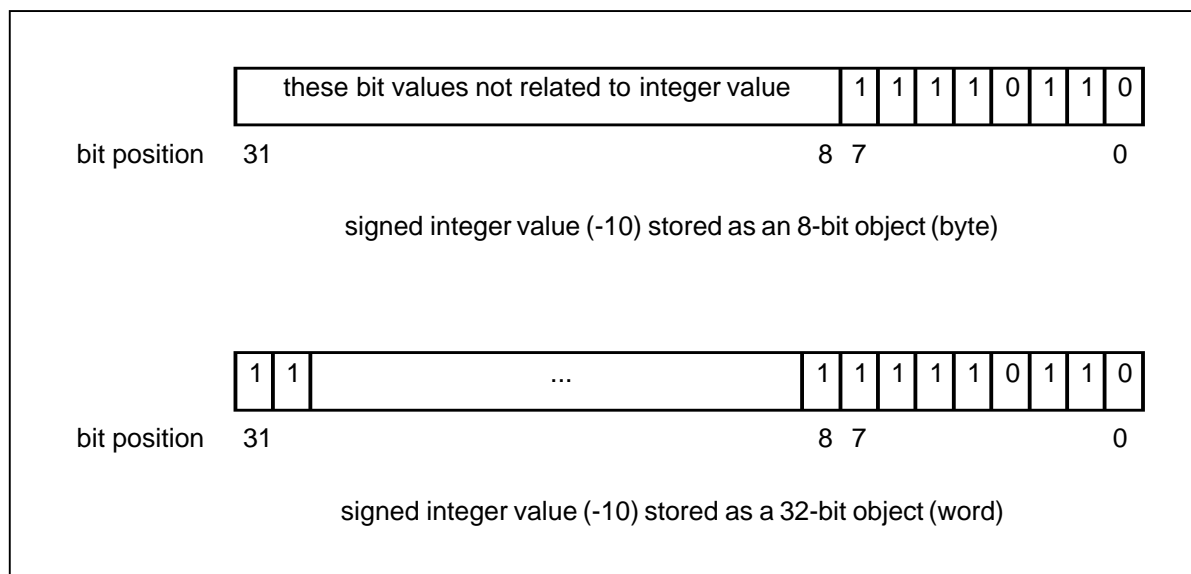


Figure 3.2 Storing a signed integer in different length objects

3.2 Memory

The ST20 processor is a 32-bit word machine, with byte addressing and a 4 Gbyte address space. This section explains how data is arranged in that address space. The address of an object is the address of the base, i.e. the byte with the lowest address.

3.2.1 Word address and byte selector

A machine address, or pointer, is a single word of data which identifies a byte in memory - i.e. a byte address. It comprises two parts, a word address and a byte selector. The byte selector occupies the two least significant bits of the word; the word address the thirty most significant bits.

An address is treated as a signed value, the range of which starts at the most negative integer and continues, through zero, to the most positive integer. This enables the standard arithmetic and comparison functions to be used on pointer values in the same way that they are used on numerical values.

Certain values can never be used as pointers because they represent reserved addresses at the bottom of memory space. They are reserved for use by the processor and initialization. A full list of names and values of constants used in this manual is given in Appendix A.

In particular, the null process pointer (known as *NotProcess*) has the value *MostNeg*, since zero could be a valid process address.

3.2.2 Alignment

A data object is said to be *word-aligned* if it is at an address with a byte selector of zero, i.e. the full address of the object is divisible by 4. Similarly, a data object is said to

be *half-word-aligned* if it is at an address with an even byte selector, i.e. the full address of the object is divisible by 2.

Word objects, including addresses, are normally stored word-aligned in memory. This is usually desirable to make the best use of any 32-bit wide memory. Also most instructions that involve fetching data from or storing data into memory, use word aligned addresses and load or store four contiguous bytes.

However, there are some instructions that can manipulate part of a word. A half-word object is normally half-word-aligned, so it can be stored either in the least significant 16 bits of a word or in the most significant 16 bits. A data item that is represented in two contiguous words is called a *double word* object and is normally word-aligned.

3.2.3 Ordering of information in memory

Data is stored in memory using the little-endian rule. Objects consisting of more than one byte are stored in consecutive bytes, with the least significant byte at the lowest address and the most significant at the highest address.

Figure 3.3 shows the ordering of bytes in words in memory. If X is a word-aligned address then the word at X consists of the bytes at addresses X to $X+3$, where the byte at X is the least significant byte and the byte at $X+3$ is the most significant byte of the word.

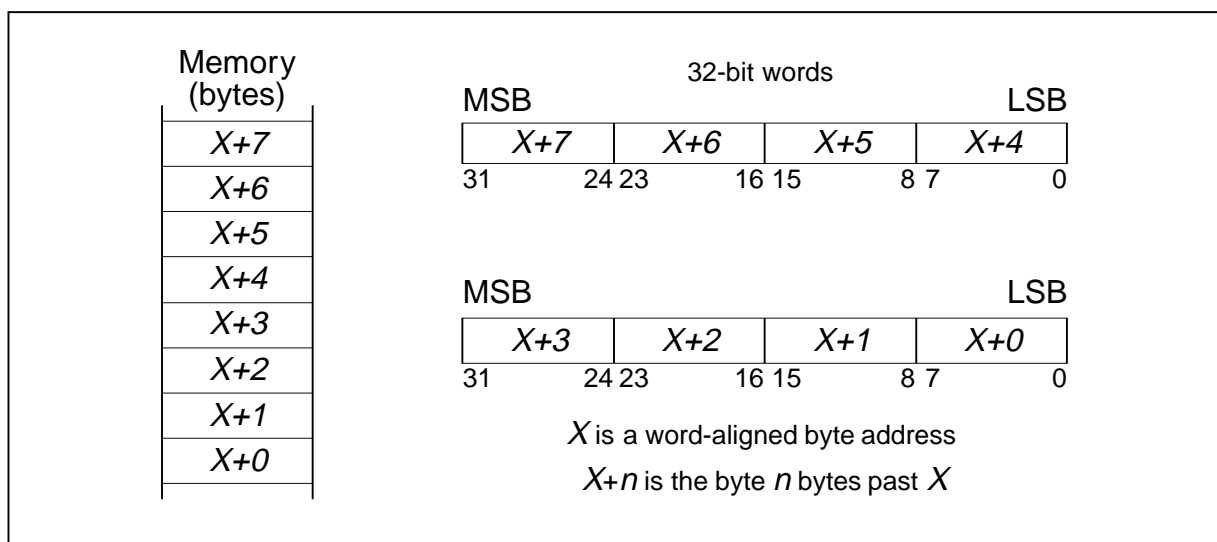


Figure 3.3 Bytes in words in memory

3.2.4 Work space

The ST20-C1 uses a stack-based data structure in memory to hold the local working data of a program, called the work space. The work space is a word-aligned collection of 32-bit words pointed to by the work space pointer register (**Wptr**).

The programmer's model is that all local data is held in the work space, i.e. in memory, and must be brought into the evaluation stack to be operated on, and then written back from the evaluation stack to the work space.

3.3 Registers

An implementation of the ST20-C1 core may include a *register cache*. This provides a mechanism to accelerate access to local work space without changing the programmer's model of how the work space operates or impacting either the excellent code density or low interrupt latency associated with a stack-based instruction set.

3.3 Registers

This section introduces the ST20-C1 core registers that are visible to the programmer. Seven registers, known as process state registers, define the local state of the executing process. These registers are preserved through exceptions. One other register is provided for performing input/output, and is not preserved through exceptions. All registers are 32-bit. Each instruction explicitly refers to specific registers, as described in the instruction definitions.

The state of an executing process at any instant is defined by the contents of the machine registers listed in Table 3.1. The registers are illustrated in Figure 3.4 and described in the rest of this section.

Register	Description
Areg	Evaluation stack register A
Breg	Evaluation stack register B
Creg	Evaluation stack register C
Iptr	Instruction pointer register, pointing to the next instruction to be executed
Status	Status register
Wptr	Work space pointer, pointing to the stack of the currently executing process
Tdesc	Task descriptor
IOreg	Input and output register

Table 3.1 Processor registers

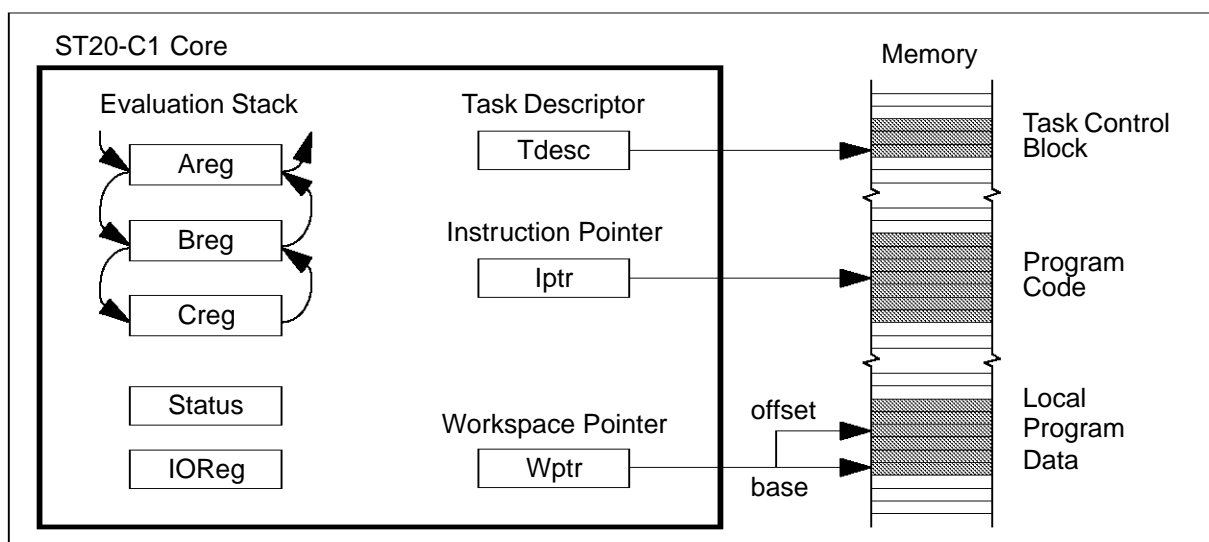


Figure 3.4 Register set

3.3.1 Evaluation stack

The registers **Areg**, **Breg** and **Creg** are organized as a three register evaluation stack, with **Areg** at the top. The evaluation stack is used for expression evaluation and to hold operands and results of instructions. Generally, instructions may pop values from or push values onto the evaluation stack or both, and do not address individual evaluation stack registers.

Pushing a value onto the stack means that the value initially in **Breg** is pushed into **Creg**, the value in **Areg** is pushed into **Breg** and the new value is put in **Areg**. Popping a value from the stack means that a value is taken from **Areg**, the value initially in **Breg** is popped into **Areg**, and the value in **Creg** is popped into **Breg**. The value left in **Creg** varies between instructions, but is generally the value initially in the **Areg**. These actions are illustrated in Figure 3.5 and Figure 3.6.

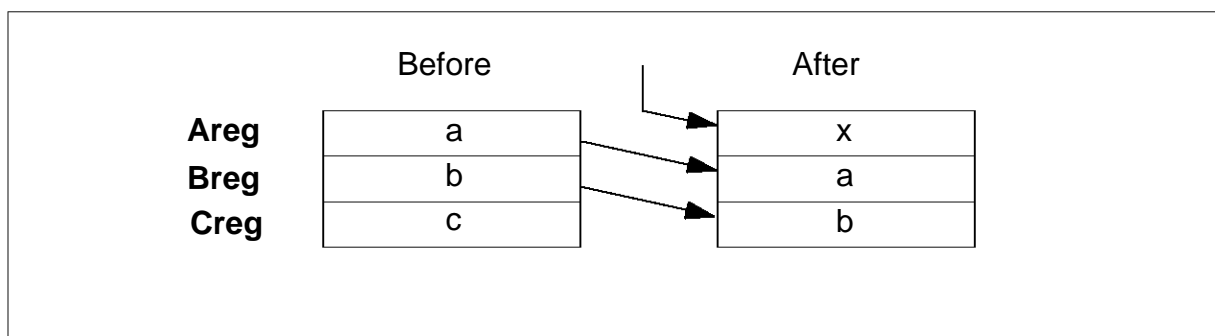


Figure 3.5 Pushing a value x onto the evaluation stack

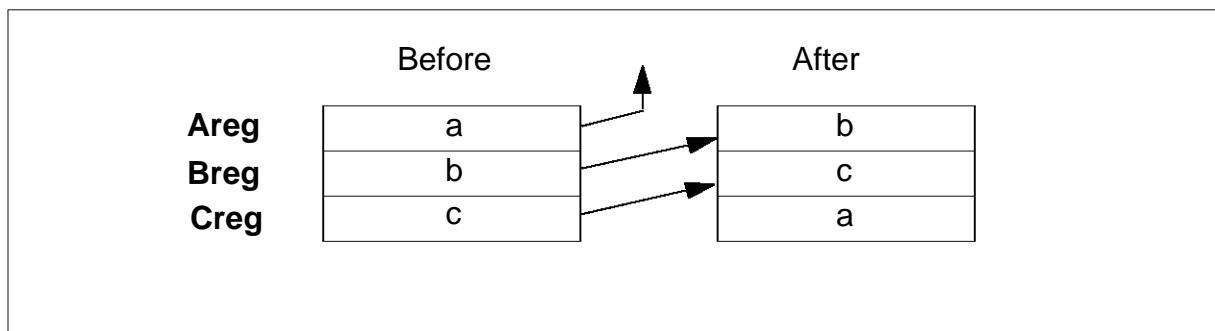


Figure 3.6 Popping a value from the evaluation stack

3.3.2 Status register

The status register contains status bits which describe the current state of the executing process and any errors which may have been detected. Initially the status register is set to the value given in Table 7.3.

The contents of the status register are summarized in Table 3.2 and described in more detail in the following paragraphs. Generally the status register is local except for the

3.3 Registers

Bit numbers	Full name	Meaning when set or meaning of value
0 - 7	mac_count	Multiply-accumulate number of steps.
8 - 10	mac_buffer	Multiply-accumulate data buffer size code.
11 - 12	mac_scale	Multiply-accumulate scaling code.
13	mac_mode	Multiply-accumulate accumulator format code.
14	global_interrupt_enable	Enable external interrupts until explicitly disabled.
15	local_interrupt_enable	Enable external interrupts. Clearing this bit disables interrupts until the current process is descheduled.
16	overflow	An arithmetic operation gave a positive overflow.
17	underflow	An arithmetic operation gave a negative overflow.
18	carry	An arithmetic operation produced a carry.
19	user_mode	A user process is executing.
20	interrupt_mode	An interrupt handler is executing or trapped.
21	trap_mode	A trap handler is executing.
22	sleep	The processor is due to go to sleep.
23	reserved	Reserved.
24	start_next_task	The CPU must start executing a new process.
25	timeslice_enable	Timeslicing is enabled.
26 - 31	timeslice_count	Timeslice counter.

Table 3.2 Status register bits

global interrupt enable and timeslice enable, which are global and carried from one process to another across a context switch.

- The **mac_count**, **mac_buffer**, **mac_scale** and **mac_mode** fields are used by the multiply-accumulate instructions to hold initialization data which must be saved when an exception occurs. See Chapter 5 for details of multiply accumulation.
- The **global_interrupt_enable** bit enables external interrupts. Interrupts remain enabled or disabled until explicitly disabled or enabled again. This bit is global and is maintained when a process is descheduled.
- The **local_interrupt_enable** enables external interrupts. Clearing this bit disables external interrupts until the current process is descheduled. This is needed when a process delegates part of its processing to a peripheral and then deschedules until completion, as described in section 4.11.3.
- **Overflow**, **underflow** and **carry** bits relating to arithmetic state are kept in the status word.

The ST20-C1 maintains “sticky” bits in the status word which indicate whether an overflow or underflow has occurred. This allows a complete expression to be evaluated before testing whether an overflow has occurred. Overflow and underflow are chosen as they apply both to addition as well as multiply as opposed to a more traditional method of replicating two bits out of the carry

chain. In addition, they allow for saturated arithmetic to be implemented relatively easily.

A (non-sticky) carry bit is provided to allow efficient implementation of long addition and subtraction. The carry bit is only manipulated by the *addc* and *subc* instructions allowing the other add instructions to be used in address formation of multi-word values where carry propagation is required so that the carry is not lost in the address formation evaluations.

- The **user_mode** bit indicates when the machine is handling a user process, i.e. a process which is not an exception handler. The **interrupt_mode** bit indicates when the machine is handling an interrupt, or a trap from an interrupt handler and the **trap_mode** bit indicates when the machine is executing a trap handler. An operating system may need to distinguish between modes to allow it to perform scheduling activities from a trap handler. These bits are also required to enable the *eret* instruction to determine whether a signal to the interrupt controller is required.
- The **sleep** bit indicates that the CPU is due to go to sleep, i.e. to turn off its clocks and go into low power mode. This bit is set when the CPU detects there is no user process to execute and is cleared when the CPU goes to sleep.
- The **start_next_task** bit when set causes the processor to attempt to run the next process from the scheduling queue.
- The **timeslice_enable** bit and **timeslice_count** field are used for timeslicing, as described in section 7.4.

The instructions to use the status register are described in section 4.12.

3.3.3 The work space pointer

All programs need somewhere to store local working data, e.g. local variables in the application code. In the ST20 architecture, this local storage is termed the *work space* of the program.

The **Wptr** register is the local work space pointer, which holds the address of the stack of the executing process. The stack is downward pointing, so space is allocated by moving the **Wptr** to a lower address. This address is word aligned and therefore has the two least significant bits set to zero. When a process is descheduled, the **Wptr** is stored as part of the process descriptor block, which is pointed to by **Tdesc**.

The **Wptr** is used as a base for addressing *local* variables. A word offset from the **Wptr** is the operand for the instructions *ldl* (load local), *stl* (store local) and *ldlp* (load local pointer).

The ST20-C1 simplifies the normal stack scheme by decoupling the load/store action from the pointer update:

- Load-local and store-local instructions access values in the work space with addresses relative to the **Wptr**, but do not change the value of **Wptr**.
- Separate instructions (*ajw*, *gajw*) are provided to update the work space

3.4 Instruction encoding

pointer by any amount in one step without needing a series of increments or decrements.

On calling a function or procedure, the **Wptr** is normally decreased to a lower address to allocate space for the parameters and local variables of the function. This is performed using the instruction *ajw*. The **Wptr** is returned to its initial value before returning from the function to free the local work space.

3.3.4 The task descriptor

The task descriptor **Tdesc** points to the process descriptor block for the currently executing process. The value held in the **Tdesc** becomes the process identifier when the process is not executing.

The process descriptor block is a block of memory whose contents depend on the state of the process. It will generally hold the saved **Wptr** and **lptr** for the process, and may hold a link to the next process if the process is in a queue of waiting processes. The process descriptor block is described in section 7.2.

3.3.5 IO register

The bits of the **IOreg** are mapped to external connections on the ST20-C1 core. They may be used to signal to, or read signals from, peripherals on or off chip. The *io* instruction is used to read and write to the **IOreg** and is described in section 4.11. The **IOreg** is global, and remains unchanged by any context switch. The bits of the **IOreg** are defined in Table 3.3.

Bits	Purpose
0-15	Output data
16-31	Input data

Table 3.3 **IOreg** bits

In some ST20 variants, some bits of the IO register may be reserved for system use. The reserved bits will be the most significant bits of the appropriate half word. The number of any such bits is given in the data sheet for each variant.

3.4 Instruction encoding

The ST20-C1 is a zero-address machine. Instruction operands are always implicit and no bits are needed in the instruction representation to carry address or operand location information. This results in very short instructions and exceptionally high code density.

The instruction encoding is designed so that the most commonly executed instructions occupy the least number of bytes. This reduces the size of the code, which saves memory and reduces the memory bandwidth needed for instruction fetching. This section describes the encoding mechanism.

A sequence of single byte *instruction components* is used to encode an instruction. The ST20 interprets this sequence at the instruction fetch stage of execution. Most

programmers, working at the level of microprocessor assembly language or high-level language, need not be aware of the existence of instruction components and do not generally need to consider the encoding.

This section has been included to provide a background. Appendix C discusses consequential issues which need to be considered in order to implement a code generator.

3.4.1 An instruction component

Each instruction component is one byte long, and is divided into two 4-bit parts. The four most significant bits of the byte form a *function code*, and the four least significant bits are used to build an *instruction data value* as shown in Figure 3.7.

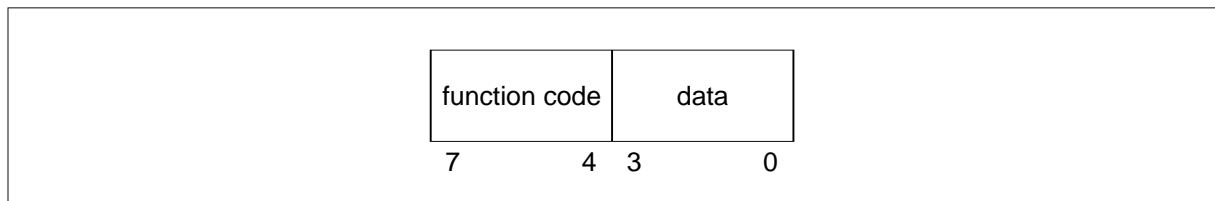


Figure 3.7 Instruction format

This representation provides for sixteen function code values (one for each function), each with a data field ranging from 0 to 15.

Instructions that specify the instruction directly in the function code are called *primary instructions* or *functions*. There are 13 primary instructions, and the other three possible function code values are used to build larger data values and other instructions. Two function code values, *prefix* and *negative prefix*, are used to extend the instruction data value by prefixing. One function code *operate (opr)* is used to specify an instruction indirectly using the *instruction data value*. *opr* is used to implement *secondary instructions* or *operations*.

3.4.2 The instruction data value and prefixing

The data field of an instruction component is used to create an instruction data value. Primary instructions interpret the instruction data value as the operand of the instruction. Secondary instructions interpret it as the operation code for the instruction itself.

mnemonic	name
<i>prefix n</i>	prefix
<i>negative prefix n</i>	negative prefix

Table 3.4 Prefixing instruction components

The instruction data value is a signed integer that is represented as a 32-bit word. For each new instruction sequence, the initial value of this integer is zero. Since there are only 4 bits in the data field of a single instruction component, it is only possible for most instruction components to initially assign an instruction data value in the range 0 to 15. Prefix components are used to extend the range of the instruction data value.

3.4 Instruction encoding

One or more prefixing components may be needed to create the full instruction data value. The prefixes are shown in Table 3.4 and explained below.

All instruction components initially load the four data bits into the least significant four bits of the instruction data value.

prefix loads its four data bits into the instruction data value, and then shifts this value up four places. Consequently, a sequence of one or more prefixes can be used to extend the data value of the following instruction to any positive value. Instruction data values in the range 16 to 255 can be represented using one *prefix*.

negfix is similar, except that it complements all 32 bits of the instruction data value before shifting it up, thus changing the sign of the instruction data value. Consequently, a sequence of one or more *prefixes* with one *negfix* can be used to extend the data value of a following instruction to any negative value. Instruction data values in the range -256 to -1 can be represented using one *negfix*.

When the processor encounters an instruction component other than *prefix* or *negfix*, it loads the 4-bit data field into the instruction data value. The instruction encoding is now complete and the instruction can be executed. The instruction data value is then cleared so that the processor is ready to fetch the next instruction component, by building a new instruction data value.

For example, to load the constant 0x11, the instruction *ldc 0x11* is encoded with the sequence:

prefix 1; ldc 1

The instruction *ldc 0x2A68* is encoded with the sequence:

prefix 2; prefix A; prefix 6; ldc 8

The instruction *ldc -1* is encoded with the sequence:

negfix 0; ldc F

3.4.3 Primary Instructions

Research has shown that computers spend most time executing a small number of instructions such as:

- instructions to load and store from a small number of 'local' variables;
- instructions to add and compare with small constants; and
- instructions to jump to or call other parts of the program.

For efficiency, in the ST20 these are encoded directly as primary instructions using the function field of an instruction component.

Thirteen of the instruction components are used to encode the most important operations performed by any computer executing a high level language. These are used (in conjunction with zero or more prefixes) to implement the primary instructions. Primary instructions interpret the instruction data value as an operand for the instruction. The mnemonic for a primary instruction always includes this operand, shown in this manual as *n*.

The mnemonics and names for the primary instructions are listed in Table 3.5.

mnemonic	name
<i>adc n</i>	add constant
<i>ajw n</i>	adjust work space
<i>fcall n</i>	function call
<i>cj n</i>	conditional jump
<i>eqc n</i>	equals constant
<i>j n</i>	jump
<i>ldc n</i>	load constant
<i>ldl n</i>	load local
<i>ldlp n</i>	load local pointer
<i>ldnl n</i>	load non-local
<i>ldnlp n</i>	load non-local pointer
<i>stl n</i>	store local
<i>stnl n</i>	store non-local

Table 3.5 Primary instructions

3.4.4 Secondary instructions

The ST20 encodes all other instructions, known as *secondary instructions*, indirectly using the instruction data value.

mnemonic	name
<i>opr n</i>	operate

Table 3.6 Operate instruction

The function code *opr* causes the instruction data value to be interpreted as the operation code of the instruction to be executed. This selects an operation to be performed on the values held in the evaluation stack, so that a further 16 operations can be encoded in a single byte instruction. The *prefix* instruction component can be used to extend the instruction data value, allowing any number of operations to be encoded.

Secondary instructions do not have an operand specified by the encoding, because the instruction data value has been used to specify the operation.

To ensure that programs are represented as compactly as possible, the operations are encoded in such a way that the most frequently used secondary instructions are represented without using prefix instructions.

For example, the instruction *add* is encoded by:

opr 4

The instruction *and* is encoded by:

opr F9

3.4 Instruction encoding

which is in turn encoded with the sequence:

prefix F; opr 9

3.4.5 Summary of encoding

The encoding mechanism has important consequences.

- It produces very compact code.
- It simplifies language compilation, by providing a completely uniform way of allowing a primary instruction to take an operand of any size up to the processor word-length.
- It allows these operands to be represented in a form independent of the word-length of the processor.
- It enables any number of secondary instructions to be implemented.

To aid clarity and brevity, prefix sequences and the use of *opr* are not explicitly shown in this guide. Each instruction is represented by a mnemonic, and for primary instructions an item of data, which stands for the appropriate instruction component sequence. Hence the examples above would be just shown as: *ldc 17*, *add*, and *and*. Where appropriate, an expression may be placed in a code sequence to represent the code needed to evaluate that expression.

4 Using ST20-C1 instructions

This chapter describes the purpose for which the sequential instructions are intended, except for the multiply-accumulate instructions, which are described in Chapter 5. These instructions are described in the context of their intended use. Some instructions are designed for use in a particular sequence of instructions, so this chapter describes those sequences. Instructions for exceptions are described in Chapter 6 and multi-tasking instructions are described in Chapter 7.

The architecture of the ST20-C1, including the registers and memory arrangement, is described in Chapter 3.

4.1 Manipulating the evaluation stack

The evaluation stack consists of the registers **Areg**, **Breg** and **Creg**. The general action of the evaluation stack is described in section 3.3.1.

Instructions are provided for shuffling and re-ordering the values on the evaluation stack, as listed in Table 4.1.

Mnemonic	Name
<i>rot</i>	rotate stack
<i>arot</i>	anti-rotate stack
<i>dup</i>	duplicate stack
<i>rev</i>	reverse stack

Table 4.1 Evaluation stack manipulation instructions

rot pops the value from **Areg** off the evaluation stack and rotates it into **Creg**, and *arot* pushes the value from **Creg** onto the stack. *rev* swaps the **Areg** and **Breg**, and *dup* pushes a copy of **Areg** onto the stack.

Table 4.2 shows how each of these affects the evaluation stack. Each row shows the contents of the evaluation stack after one of these instructions is executed if the initial values of the **Areg**, **Breg** and **Creg** are a, b and c respectively.

Instruction	Areg	Breg	Creg
<i>rot</i>	b	c	a
<i>arot</i>	c	a	b
<i>rev</i>	b	a	c
<i>dup</i>	a	a	b

Table 4.2 Evaluation stack manipulation

Many instructions leave the initial **Areg** in **Creg**. This value may be restored into the **Areg** by using *arot*.

4.2 Loading and storing

The loading and storing instructions are listed in Table 4.3.

Mnemonic	Name	Description
<i>ldc n</i>	load constant	Load the constant <i>n</i> .
<i>ldl n</i>	load local	Load the value from <i>n</i> words above Wptr .
<i>stl n</i>	store local	Store a value to <i>n</i> words above Wptr .
<i>ldnl n</i>	load non-local	Load the value from <i>n</i> words above Areg .
<i>stnl n</i>	store non-local	Store a value to <i>n</i> words above Areg .
<i>lbinc</i>	load byte and increment	Load a byte and increment the address by 1 byte.
<i>sbinc</i>	store byte and increment	Store a byte and increment the address by 1 byte.
<i>lsinc</i>	load sixteen and increment	Load a half word and increment the address by 2 bytes.
<i>lsxinc</i>	load sixteen sign extended and increment	Load a half word and sign extend to 32 bits and increment the address by 2 bytes.
<i>ssinc</i>	store sixteen and increment	Store a half word and increment the address by 2 bytes.
<i>lwinc</i>	load word and increment	Load a word and increment the address by 4 bytes.
<i>swinc</i>	store word and increment	Store a word and increment the address by 4 bytes.

Table 4.3 Loading and storing instructions

On the ST20, the term *loading* means pushing a value onto the evaluation stack. The value to be loaded may be a value read from memory, a constant, a copy of another register or a calculated value. *Storing* means popping a value from the evaluation stack. The value may be written into memory or written into another register. The evaluation stack is described in section 3.3, and evaluation of expressions is described in section 4.3.

Relative addresses are used for accessing memory in order to reduce code size, as the operand values are smaller than full machine addresses. Data structures are word-aligned, so relative addresses can be word offsets, reducing the operand size further.

The most common operations performed by a program are loading and storing of a small number of variables, and loading small literal values.

4.2.1 Loading constants

One primary instruction *ldc* is provided for loading a general constant, for initializing a variable or register or for a constant in an expression.

4.2.2 Local and non-local variables

When loading from and storing to memory, the ST20 distinguishes between local and non-local addressing. Local addressing means that the address is given as a word offset from the **Wptr**. Non-local addressing means that the address is given as a word offset from the **Areg**. In practice, the **Wptr** points to the stack, so local addressing is

4.2 Loading and storing

normally used for local variables on the stack while non-local addressing is normally used for all other variables.

The primary instructions *ldl* and *stl* perform loading and storing of local variables. For example to load a value *x* words above the **Wptr** and write to a location *y* words above the **Wptr**:

```
ldl x;  
stl y;
```

The primary instructions *ldnl* and *stnl* perform loading and storing of non-local variables. For example, to load a value *x* above a base address *x_base* and store to a location *y* words above *y_base*, where *x_base* and *y_base* are held in local variables:

```
ldl x_base; ldnl x;  
ldl y_base; stnl y;
```

Note that for the purposes of this manual, *ld X* denotes loading the value from a variable *X*, where *X* may be a local or non-local variable, so either *ldl* or *ldnl* may be used as appropriate. Similarly *st X* denotes storing a value into a variable *X*, where *X* may be a local or non-local variable, so either *stl* or *stnl* may be used.

4.2.3 Byte and half-word values

Instructions are provided for loading and storing byte and half-word variables. In each case, the address is initially in the **Areg** and is incremented by the size of the object, so that repeated loads and stores can be used to copy a block of memory.

The load instructions place the loaded value in the **Areg**, the incremented address in the **Creg** and leave the **Breg** unaffected. The store instructions write the initial **Breg** into memory at the address in the **Areg**, leaving the incremented address in the **Breg**, the initial **Creg** in the **Areg** and the initial **Breg** pushed down to the **Creg**.

Byte loading and storing

lbinc loads the byte at the address in **Areg**, into the evaluation stack. *lbinc* replaces the address in **Areg** with the byte stored at that address, treating it as an unsigned integer by setting the twenty-four most significant bits in **Areg** to 0. The incremented address is left in **Creg**.

sbinc writes the least significant byte in **Breg** to the location addressed by **Areg**. The address is incremented by 1 and put in the **Breg**.

Half-word loading and storing

lsinc and *lsxinc* load the half-word object at the address in **Areg**, into the evaluation stack. *lsinc* replaces the address in **Areg** with the half word, treating it as an unsigned integer by setting the sixteen most significant bits in **Areg** to 0. *lsxinc* is similar to *lsinc*, but treats the half-word as a signed integer in twos-complement format, and hence sign extends the representation by setting the sixteen most significant bits in **Areg** to the same value as the most significant bit of the half-word object. Sign extension is discussed in section 4.4.6.

ssinc writes the half word in the two least significant bytes of **Breg** to the location addressed by **Areg**.

4.2.4 Memory block copy

A block memory copy may be implemented using the instructions *lwinc* and *swinc*. These instructions load or store a word, and increment the addresses used.

To copy *n* bytes from *source* to *destination*, where *source* and *destination* are both word-aligned, a loop should be written, using the temporary variable *limit*, as in the following code:

```

        ld source; ld n; ld destination
        add; stl limit
LOOP:   lwinc; rev; swinc
        ldl limit; arot
        gt; cj END; j LOOP;
END:

```

This is the most efficient method of copying, since it reads and writes full words, making the best use of any 32-bit memory. However, this is not always possible if the alignment of the source and destination blocks are different. In that case the byte or half-word load and store should be used.

4.3 Expression evaluation

Expression evaluation and address calculation is performed using the evaluation stack. For example, the evaluation of operations with two integer operands is performed by instructions that operate on the values of **Areg** and **Breg**. The result is left in **Areg**.

Arithmetic and boolean calculations are considered in sections 4.4 and 4.7 respectively. This section describes how the evaluation stack is used. Loading and storing instructions are described in section 4.2.

In this and subsequent sections, in examples of assembly code, a single letter or identifier written as an instruction is either an expression or a segment of code. If it is an expression then it means ‘evaluate the expression and leave the result in the **Areg**’.

4.3.1 Using the evaluation stack

A compiler normally loads a constant expression *c* using *ldc*:

```
ldc c
```

Loading from a constant table is described in section 4.3.3.

An expression consisting of a single local variable is loaded using

```
ldl x
```

Methods for loading non-local variables are discussed in section 4.2, and array elements in section 4.5.

4.3 Expression evaluation

Evaluation of expressions sometimes requires the use of temporary variables in the process work space, but the number of these can be minimized by careful choice of the evaluation order. The details of how this is achieved by a compiler are described in Appendix C in section C.3.

4.3.2 Loading operands

The three registers of the evaluation stack are used to hold operands of instructions. Evaluation of an operand or parameter may involve the use of more than one register. Care is needed when evaluating such operands to ensure that the first operand to be loaded is not pushed off the bottom of the evaluation stack by the evaluation of later operands. The processor does not detect evaluation stack overflow.

Three registers are available for loading the first operand, two registers for the second and one for the third. Consequently, the instructions are designed so that **Creg** holds the operand which, on average, is the most complex, and **Areg** the operand which is the least complex.

In some cases, it is necessary to evaluate the **Areg** and **Breg** operands in advance, and to store the results in temporary variables. This can sometimes be avoided using the reverse instruction. Any of the following sequences may be used to load the operands *A*, *B* and *C* into **Areg**, **Breg** and **Creg** respectively.

- 1 *C; B; A;*
- 2 *C; A; B; rev;*
- 3 *B; C; rev; A;*
- 4 *A; C; rev; B; rev;*

The choice of loading sequence, and of which operands should be evaluated in advance is determined by the number of registers required to evaluate each of the operands. The algorithm used by compilers is given in Appendix C in section C.4.

4.3.3 Tables of constants

The ST20-C1 instruction set has been optimized so that the loading of small constants can be coded compactly — for example it allows the loading of constants between 0 and 15 to be coded in a single byte. Analysis of programs shows that such small constants occur markedly more frequently than large constants. However when a large constant does need to be loaded the necessary prefix sequence may be long. Other techniques may be more efficient in these cases.

A simple mechanism to increase the code compactness is to use a table of constants. This is implemented by storing all the long constants into a look-up table. This table and all its constant entries must be aligned on a word boundary. The address of this table is held in a local variable which is used to index the array. Then to load the

constant from the n th entry in the constant table stored at address *constants_ptr* the following code would be used:

```
ldl constants_ptr; ldnl n;
```

where the instruction *ldnl n* is explained in section 4.2.2.

This code sequence only takes 2 bytes, provided *constants_ptr* is less than 16 words from the work space pointer address and there are no more than 16 word-length constants. At worst it is unlikely to take more than 4 bytes. Hence, if a constant takes 4 or more bytes to load using *ldc* then this sequence often improves code compactness — especially if the constant is used more than once.

4.3.4 Assignment

Single words, half words and bytes may be assigned using the load and store instructions described in section 4.2.

Word assignment

If x and y are both single word variables and e is a word valued expression then word assignments are compiled as

$x = y$ compiles to *ld y; st x;*

$x = e$ compiles to *e; st x;*

Byte assignment

If a and b are both single byte variables and e is a byte valued expression then byte assignments are compiled as

$b = a$ compiles to *address(a); lbinc; address(b); sbinc;*

$b = e$ compiles to *e; address(b); sbinc;*

where *address(variable)* is the address of variable. Forming addresses is discussed in section 4.5.

Half word assignment

If a and b are both half-word variables and e is a half-word valued expression then half-word assignments are compiled as

$b = a$ compiles to *address(a); lsinc; address(b); ssinc;*

$b = e$ compiles to *e; address(b); ssinc;*

where *address(variable)* is the address of variable. Forming addresses is discussed in section 4.5.

4.4 Arithmetic

This section describes the use of the arithmetic instructions except for the multiply-accumulate instructions, which are described in Chapter 5, and forming addresses, which is described in section 4.5. Boolean expression evaluation is discussed in

4.4 Arithmetic

section 4.7, and the general principles of expression evaluation are described in section 4.3.

4.4.1 Addition, subtraction and multiplication

Single length signed arithmetic is provided by the operations listed in Table 4.4.

Mnemonic	Name
<i>adc n</i>	add constant
<i>add</i>	add
<i>sub</i>	subtract
<i>mul</i>	multiply
<i>smul</i>	short multiply

Table 4.4 Single length signed integer arithmetic instructions

Each of these instructions except *smul* can signal *overflow* or *underflow* by setting the appropriate bit in the status register. An overflow occurs if the result is greater than *MostPos* and an underflow if it is less than *MostNeg*. If overflow or underflow occurs, then the 32 least significant bits of the full result are left in the **Areg**. The overflow and underflow are 'sticky', so when one has been set, it is not cleared and the other cannot be set by subsequent arithmetic. The overflow and underflow bits may be used for saturated arithmetic, as described in section 4.4.3.

The primary instruction *adc n* adds the constant value *n* to **Areg**. **Breg** and **Creg** are unaffected. This is used for incrementing and decrementing variables and counters.

If *op* is one of *add*, *sub*, *mul* or *smul*, then the instruction sequence

ldl X; ldl Y; op;

evaluates the expression

X op Y

i.e. it takes the value in **Breg** as the left hand operand and the value in **Areg** as the right hand operand, and loads the result into **Areg**. The content of **Creg** is popped into **Breg** and the initial **Areg** is rotated into **Creg**.

smul multiplies two half-word values producing a 32-bit result. It cannot overflow or underflow and is faster than *mul*.

4.4.2 Division and remainder

Division and remainder are performed using the operations listed in Table 4.5.

Mnemonic	Name
<i>divstep</i>	divide step
<i>unsign</i>	unsign argument

Table 4.5 Division and remainder instructions

Each *divstep* generates four bits of the unsigned quotient, so eight *divsteps* are needed for a full 32-bit unsigned division, and will also generate a remainder. The result of the division is the integer division rounded towards zero (truncated). The quotient is left in **Breg**, and the remainder in **Creg**, so a rotation pops the quotient into the **Areg**.

unsign is used to separate the sign from the magnitude of the operands before performing the division. Division is then performed on the magnitudes, and the signs of the results may be derived from the signs of the operands.

Overflow can occur only if the divisor (**Areg**) is zero, or if the dividend (**Breg**) is *MostNeg* and the divisor is -1. *divstep* does not detect these cases, and does not set any status bits, so a check should be applied before performing the division.

The following code sequence performs the integer division *a/b*. The signed quotient is left in **Areg**.

```

a; b; ldc 0; arot; unsign;
arot; unsign; cj POS;
ldc 0; rot;
divstep; divstep; divstep; divstep;
divstep; divstep; divstep; divstep;
rot; not; adc 1;
j END;
POS: rot;
divstep; divstep; divstep; divstep;
divstep; divstep; divstep; divstep;
rot;
END:

```

The following code sequence performs the remainder *a rem b*. The signed remainder is left in **Areg**.

```

a; b; ldc 0; rev; unsign;
eqc 2; arot; unsign; cj POS;
divstep; divstep; divstep; divstep;
divstep; divstep; divstep; divstep;
arot; not; adc 1;
j END;
POS: rot;
divstep; divstep; divstep; divstep;
divstep; divstep; divstep; divstep;
arot;
END:

```

4.4.3 Saturated arithmetic

In saturated arithmetic, when an overflow or underflow occurs the result is set to the most positive or most negative possible result respectively, instead of the least significant bits of the full result. This ensures that the result is as near as possible to the real value and prevents glitches caused by wrap-around.

4.4 Arithmetic

Saturated arithmetic is achieved on the ST20-C1 by evaluating an expression and then performing the *saturate* instruction. If an overflow or underflow has occurred then the corresponding status bit will have been set, which will cause *saturate* to change the value in **Areg** to the most positive or most negative value respectively. *saturate* clears the overflow and underflow bits.

For example, to perform a saturated multiply of *a* and *b*:

```
ld a; ld b;  
mul; saturate;
```

4.4.4 Unary minus

The expression $(-e)$ can be evaluated with overflow signalling by:

```
e; not; adc 1;
```

or

```
ldc 0; e; sub;
```

The first sequence, using *not*, requires one less stack register than the second. *not* is a bitwise inversion which is described in section 4.8.

4.4.5 Long arithmetic

The long arithmetic instructions are listed in Table 4.6.

Mnemonic	Name
<i>addc</i>	add with carry
<i>subc</i>	subtract with carry
<i>umac</i>	unsigned multiply accumulate

Table 4.6 Long arithmetic instructions

Multiple length addition and subtraction

Multiple length addition or subtraction are performed using *addc* and *subc*, executed once for each word of the result. For both instructions, the carry (or borrow) is held in the carry bit of the status register. This keeps the carrying separate from overflow, so address calculations may be safely performed using *add*, *sub* and *wsub* without affecting the carry.

The *addc* instruction forms $(\mathbf{Breg} + \mathbf{Areg}) + \mathbf{Status}_{\text{carry}}$ leaving the least significant word of the result in **Areg** and the most significant (carry) bit in the carry bit of the status register. The **Areg** is rotated into the **Creg**.

Similarly, the *subc* instruction forms $(\mathbf{Breg} - \mathbf{Areg}) - \mathbf{Status}_{\text{carry}}$ leaving the least significant word of the result in **Areg** and the borrow bit in the carry bit of the status register. The **Areg** is rotated into the **Creg**.

Addition of two double length unsigned values, *X* and *Y*, giving *Z*, without overflow signalling can therefore be compiled as follows

```
ldc 0;
ldl Xlo; ldl Ylo; addc; stl Zlo;
ldl Xhi; ldl Yhi; addc; stl Zhi
```

The subscripts 'lo' and 'hi', used here and in subsequent text, specify the least and most significant word respectively of the double word variable with which they are associated.

Subtraction of two double length values, Y from X giving Z, without overflow signalling is compiled as

```
ldc 0;
ldl Xlo; ldl Ylo; subc; stl Zlo;
ldl Xhi; ldl Yhi; subc; stl Zhi
```

Overflow signalling for signed arithmetic may be added by performing an extra *addc* or *subc* to produce a final word which contains only a sign (0 for positive or -1 for negative) unless an overflow has occurred. For example, the following code could be used to perform double length signed addition with overflow signalling:

```
clear carry, overflow and underflow status bits
ldl Xlo; ldl Ylo; addc; stl Zlo;
ldl Xhi; ldl Yhi; addc; stl Zhi;
ldc 0; dup; addc;
dup; adc #7fffff;
rev; adc #8000001;
```

- overflows if and only if carry word > 0
- underflows if and only if carry word < -1

Multiple length multiplication

The *umac* instruction multiplies two single word unsigned operands in **Areg** and **Breg**, and adds the single word carry operand in **Creg** to form a double length unsigned result. The more significant (carry) word of the result is left in **Breg**, the less significant in **Areg**. No overflow can be signalled by this instruction.

Multiplication of a single length unsigned value X by a double length unsigned value Y (leaving the 'carry' in **Areg**) can be performed by:

```
ldc 0;
ldl X; ldl Ylo; umac; stl Zlo;
ldl X; ldl Yhi; umac; stl Zhi
```

Double length unsigned multiplication is more complex. The product of two unsigned double length words X and Y can be expressed as:

$$\begin{aligned} X * Y &= (X_{hi} * 2^{32} + X_{lo}) * (Y_{hi} * 2^{32} + Y_{lo}) \\ &= (X_{hi} * Y_{hi}) * 2^{64} + (X_{hi} * Y_{lo} + X_{lo} * Y_{hi}) * 2^{32} + (X_{lo} * Y_{lo}) \end{aligned}$$

This can be coded as follows:

```
ldc 0;
ldl Xlo; ldl Ylo; umac; stl Z0
ldl Xlo; ldl Yhi; umac; rev; stl Z2
```

```
ldl Xhi; ldl Ylo; umac; stl Z1;  
ldl Xhi; ldl Yhi; umac; rev; stl Z3;  
ldc 0; rev; ldl Z2; addc; stl Z2;  
ldl Z3; addc; stl Z3
```

This gives a quadruple length unsigned result Z , where Z_0 is the least significant and Z_3 the most significant word of Z .

4.4.6 Object length conversion

Object length conversion operations are provided by the instructions listed in Table 4.7.

Mnemonic	Name
<i>xbword</i>	sign extend byte to word
<i>xsword</i>	sign extend sixteen to word

Table 4.7 Object length conversion instructions

Section 3.1 explains that data can be represented in data objects of various sizes. This section describes the instructions that can be used to convert between these representations.

Most of the ST20-C1 integer arithmetic instructions operate on signed integers held in the evaluation stack registers as 32-bit objects, and produce results in this form. Object length conversion is important for conversion of high level language data types. The ST20-C1 therefore provides instructions that allow a byte or half-word signed integer to be sign extended to 32-bits by copying the sign bit to all the bits that were previously not significant, as shown in Figure 3.2. The sign extension is performed on the value in the **Areg** and the result is placed in the **Areg**. The other registers are not affected.

xbword extends a signed byte to a word by copying bit 7 into bits 8 to 31. *xsword* extends a signed half-word to a word by copying bit 15 into bits 16 to 31.

lsxinc loads a sixteen bit value, sign extends it to 32 bits and increments the address by two bytes. This is the same as:

```
lsinc; xsword;
```


4.5 Forming addresses

The addressing instructions provide access to items in data structures using short sequences of single byte instructions. These instructions are listed in Table 4.8.

Mnemonic	Name	Meaning
<i>ldlp n</i>	load local pointer	Load the value Wptr + 4 <i>n</i> .
<i>ldnlp n</i>	load non-local pointer	Load the value Areg + 4 <i>n</i> .
<i>ldpi n</i>	load pointer to instruction	Load the value Iptra + <i>n</i> .
<i>wsub</i>	word subscript	Load the value Areg + 4. Breg .

Table 4.8 Addressing instructions

4.5.1 The address of a variable

The absolute address of a local work space location is loaded using the *ldlp* primary instruction. *ldlp* 0 can be used to load the value in the **Wptr**.

The *ldnlp* primary instruction is provided to calculate the absolute address of a non-local variable.

The meaning of local and non-local is described in section 4.2.

4.5.2 The address of an instruction

The address of a location in the program being executed can be obtained by the *ldpi* operation as follows. The address of the location *x* bytes past the next instruction (which is itself pointed to by the instruction pointer register) can be pushed onto the evaluation stack by

ldc x; ldpi

For example, the address of a label *L* can be loaded by

ldc (L-M); ldpi
M:

where the label *M* is the address of the instruction that follows the *ldpi* instruction. First the offset in bytes from *M* to *L* is loaded into **Areg**. The *ldpi* then uses this offset and the value in the instruction pointer register (which will be the address of label *M*) to load the address of label *L* into **Areg**. This technique is useful for generating relocatable code. **Breg** and **Creg** are unaffected.

4.5.3 Arrays

The *wsub* instruction interprets **Areg** as the address of the beginning of a vector of word-sized data objects, and **Breg** as an index into that vector. After execution, **Areg** holds the address of the indexed element, and **Creg** is popped into **Breg**, leaving **Areg** rotated into **Creg**. The operation performed by *wsub* is to multiply the integer in **Breg** by four and to add this to the address in **Areg** (without overflow checking).

4.5 Forming addresses

Access to a component of an array can be split into two sections; first the address of the component must be constructed, and then the transfer of data to or from that component must be performed.

Evaluating a subscript

Array subscripts can be evaluated efficiently using the *smul* or *mul* instruction. If array *A* has been declared by

```
int A[S1] ... [Sn];
```

where S_i ($i = 1..n$) are the dimensions, then one way of arranging this in memory is to have all elements of the array in a contiguous block. For the purposes of this section, suppose that the elements in the last dimension are stored adjacently; otherwise change the order of the dimension subscripts. For example Figure 4.1 shows the elements of a particular three dimensional array (*Array*) stored in this way.

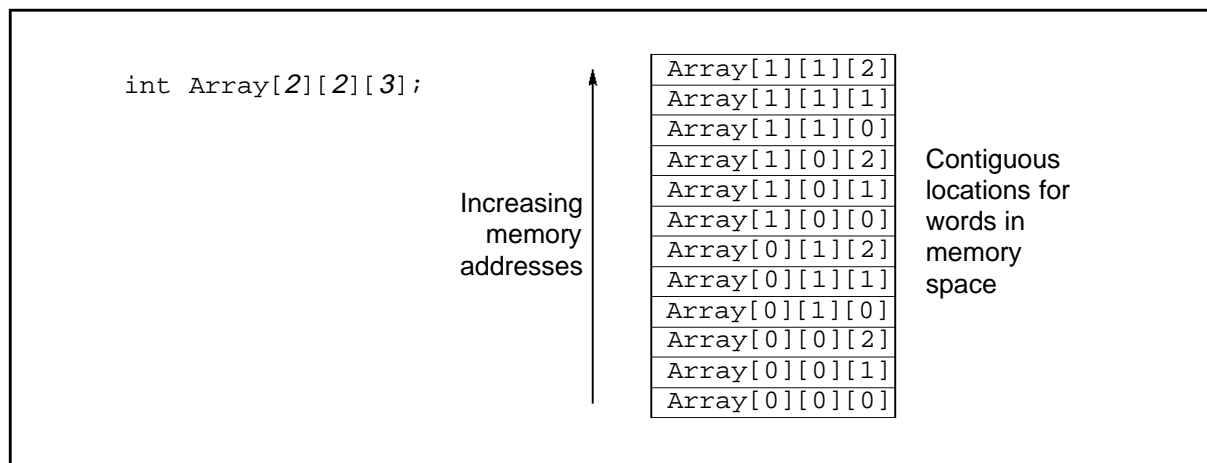


Figure 4.1 A possible method of storing an array of integers

If an access is required to the following array element

```
A[e1] ... [en]
```

then the code to evaluate the subscript is

```
e1;  
ldc S2; mul; e2; add;  
ldc S3; mul; e3; add;  
...  
ldc Sn; mul; en; add;
```

For example to evaluate the subscript for element `Array[x][y][z]`, (where `Array` is declared as in Figure 4.1) the code sequence is

```
ld x;  
ldc 2; mul; ld y; add;  
ldc 3; mul; ld z; add;
```

If x is 1, y is 0 and z is 2, then this evaluates to 8, which as can be seen from Figure 4.1, is the correct offset from the base of the array.

Accessing a word addressed array

Let *Wa_ptr* be a pointer to an array *Wa* that starts at a word boundary, and in which all component types are measured in words. Let *e* be a subscript expression. The address of component *e* of *Wa* is

e; Wa_ptr; wsub;

or if *e* is a constant expression this can be optimized to:

Wa_ptr; ldnlp e;

Accessing a byte addressed array

Similarly, let *Ba_ptr* be a pointer to an array (*Ba*) which may start at any byte location, and in which each component type is measured in bytes. Let *e* be a subscript expression.

The address of component *e* of *Ba* is:

e; Ba_ptr; add;

4.6 Comparisons and jumps

This section describes the arithmetical comparison instructions and their use in conditional program behavior. Unconditional jumps are also described. Functions and procedures are described in section 4.10, and evaluation of boolean expressions is described in section 4.7.

Comparisons, conditional behavior and jumps are provided by the instructions listed in Table 4.9.

Mnemonic	Name
<i>eqc n</i>	equal to constant
<i>gt</i>	greater than
<i>gtu</i>	greater than unsigned
<i>order</i>	order
<i>orderu</i>	order unsigned
<i>cj n</i>	conditional jump
<i>j n</i>	jump
<i>jab</i>	jump absolute

Table 4.9 Comparison and jump instructions

4.6.1 Representation of true and false

The ST20 uses 0 as *false* and 1 as *true*. These values are generated by predicate operations (for example comparisons). They can be loaded with single byte load constant instructions.

Implementation of languages with different representations of *true* and *false*

It is easy to implement programming languages that use a different representation of *true* and *false*. For example, using

eqc X; not; adc 1

in place of *eqc X* and

gt; not; adc 1

in place of *gt*, does not affect the representation of a *false* result, but changes the representation of *true* to -1, which is used in some programming languages.

4.6.2 Comparison

The primary instruction *eqc n* loads **Areg** with a truth value — *true* if **Areg** is initially equal to the instruction operand (*n*), *false* otherwise. **Breg** and **Creg** are unaffected.

gt and *gtu* take integer operands in **Areg** and **Breg** and produce a boolean result which is loaded into **Areg**. They also load the value in **Creg** into **Breg**, saving a copy of the initial **Areg** in **Creg**.

The *gt* instruction loads **Areg** with *true* if **Breg** > **Areg**, *false* otherwise, treating **Areg** and **Breg** as signed values. Similarly *gtu* loads **Areg** with *true* if the *unsigned* value of **Breg** is greater than the *unsigned* value of **Areg**; *false* otherwise.

4.6.3 Jump and conditional jump

There are two relative jump instructions; both are primary instructions.

The unconditional jump instruction, *j n*, adds its operand (*n*) to the address of the instruction immediately following it and puts the result into **Iptr**, thus transferring execution to another part of the program.

The conditional jump instruction, *cj n*, performs a jump if the value in **Areg** is 0 and does not affect the evaluation stack. If the value in **Areg** is not 0 *cj* rotates the value in **Areg** to the bottom of the evaluation stack and continues with the next instruction. Consequently *cj n* serves as ‘jump if *false*’ provided that the language being implemented interprets 0 as *false* (see section 4.6.1).

4.6.4 Conditional transfer of control

The conditional expressions used in a conditional branch of an *if* construct are compiled using the conditional jump. The statement:

```
if (E) {  
    P  
}
```

This compiles to:

```
    E; cj L;  
    P; j ENDIF;  
L:
```

where the label *ENDIF:* is at the end of the code for the *if* construct.

The compilation of a *while* loop is shown by the following example.

```
while (E) {
    P
}
```

This compiles to:

```
L:          E; cj ENDWHILE
           P; timeslice; j L
ENDWHILE:
```

Note that this loop includes a *timeslice* instruction. This causes the current process to be descheduled if a timeslice is due and timeslicing is enabled. The presence of this ensures that the process cannot occupy the CPU for too long provided timeslicing is enabled. It is good practice for multi-tasking programs to include a *timeslice* instruction in every loop. Timeslicing is described in section 7.4. Single task programs do not need to timeslice, but should have timeslicing disabled, so the *timeslice* instruction has no effect.

A *repeat .. until* loop is shown by the following example.

```
repeat {
    P
} until E
```

This compiles to:

```
          j K
L:        E; eqc 0; cj END
K:        P; timeslice; j L
END:
```

4.6.5 Ordering instructions

Two instructions are provided to select the smaller of two values. If **Breg** is smaller than **Areg** as a signed integer, *order* will swap **Areg** and **Breg**; otherwise *order* will have no effect. This can be used to find the minimum of two signed variables:

```
ldl a; ldl b; order;
stl minimum; stl maximum
```

Similarly *orderu* can be used to find the minimum or maximum of two unsigned values

4.7 Evaluation of boolean expressions

This section describes the operations using the logical *true* and *false* values, as used with the conditional jump *cj*. Conditional behavior and comparisons are described in section 4.6. Bitwise boolean operations are described in section 4.8. General issues concerning expression evaluation are discussed in section 4.3.

4.7 Evaluation of boolean expressions

The following shows the correspondence between C logical expressions and ST20-C1 instructions. X and Y represent expressions, and K represents a constant. The symbol ' \neg ' is a logical NOT (see section 4.7.1).

<i>true</i>	=	<i>ldc 1</i>
<i>false</i>	=	<i>ldc 0</i>
<i>! X</i>	=	$\neg(X)$
<i>X == Y</i>	=	<i>X; Y; sub; eqc 0</i>
<i>X != Y</i>	=	$\neg(X; Y; sub; eqc 0)$
<i>X == K</i>	=	<i>X; eqc K</i>
<i>X != K</i>	=	$\neg(X; eqc K)$
<i>X > Y</i>	=	<i>X; Y; gt</i>
<i>X < Y</i>	=	<i>Y; X; gt</i>
<i>X >= Y</i>	=	$\neg(Y; X; gt)$
<i>X <= Y</i>	=	$\neg(X; Y; gt)$

Further optimizations can be made to the 'not equals' comparison when followed by a conditional jump.

<i>X != Y; cj L</i>	=	<i>X; Y; sub; cj L</i>
<i>X != 0; cj L</i>	=	<i>X; cj L</i>

4.7.1 Evaluation of NOT

If zero represents false and 1 represents true, then logical NOT can be performed by *eqc 0*.

4.7.2 Evaluation of AND and OR

For evaluation of logical AND and OR operations, the instruction sequence depends on whether strict or non-strict evaluation is used, i.e. whether both operands are always evaluated. This is important if side-effects may occur, such as a trap, or if the second operand is not always defined, as in:

```
if ((ptr != NULL) && (ptr->tag == TAG_VAL)) ...
```

In this example, *ptr->tag* is not defined if *ptr* is *NULL*. For languages such as ANSI C, non-strict evaluation is required, so the following short-cuts must be used:

<i>X OR Y</i>	=	$\neg(\neg(X); cj L; \neg(Y); L:)$
<i>X AND Y</i>	=	<i>X; cj L; Y; L:</i>

For non-strict evaluation, the following laws should be applied to the compilation of conditional expressions before code is generated to ensure that the jump is taken as early as possible:

$\neg(X \text{ AND } Y)$	=	$(\neg X) \text{ OR } (\neg Y)$	[= $\neg(X; cj L; Y; L:)$]
$\neg(X \text{ OR } Y)$	=	$(\neg X) \text{ AND } (\neg Y)$	[= $\neg(X); cj L; \neg(Y); L:]$]
$(X \text{ OR } Y); cj L$	=	$(\neg X); cj M; Y; cj L; M:$	
$(X \text{ AND } Y); cj L$	=	$X; cj L; Y; cj L$	

In other languages, evaluation of boolean expressions may be strict (for example, ADA gives the programmer the choice) and so both expressions in dyadic logical operations may need to be evaluated.

Where *false* is represented by 0, and *true* is represented by any *fixed* bit pattern other than 0 (e.g. *true* is always 1, or *true* is always -1), then the following transformations apply:

$$\begin{array}{lcl} X \text{ OR } Y & = & X \text{ BITOR } Y \\ X \text{ AND } Y & = & X \text{ BITAND } Y \end{array}$$

and the bitwise instructions given in section 4.8 can be used:

Note that even for some non-strict evaluations, the above sequence may be preferable. Where *Y* is a simple boolean expression such as a local variable, its evaluation does not cause any side-effects, and so it does no harm to implement a non-strict evaluation using a bitwise operation.

4.8 Bitwise logic and bit operations

Bitwise logic and bit operations are provided by the instructions listed in Table 4.10.

Mnemonic	Name
<i>and</i>	and
<i>or</i>	or
<i>xor</i>	exclusive or
<i>not</i>	bitwise not
<i>bitld</i>	bit load
<i>bitst</i>	bit store
<i>bitmask</i>	bit mask
<i>rmw</i>	memory read modify write

Table 4.10 Bitwise logic and bit instructions

The *not* operation has only one operand that is taken from **Areg**. The result of this, which is a bitwise inversion of all bits in the operand, is loaded into **Areg**, leaving **Breg** and **Creg** unaffected.

and, *or* and *xor* are bitwise logical operations on two operands that are taken from **Areg** and **Breg**. For each, the result is loaded into **Areg**. The data previously held in **Creg** is popped into **Breg** and the initial **Areg** is left in **Creg**. These operations are commutative.

bitld, *bitst* and *bitmask* are used for setting, clearing and testing bits of a word. *bitld* returns the value of a single bit from a value in **Breg**, *bitst* sets or clears a single bit and *bitmask* creates a mask with a single bit set. In each case the bit number is initially in **Areg** and the result is put in **Areg**. For *bitld* and *bitst*, the value containing the bit to be tested, set or cleared is initially in **Breg**.

4.9 Shifting and byte swapping

4.8.1 Memory bit test and clear or set

Bits of a word in memory may be tested and set or cleared by the instruction *rmw*. The address of the memory word is held in **Areg** and a bit masks in **Breg** and **Creg**. *rmw* clears the bits of the memory word that are set in **Creg**, and then sets the bits of the memory word that are set in **Breg**. The initial memory word is loaded into **Areg**, with **Areg** pushed down to **Breg** and **Breg** pushed down to **Creg**.

4.9 Shifting and byte swapping

The shift and byte swapping operations are provided by the instructions listed in Table 4.11.

Mnemonic	Name
<i>shl</i>	shift left
<i>shr</i>	shift right
<i>ashr</i>	arithmetic shift right
<i>swap32</i>	byte swap 32

Table 4.11 Shifting and byte swapping instructions

The shift operations (*shl*, *shr* and *ashr*) shift the operand in **Breg** by the number of bits specified by the unsigned integer in **Areg** and put the result in **Areg**. *shl* and *shr* fill the vacated bit positions with zero bits, while *ashr* fills the vacated bits with copies of bit 31, which is the original sign bit. If **Areg** is zero, the result is the initial value of **Breg**. When the value in **Areg** is greater than the number of bits in the object being shifted, the result of the operation is undefined. The data previously held in **Creg** is popped into **Breg**, and the initial **Breg** is left in the **Creg**.

swap32 reverses the order of the bytes in **Areg** by swapping byte 0 with byte 3 and swapping byte 1 with byte 2.

4.10 Function and procedure calls

The function and procedure call operations are provided by the instructions listed in Table 4.12.

Mnemonic	Name
<i>fcall</i>	function call
<i>jab</i>	jump absolute
<i>ajw</i>	adjust work space
<i>gajw</i>	general adjust workspace

Table 4.12 Function and procedure instructions

The primary instruction *fcall n* calls a function or procedure. It stores the instruction pointer (which holds the return address) in the word pointed to by the **Wptr**. The operand to the call - *n* - is added to the address of the next instruction to produce the address of the first instruction of the procedure or function being called. Since the call address is relative, the code is relocatable.

A function called using *fcall* must have a fixed offset at compile time. *jab* is used for calling functions and procedures at dynamically calculated addresses, for example when using function pointers.

The *jab* instruction is also used to perform the return. The return address must have been restored with a *ldl* from the stack into the **Areg**. A procedure or function that requires local work space will normally include *ajw* instructions to allocate and deallocate space.

When the *jab* instruction is executed, the programmer must ensure that:

- the **Areg** holds the return address;
- any workspace claimed by the procedure should have been released so that the **Wptr** has returned to the value it held at the start of the procedure.

The *jab* instruction uses one word of the evaluation stack, so the other two words can therefore be used to return up to two values to the calling code, including a pointer to a block of additional data to be returned.

4.10.1 Adjusting work space

The primary instruction *ajw* is used to perform a relative adjustment to the stack pointer **Wptr** to:

- create work space on the stack at the beginning of the function and
- return the work space pointer at the end of the function.

ajw n increases the value of the workspace pointer by the number of words in its operand value, *n*. Work space is created at the beginning of a function or procedure with a negative operand and released before returning with a positive operand.

The amount of extra work space needed will normally include:

- space to save any parameters passed in the evaluation stack;
- space for local variables and temporaries;
- space for any hidden system variables such as the static chain.

For example, a function with *w* words of local work space might be:

```
T myfunction (param_1, param_2)
{
    local variable declarations;
    P;
    return (E);
}
```

This can be compiled as:

```
ajw -w;
stl param_1; stl param_2;
P;
E;
ajw w;
```

4.10 Function and procedure calls

```
ldl 0;  
jab;
```

ST20-C1 processors may have a workspace cache which holds a copy of a few words at the bottom of the work space. This cache is transparent to the programmer but may substantially improve performance. It is refilled whenever the **Wptr** is adjusted, so the *ajw* instruction should not be used excessively.

4.10.2 Parameters

It is convenient to load the first three parameters of the procedure or function into the evaluation stack registers, and to arrange the work space of the calling code so that the additional parameters can be stored in locations 1, 2, . . . of the work space before the procedure is called. Location zero of the work space is used for the return address. This is illustrated in Figure 4.2, which shows a possible work space layout for a function or procedure with six parameters and four local variables.

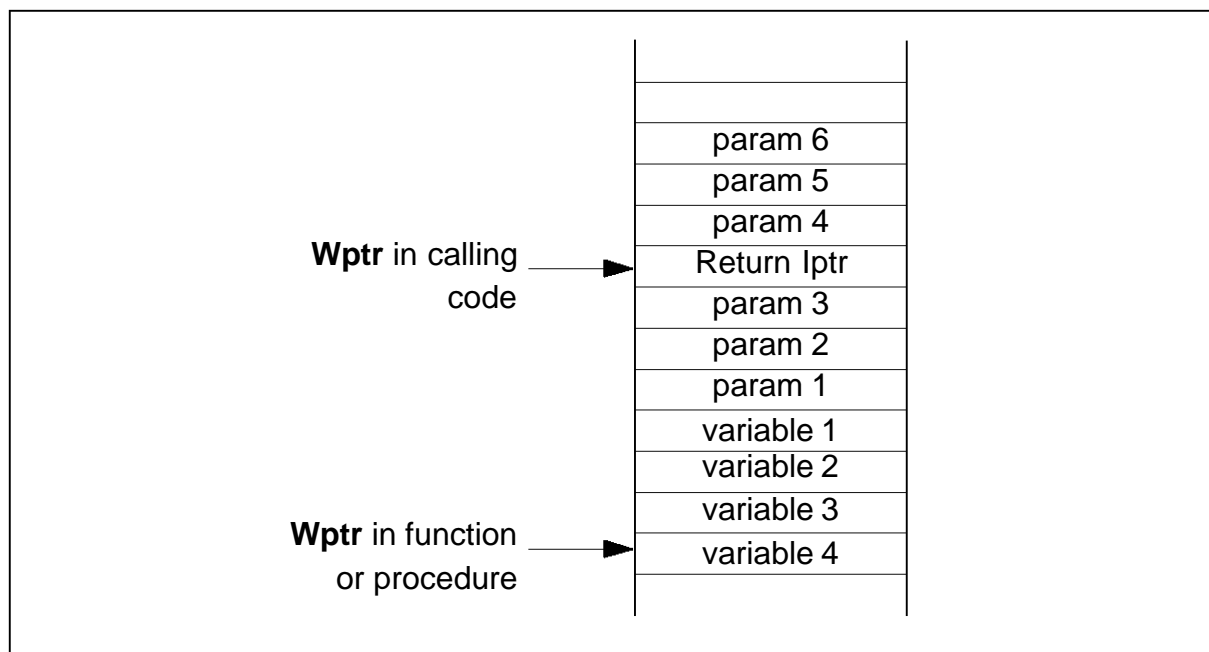


Figure 4.2 Example function or procedure workspace

To enable the procedure to access non-local variables the parameters of a procedure may include a link to the environment in which the procedure was declared.

4.10.3 Returning results

Up to two results of size less than or equal to the word length of the processor can be returned from a function in the evaluation stack — the *jab* instruction uses the third register. Further results, or results larger than the word length, can be returned by passing into the function the addresses of locations to store these results as extra parameters.

A C function is used for purposes of illustration. For simplicity, it is assumed that the single result can be returned in the evaluation stack:

```
ajw -local_variables-1;
P;
E;
ldl local_variables;
ajw local_variables+1;
jab;
```

One of the loading sequences described earlier may be required if the expressions returned in the registers contain evaluations.

4.10.4 Calling a function

The first three parameters should be loaded into the evaluation stack before the *fcall* instruction. These parameters can be stored as local variables after the workspace pointer has been moved down, to make the best use of the work space cache. The remainder of the parameters passed should be loaded into the work space before *fcall* is executed.

When the function returns, the results whose addresses were passed will already have been stored so all that remains is to store up to 2 results returned in the evaluation stack.

For example the function call

$$V = F(E_1, \dots, E_n)$$

could be compiled by

```
E3; stl 0; . . . ; E_n; stl (n-3);
E2; E1; static_link; fcall F;
stl V;
```

The compiler must have already allocated sufficient workspace for the parameters that are stacked explicitly.

Single result functions

In most programming languages, a function that returns a single result can be used in an expression as well as in an assignment.

A common form of function returns a single value contained in a word — the mechanism described above will return this in **Areg**. When compiling expressions, (using the algorithm described in section C.3 in Appendix C) the depth of such a function call should be taken as being infinite — i.e. deeper than any other form of expression. This is because the function call will always lose any other information in the registers. By giving it infinite depth the expression compilation algorithm will never call a function while another expression result is being held in a register.

4.10.5 Other work space allocation techniques

The *gajw* instruction exchanges the contents of **Wptr** and **Areg**, allowing work spaces to be allocated dynamically, and allowing dynamic switching between existing work spaces. If a process work space holds a pointer to a new work space, then the following code changes to the new work space and stores a pointer to the old work space.

```
ldl Wnew; gajw; stl Wold;
```

The old work space can be restored by

```
ldl Wold; gajw;
```

In addition, the old work space can be accessed from the new work space, using

```
ldl Wold; ldnl x;  
ldl Wold; stnl x;  
ldl Wold; ldnlp x;
```

4.11 Peripherals and I/O

The peripheral and I/O instructions are listed in Table 4.13.

Mnemonic	Name
<i>io</i>	input / output
<i>bitmask</i>	bit mask
<i>ldtdesc</i>	load task descriptor
<i>stop</i>	stop process

Table 4.13 Peripheral and I/O instructions

4.11.1 Using the IO register

The IO register is a 32-bit register used for simple bit control of devices outside the core. The bits of the register are directly mapped to external connections on the ST20-C1 core. The connections to and from the IO register may be to on-chip or external peripherals depending on the particular chip design. The bits of the IO register are defined in Table 3.3. Some bits at the most significant end of each half word may be reserved for the system in some ST20 variants; see the data sheet for the variant.

Setting an output bit will cause the corresponding connection to be driven high, and clearing the bit will drive the connection low. Similarly any input or output bit may be tested for the state of the connection; if the connection is high the bit will be set and if the connection is low the bit will be clear.

The instruction *io* sets and clears bits of the IO register and loads a copy of the initial IO register. A bit in the bottom half-word may be set in the IO register by:

```
ldc 0; ldc bit_number; bitmask; io;
```

A bit in the bottom half-word may be cleared in the IO register by:

```
ldc bit_number; bitmask; ldc 0; io;
```

Any bit may be read from the IO register by:

```
ldc 0; dup; io;  
ldc bit_number; bitld;
```

The IO register is global and is not changed or saved by a context switch. If more than one process accesses the IO register then it may need to be protected by a semaphore. On reset the IO register is set to all zeros.

4.11.2 Memory-mapped peripherals

On-chip peripherals may have memory-mapped registers in the address space. Access to these registers is performed in the same way as accessing memory. If a peripheral has a block of word-aligned registers with base address *peripheral* then a register with word offset *register* may be read by:

```
ld peripheral;  
ldnl register;
```

and *value* may be written to the register by:

```
ld value;  
ld peripheral;  
stnl register;
```

4.11.3 Channel-type peripherals

Some peripherals, for example peripherals using DMA (direct memory access), may use a channel-type control model. This section describes how to use such peripherals, which use a micro-interrupt to notify the CPU that an assigned job is completed.

This type of peripheral works best with a multi-tasking program, so that the CPU has other processes to execute while the peripheral is busy. However, if multi-tasking is not otherwise required, then an interrupt model can be used. Multi-tasking is described in Chapter 7 and interrupts and the exception vector table are described in Chapter 6.

Multi-tasking

The principle of using the channel model with multi-tasking is that the CPU tells the peripheral to start a job and then deschedules the current process. The job might be peripheral input/output or DMA transfer. This allows the CPU to continue executing other processes while the job is in progress. When the peripheral completes the job it signals to the CPU, which reschedules the process.

To enable this to happen, the task descriptor of a user process can be entered into the exception vector table. This entry is called the peripheral channel. The peripheral signals a micro-interrupt, which interrupts the CPU with the exception level associated with the user process. The CPU recognizes that the exception vector table entry is a user process because bit zero is *UserProcessType*, and either adds the process to the end of the scheduling queue or takes a schedule exception trap if installed. The scheduling exception trap allows a scheduling kernel to control the rescheduling of the process.

In more detail, the steps to perform a job using this model are:

- 1 The CPU saves the task descriptor of the current process in the exception vector table at the exception level for the peripheral.
- 2 The CPU tells the peripheral the number of bytes to be read or written, the address of the start of the data or input buffer.
- 3 The CPU signals to the peripheral that the job can start.
- 4 The CPU deschedules the process, using the *stop* instruction, to wait for the peripheral to complete the job.
- 5 The CPU executes other processes while the peripheral performs the job.
- 6 The peripheral completes the job and sends a micro-interrupt to the CPU, with the exception level.
- 7 The CPU reads the exception vector table and recognises the entry as a user process. The process is added to the back of the scheduling queue, or if the schedule execution trap is enabled then the trap is taken.

The code to execute steps 3 to 4 must not be interrupted, since otherwise the peripheral job may be completed before the process is descheduled by a *stop* instruction, which may crash the processor. Interrupts may be temporarily disabled by clearing the *local_interrupt_enable* bit of the status register, which is set automatically when the process is descheduled. Steps 5 to 7 happen automatically and need no coding.

The code to drive the peripheral will depend on the peripheral and the interface to it. Typically the parameters (e.g. the byte count and the buffer address) would be written to memory mapped registers and then a further write would be needed to start the peripheral job.

The code to perform a DMA to transmit *param_count* bytes to or from *param_buffer* using exception level *except_level* where the DMA registers *periph_count*, *periph_buffer* and *periph_start* are in a block at *peripheral* would be similar to the following:

```
ldtdesc; ld ExceptionBase; stnl except_level;

ld param_count; ld peripheral; stnl periph_count;
rev; ld param_buffer; stnl periph_buffer;

ldc local_interrupt_enable; bitmask; statusclr;
rot; ldc start_value; arot; stnl periph_start;
stop;
```

The process will resume at the next instruction after the *stop* when the peripheral job is complete.

Single tasking

For programs which are not multi-tasking, it is not desirable to deschedule the program while the peripheral is busy. The main program should continue with other jobs while the peripheral is busy. An interrupt handler can be written to signal to the main program that the peripheral job is complete. The descriptor of the interrupt handler exception control block is placed in the exception vector table. The descriptor

is the address of the control block ORed with the type flag *ExceptionProcessType* in bit 0.

The code to perform a DMA to transmit *param_count* bytes to or from *param_buffer* using exception level *except_level* where the DMA registers *periph_count*, *periph_buffer* and *periph_start* are in a block at *peripheral* and the interrupt handler is at *except_control_block* would be similar to the following:

```
ld except_control_block; ldc ExceptionProcessType; or;
ld ExceptionBase; stnl except_level;
```

```
ld param_count; ld peripheral; stnl periph_count;
ld param_buffer; arot; stnl periph_buffer;
```

```
ldc start_value; arot; stnl periph_start;
```

When the interrupt handler starts execution the peripheral job will be complete.

4.12 Status register

The status register may be manipulated using the instructions listed in Table 4.14.

Mnemonic	Name
<i>statusclr</i>	clear bits in status register
<i>statusset</i>	set bits in status register
<i>statusst</i>	test status register

Table 4.14 Semaphore instructions

In each of these instructions, **Areg** holds a bit mask. *statusclr* copies the initial status register into the **Areg** and clears the bits of the status register that are set in the initial **Areg**. For example, to clear the bit *bit_number*:

```
ldc bit_number; bitmask; statusclr
```

statusset is similar, but sets the bits of the status register that are set in the initial **Areg**. *statusst* returns in the **Areg** the status bits masked by the initial **Areg**. The **Breg** and **Creg** are unaffected.

The status register is described in section 3.3.2.

5 Multiply accumulate

This section describes the multiply-accumulate instructions and their use. All these instructions are described in the context of their intended use. Instructions for general use (arithmetic, loading, storing etc.) are described in Chapter 4. Instructions for exceptions are described in Chapter 6 and multi-tasking instructions are described in Chapter 7. The architecture of the ST20-C1, including the registers and memory arrangement, is described in Chapter 3.

Multiply accumulate operations are provided by the signal processing instructions listed in Table 5.1.

Mnemonic	Name
<i>mac</i>	multiply accumulate
<i>umac</i>	unsigned multiply accumulate
<i>smacinit</i>	initialize short multiply accumulate loop
<i>smacloop</i>	short multiply accumulate loop
<i>biquad</i>	biquad IIR filter step

Table 5.1 Multiply accumulate instructions

5.1 Data formats

A signed fractional number of N bits is described as $x.y$, where $x+y=N$. This means the number is made up from x bits before the binary point, an implied binary point, and y fractional bits. More details of the data formats are given in section 5.7.

5.2 *mac* and *umac*

mac and *umac* are general purpose multiply accumulate instructions, multiplying two 32-bit values and adding them to a 32-bit unsigned initial accumulator, giving a 64-bit accumulator. *mac* treats the multiplicands as signed and *umac* treats them as unsigned. Initially **Areg** and **Breg** hold the values to be multiplied and **Creg** holds the initial accumulator. On completion, **Areg** is the least significant word of the result accumulator, **Breg** the most significant and **Creg** holds a copy of the initial **Areg**.

5.3 Short multiply accumulate loop

The *smacloop* instruction performs a multiply-accumulate operation on two vectors of 16-bit values held in memory. It takes an initial accumulator value and two pointers, one to each of two data vectors.

- The X vector of data values is normally considered to reside within a circular buffer of programmable size, but this can be turned off. When data fetches reach the end of this buffer, the pointer wraps-around back to the start of the buffer and continues. The X vector must be word aligned.
- The Y vector of coefficients is always in a flat address space, and never wraps around. The Y vector must be half-word aligned.

The data items from each vector are read from memory in turn and the products formed between corresponding pairs from the two vectors. Each of these products is added into the running accumulator value. The instruction completes with 3 values in the stack - the final accumulator value and the two updated data pointers.

Four control values are held in the status register, as shown in Table 5.2.

Field	Size	Meaning
mac_count	8 bits	The number of steps (from 1 to 256 items).
mac_buffer	3 bits	The size code for the data buffer within which the data vector lies.
mac_scale	2 bits	Shift control for scaling coefficient values.
mac_mode	1 bit	Accumulator format - 0 indicates 16-bit (short) and 1 indicates 32-bit (long) value.

Table 5.2 *smacloop* status register fields

These values are initialized by the *smacinit* instruction. The *smacinit* instruction takes a packed control word in **Areg**, extracts the control fields and loads these into the status register. For *smacinit*, **Areg** is organized as shown in Table 5.3.

Field	Size	Least significant bit	Most significant bit
mac_count	8 bits	0	7
mac_buffer	3 bits	8	10
mac_scale	2 bits	11	12
mac_mode	1 bit	13	13

Table 5.3 *smacinit* **Areg** format

These status register values are global and are not saved when a process is timesliced or descheduled. If more than one process is performing short multiply accumulate loops then the values should be reloaded by the process code using *smacinit* after each *timeslice* and *stop* instruction.

5.3.1 X buffer size

The X vector buffer size is determined by the **mac_buffer** control field, which may take the values 0 to 7. When **mac_buffer** is 0, then no address wrapping takes place, i.e. the buffer is assumed to be of infinite size. Otherwise, the buffer size is $2^{\text{mac_buffer}+2}$, as shown in Table 5.3. The X buffer must be aligned to a multiple of its own size, so a buffer of N bytes must start at an address whose value is a multiple of N bytes.

5.3.2 Number of steps

The **mac_count** control field in the status register determines the number of multiply-accumulate steps for *smacloop*. This is an unsigned 8-bit integer. The value zero signifies 256 steps; otherwise the value of **mac_count** is the number of steps.

5.4 Biquad IIR filter

mac_buffer	Buffer size (data items)	Buffer size (bytes)
0	infinite	infinite
1	4	8
2	8	16
3	16	32
4	32	64
5	64	128
6	128	256
7	256	512

Table 5.4 **mac_buffer** coding

5.3.3 Scaling

Scaling of the input data in the X vector is controlled by the **mac_scale** field of the status register, as described in section 5.6.

5.3.4 Accumulator format mode

smacloop supports two data formats for the initial and final accumulator value. If **mac_mode** is 0 then Q15 (sign extended to 32 bits) is used, and the mode is said to be *ShortMode*. If **mac_mode** is 1 then Q31 is used, and the mode is said to be *LongMode*.

5.4 Biquad IIR filter

The *biquad* instruction performs a fixed sequence of 5 multiply-accumulates. Figure 5.1 shows an example using Q14 format. The parameters to the instruction are pointers to three vectors of 16-bit values:

- an X input data vector,
- a Y results vector and
- a coefficient vector C. This vector must be word-aligned.

biquad calculates the next item in the Y vector according to the following formula, and writes this to memory, incrementing the X and Y pointers by two bytes:

$$Y[2] = X[0].C[0] + X[1].C[1] + X[2].C[2] + Y[0].C[3] + Y[1].C[4]$$

The X and Y vectors must be either both word-aligned or neither word-aligned.

The C pointer is left unchanged. This allows successive *biquad* instructions to be executed back-to-back to generate a set of filter outputs with no additional overhead.

biquad scales the X input data according to the **mac_scale** field of the status register, as described in section 5.6. The **mac_scale** field may be set using *smacinit*, as described in section 5.3.

5.5 Data vectors

Both *biquad* and *smacloop* operate on arrays of 16-bit values, packed two per word. This allows the ST20-C1 to read two values per cycle from memory which is fundamental to the high performance of the multiply-accumulate instructions. In all cases, data values must be half-word aligned.

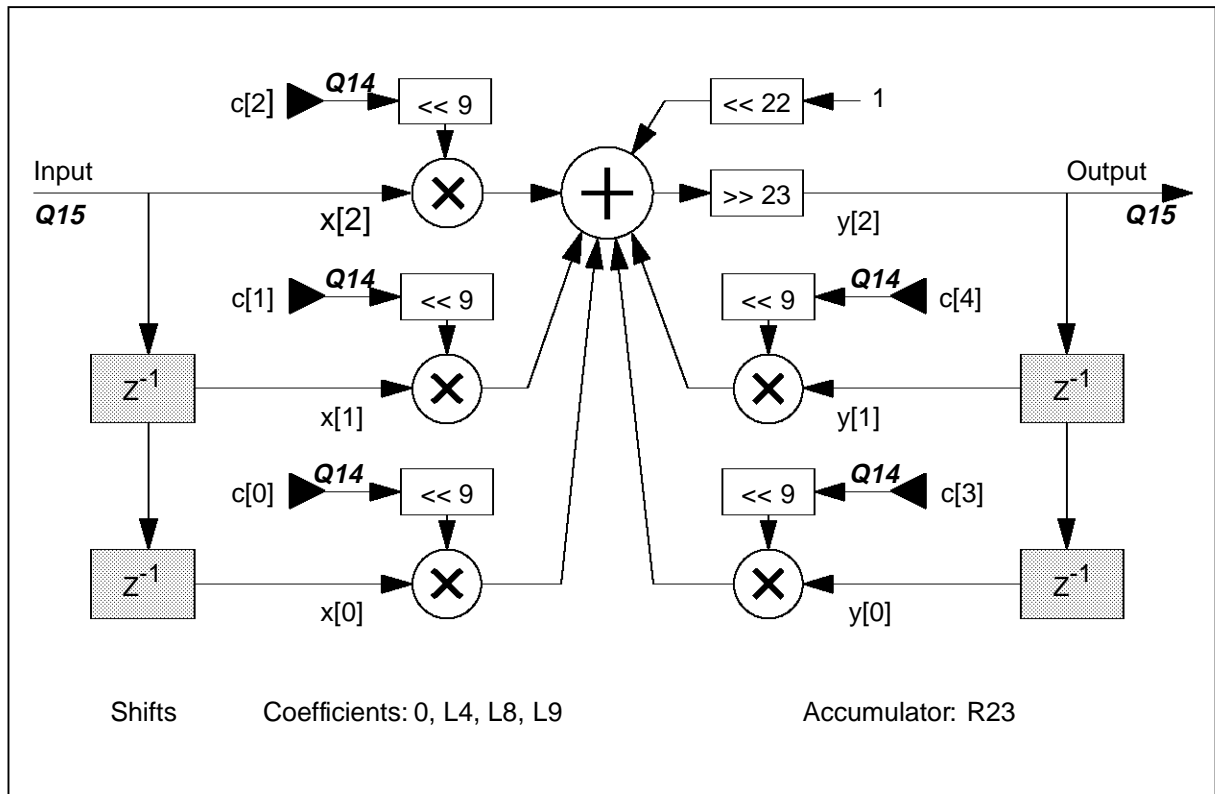


Figure 5.1 ST20-C1 *biquad* instruction example: $Q15 = Q15 \times Q14$

5.6 Scaling

The *biquad* and *smacloop* operations are performed with an oversize accumulator of 48 bits. The accumulator value is always sign-extended to the full width of the accumulator.

During a multiply-accumulate sequence the value in the accumulator may temporarily go outside the representable range of the final result, but can never overflow the accumulator for a single *biquad* or *smacloop*.

5.6.1 Accumulator scaling

The user-visible accumulator is either in *LongMode* (Q31) or *ShortMode* (Q15). For *smacloop*, the mode is defined by the **mac_mode** status register field. *biquad* only supports *ShortMode*.

Pre-scaling converts the user-visible accumulator to an internal format accumulator, as shown in Figure 5.2. The inverse operation is post-scaling which is converting an

5.6 Scaling

internal format accumulator to a user-visible accumulator.

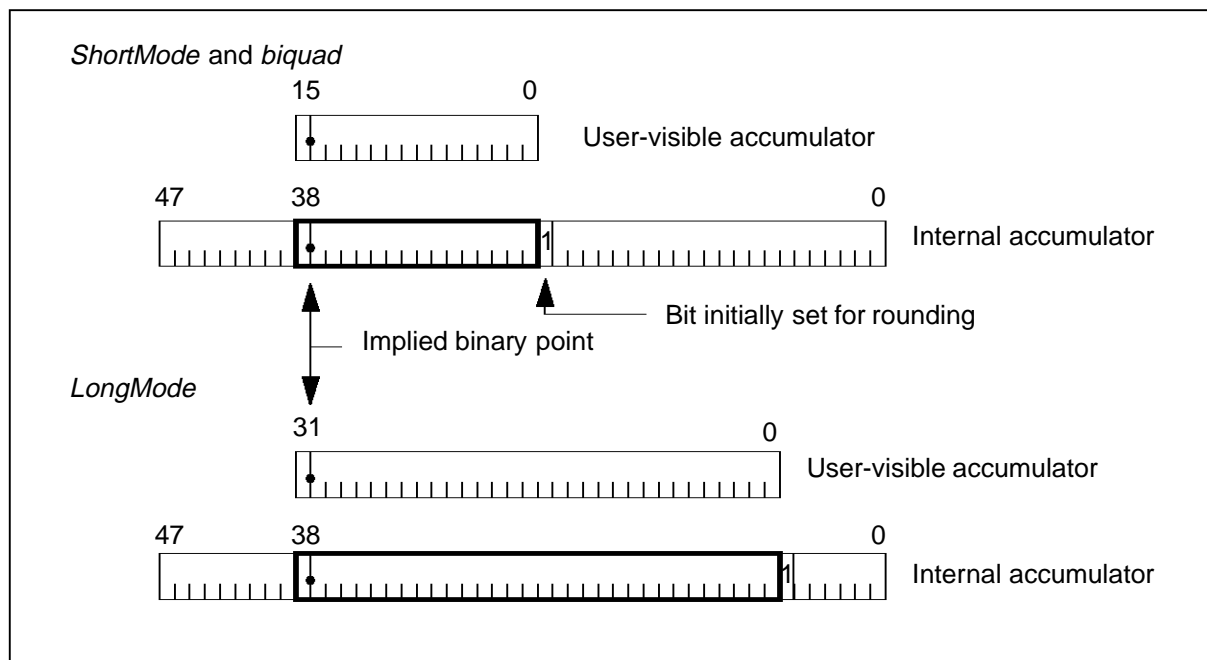


Figure 5.2 Accumulator scaling

The accumulator is left-shifted 8 bits so that the assumed binary point is moved from below bit 30 to below bit 38. The accumulator value is saturated from bit 38 upwards. At the end of the multiply-accumulate sequence the accumulator value is shifted down (right) by an extra 8 bits to compensate for the left shift of 8 on coefficient inputs.

5.6.2 Coefficient scaling

The standard data format for data and coefficients is 1.15 (Q15). The product of two 1.15 numbers is 2.30. The coefficient value for each multiply-accumulate operation is pre-scaled before being fed into the multiplier. The shift distances are controlled by the **mac_scale** field: of the status register, as shown in Table 5.5.

mac_scale	coefficient shift
0	0
1	left 4
2	left 8
3	left 9

Table 5.5 **mac_scale** values

The standard behavior for 1.15 (Q15) values is to shift the coefficients by 8 places, using **mac_scale** set to 2, which exactly compensates the extra right shift of 8 on the

final accumulator. This is shown in Figure 5.3.

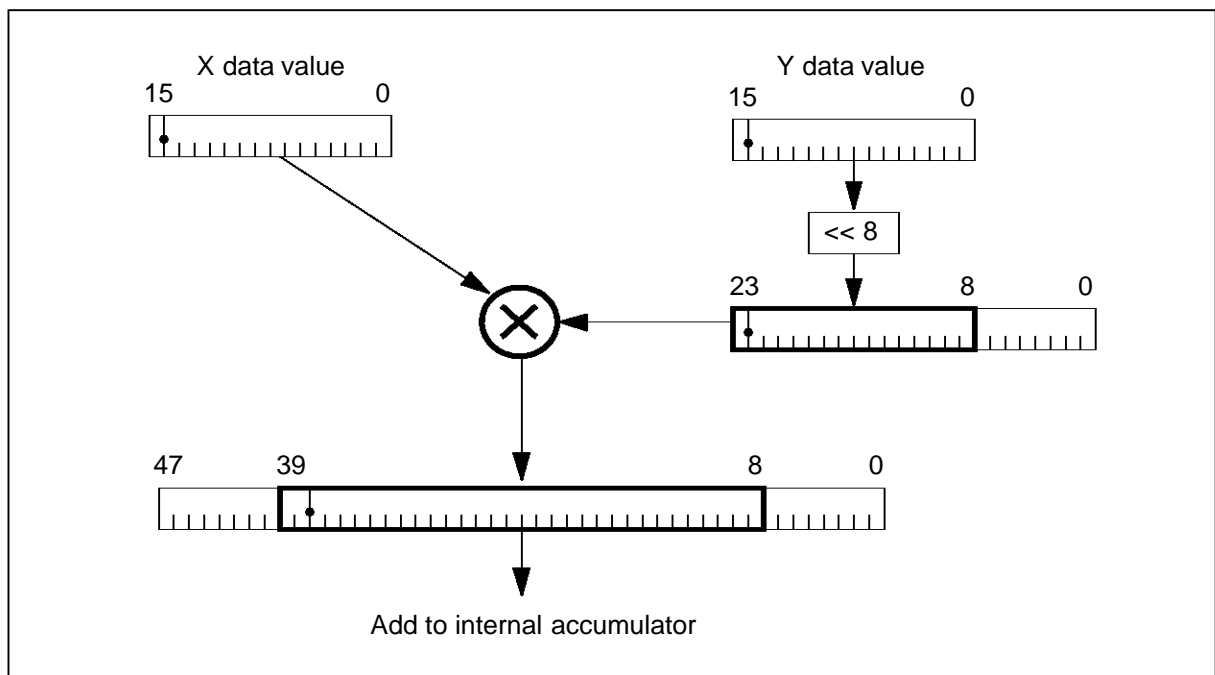


Figure 5.3 Coefficient scaling with **mac_mode** set to 2

Shifting the coefficient by 1 extra place (i.e. 9 places) is used to normalize a 2.14 (Q14) coefficient to the correct position for the binary point. This is shown in Figure 5.4.

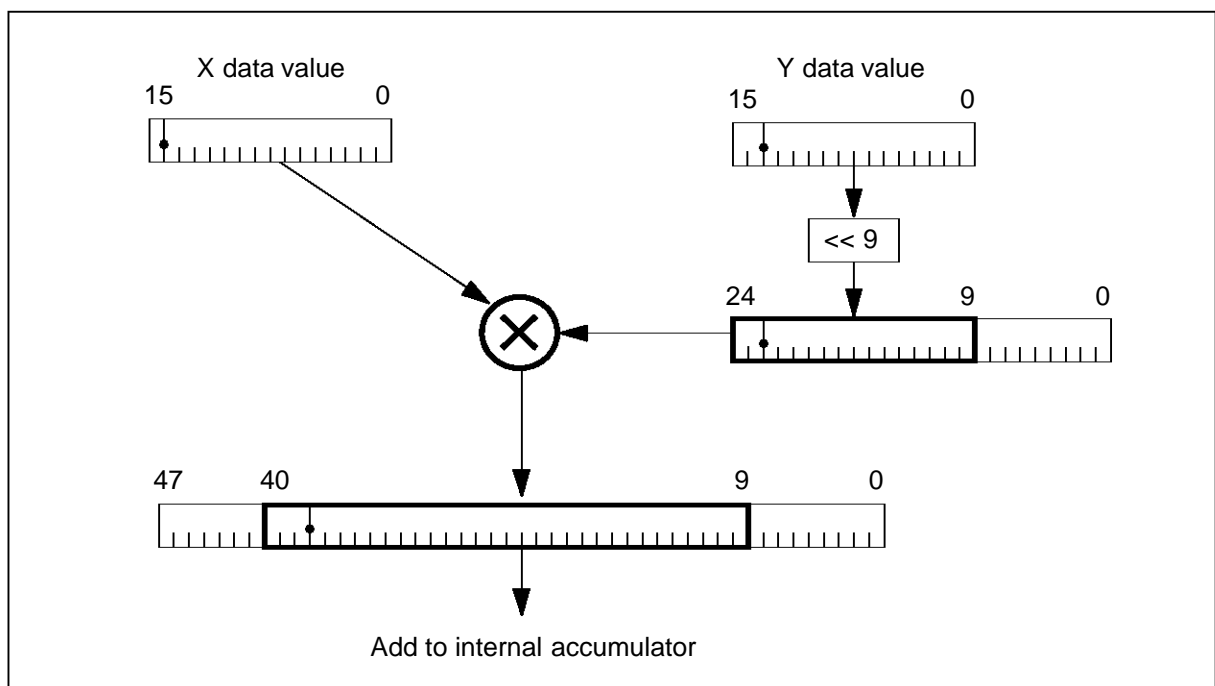


Figure 5.4 Coefficient scaling with **mac_mode** set to 3

5.6 Scaling

Under shifting (by less than 8) is used for magnitude reduction (most suitable for *smacloop*) by 4 bits (x16) using **mac_scale=1** or 8 bits (x256) using **mac_scale=0**. Under-shifting by 4 bits is shown in Figure 5.5, and by 8 bits in Figure 5.6.

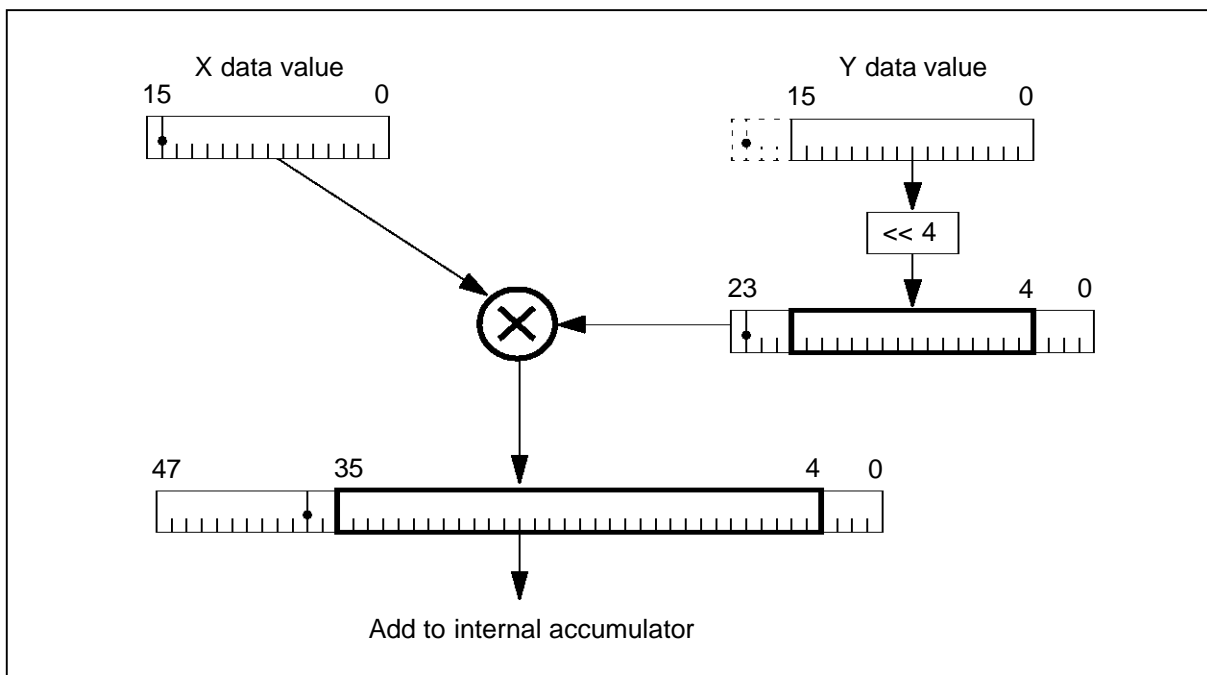


Figure 5.5 Coefficient scaling with **mac_mode** set to 1

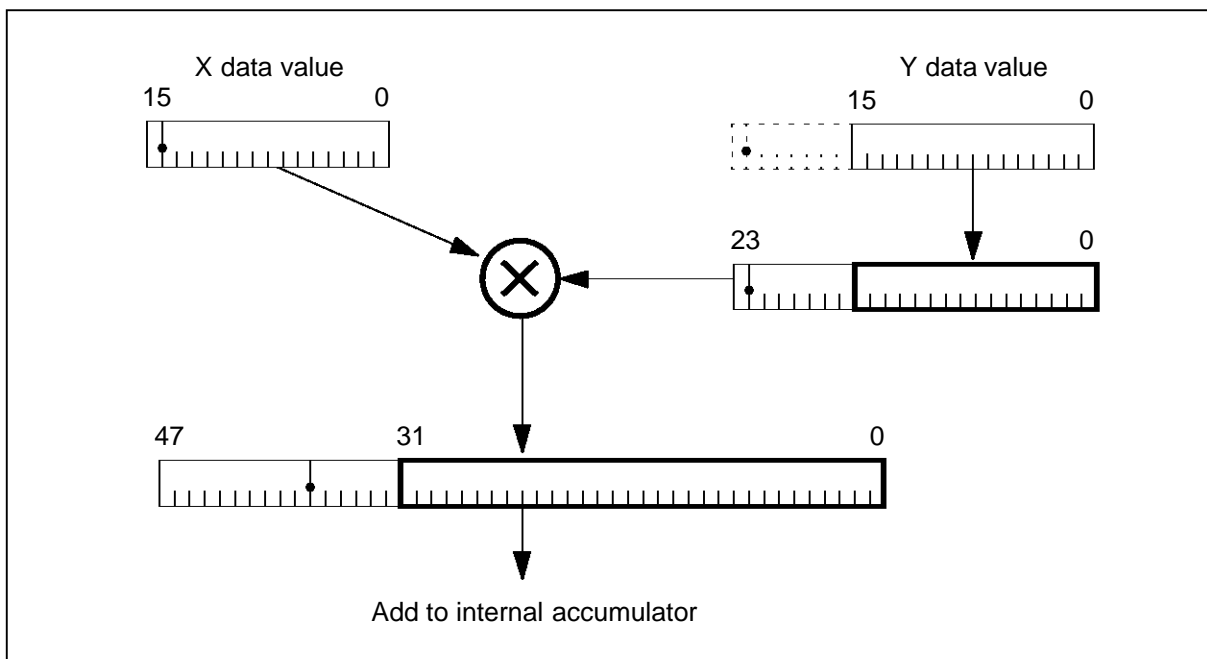


Figure 5.6 Coefficient scaling with **mac_mode** set to 0

5.6.3 Pre-scaling and rounding - *smacloop*

The initial accumulator value must be loaded into the accumulator for the start of the *smacloop* instruction. The initial accumulator is either in Q31 format (*LongMode*), or Q15 format (*ShortMode*) and is handled accordingly.

In *LongMode*,

- left shift by 7 places (right 1 + left 8)
- sign extend from bit 38 to the most significant bit
- set bit 6 = 1 (for rounding)

In *ShortMode*,

- left shift by 23 places (left 15 + left 8)
- sign extend from bit 38 to the most significant bit
- set bit 22 = 1 (for rounding)

Note that rounding is achieved by adding half of the least significant bit to the initial value, which by associativity is equivalent to adding it to the final value.

5.6.4 Pre-scaling and rounding - *biquad*

The biquad instruction starts with an empty accumulator. Since the result is always Q15 (equivalent to *ShortMode*), rounding is achieved by loading the accumulator with 2^{22} (which is half of the least significant bit of the result).

5.6.5 Post-scaling and saturation - *smacloop*

At the end of a *smacloop* instruction the final accumulator value is saturated and scaled to the appropriate format, according to **mac_mode**.

If either the *overflow* or *underflow* bits in the status register are set, then the final value is set to the appropriate exceptional value from Table 5.6. Note that this does not involve testing the accumulator value, which is considered to be invalid if either of these status register bits are set.

Otherwise, when the status register reports no overflow or underflow, bits 38 to 47 inclusive of the accumulator are tested for mutual equality. If they are all the same, the accumulator value is well-formed and post-scaling is applied to produce the final accumulator value.

In *LongMode*,

- arithmetic right shift by 7 places (left 1 + right 8)
- truncate to low 32 bits

In *ShortMode*,

- arithmetic right shift by 23 places (right 15 + right 8)
- truncate to low 32 bits

5.6 Scaling

However if the bits are not all equal, then an error has occurred. If the most significant bit (bit 47) is zero then the overall result is positive, so the error is overflow. If the most significant bit is 1 then the overall result is negative, so the error is underflow. The appropriate status register bit (*overflow* or *underflow*) is set accordingly, and the final value is taken from Table 5.6.

5.6.6 Post-scaling and saturation - *biquad*

The result of a *biquad* is always short (16-bits). The saturation test is from bit 38 upwards, and the *overflow/underflow* flags are set as required. Note the initial value of the *overflow/underflow* flags is **not** taken into account. If a saturation error occurs, the appropriate *ShortMode* value from Table 5.6 is used. If no error occurs, the scaling is:

- arithmetic right shift by 23 places
- truncate to low 32 bits

5.6.7 Error: load exceptional value

There are four exceptional values based on whether the result is to be delivered as a *ShortMode* (Q15) or *LongMode* (Q31) value, and whether the error was overflow or underflow:

mac_mode	Overflow	Underflow
<i>ShortMode</i>	00007FFF	FFFF8000
<i>LongMode</i>	7FFFFFFF	80000000

Table 5.6 Exceptional values

Note that in all cases the final value is placed in a 32-bit register (**Areg**).

5.6.8 Performance and interrupts

biquad

The *biquad* instruction takes 11 cycles to execute, assuming single cycle memory accesses. The ST20-C1 cannot be interrupted during this period.

smacloop

A *smacloop* of *n* steps takes *n*+4 cycles to complete, assuming single cycle memory accesses.

The ST20-C1 cannot be interrupted during this period, which may be up to about 8 microseconds (for single cycle memory and an operating frequency of 33 Mhz). The user may split a long multiply accumulate into a set of shorter ones passing the intermediate accumulator value from one to the next. This will reduce interrupt latency, but loses some numerical accuracy in that within a single *smacloop* intermediate values are held to 48 bits of precision, while values passed from one *smacloop* to another have at most 30 bits of precision (and are saturated).

5.7 Data formats

This section gives details of the data formats used by the *smacloop* and *biquad* instructions.

A signed fractional number of N bits is characterized as x.y, where x+y=N. This means the number is made up from x bits before the binary point, an implied binary point, and y fractional bits. Some examples are listed in Table 5.7.

Total bits	Range	Format	Short name
32	$-1 \leq n < 1$	1.31	Q31
16	$-1 \leq n < 1$	1.15	Q15
16	$-2 \leq n < 2$	2.14	Q14

Table 5.7 Example data formats

5.7.1 Range

A value n in x.y format has a range $-2^{x-1} \leq n < 2^{x-1}$.

For example, a value in the format 1.15 has range $-1 \leq n < 1$, and the format 2.14 has range $-2 \leq n < 2$.

5.7.2 Multiplication

The characteristic of the product of two fractional values is given by:

$$a.b * c.d = a+c.b+d$$

5.7.3 Supported formats

Table 5.8 shows the data formats for multiply-accumulate operations supported by the ST20-C1.

Description	Name	Format
Signed 16-bit fractional	Q14	14 significant fractional bits
Signed 16-bit fractional	Q15	15 significant fractional bits
Signed 16-bit fractional	Q31	31 significant fractional bits

Table 5.8 Supported multiply accumulate data formats

Note that a Q15 value may be (optimally) stored in a 16-bit field, or a wider (>16-bit) field with redundant sign bits. The two storage methods are described below.

5.7.4 Q15 in 16 bits

Q15 format is a 16-bit signed fractional value in the range $-1 \leq n < 1$. The value is stored in two's-complement form with a sign bit (bit 15), an implied binary point between bits 15 and 14, and 15 significant fractional bits (bit 14 to bit 0).

5.7 Data formats

A Q15 value in a 16-bit field is organized as shown in Figure 5.7.

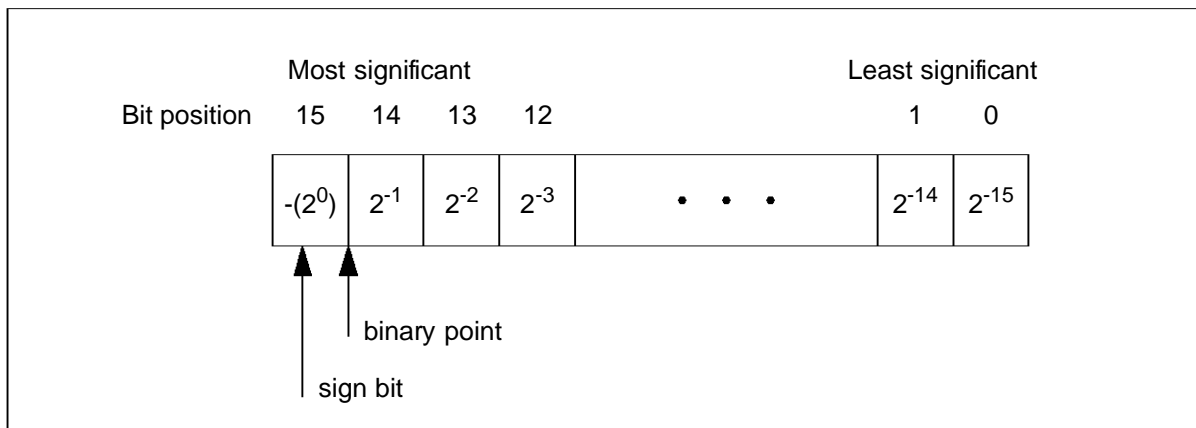


Figure 5.7 Q15 data format (16-bits)

The minimum and maximum representable values are as shown in Table 5.9.

	Hex Value	Decimal Value
Minimum	8000	-1
Maximum	7FFF	+0.9999

Table 5.9 Q15 limiting values

5.7.5 Q15 in an oversized field

When a Q15 value is stored in an oversized field, e.g. a 32-bit register, the significant bits are placed at the least significant end of the field, and the value is sign-extended to the width of the whole field.

The value still has the same range ($-1 \leq n < 1$) and the same number of significant bits.

A Q15 value in a 32-bit field is organized as shown in Figure 5.8.

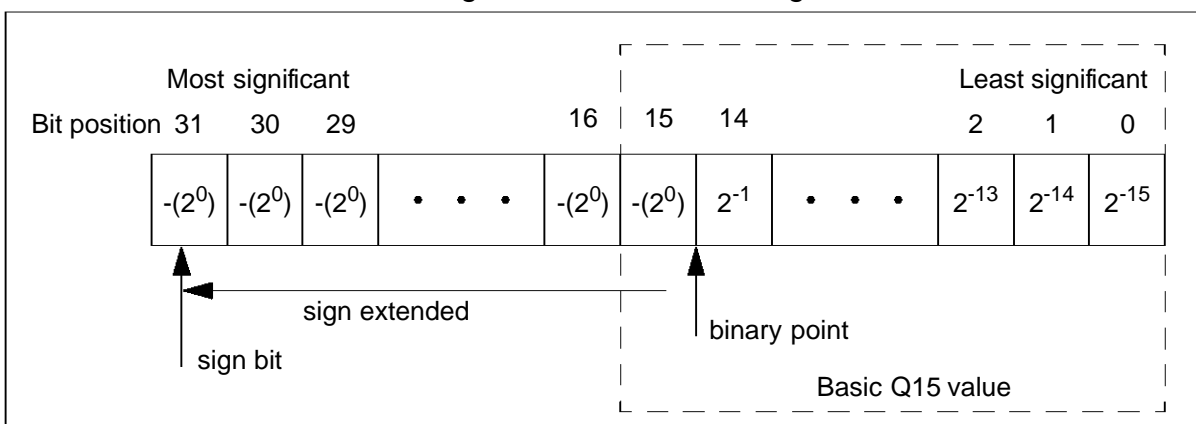


Figure 5.8 Q15 data format (32-bits)

The minimum and maximum representable values are as shown in Table 5.10.

	Hex Value	Decimal Value
Minimum	FFFF8000	-1
Maximum	00007FFF	+0.9999

Table 5.10 Q15 limiting values

Memory access

A Q15 stored in memory may be loaded to **Areg** with a *lsinc* instruction which will automatically sign-extend the value. Similarly a Q15 may be written to memory with a *ssinc* instruction (which will discard the top 16 bits).

Saturation

A well-formed Q15 value is sign extended to the width of the field, and so bits 15 to 31 inclusive will all be identical. Conversely, if the bits from 15 to 31 inclusive are not all identical, then the value is not well-formed. It has either *overflowed* (if $b_{31}=0$, so positive) or *underflowed* (if $b_{31}=1$, so negative).

A Q15 value in **Areg** may be saturated with the sequence:

```
ldc 00007FFF;
order;
ldc FFFF8000;
order;
rev;
```

5.7.6 Q31 format

Q31 format is a 32-bit signed fractional value in the range $-1 \leq n < 1$. The value is stored in two's-complement form with a sign bit (bit 31), an implied binary point between bits 31 and 30, and 31 significant fractional bits (bit 30 to bit 0).

A Q31 value in a 32-bit field is organized as shown in Figure 5.9.

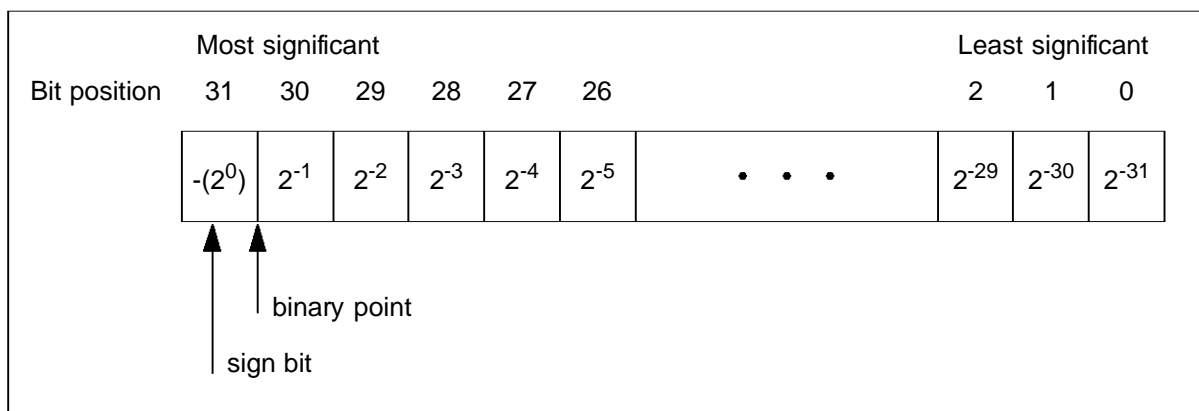


Figure 5.9 Q31 data format

The minimum and maximum representable values are as shown in Table 5.11.

5.7 Data formats

	Hex Value	Decimal Value
Minimum	80000000	-1
Maximum	7FFFFFFF	+0.99999999

Table 5.11 Q31 limiting values

6 Exceptions

This chapter describes exceptions and how to use them. The architecture of the ST20-C1, including the registers and memory arrangement, are described in Chapter 3. A full list of constants and data structures is given in Appendix A.

An *exception* is an exceptional event detected by the ST20-C1 core, which causes a context switch from the normal flow of an executing program. The event triggering the exception may be generated by software inside the core, in which case it is called a trap. Otherwise, the event may be a hardware signal from outside the core, in which case it is called an interrupt, except that the interrupt may attempt to perform a scheduling action, which may cause a trap.

When an exception occurs the CPU changes context to an *exception handler*, which is a section of code only executed when an exception occurs. The process state registers (**Areg**, **Breg**, **Creg**, **Ip**, **Wptr**, **Status** and **Tdesc**) are saved while the exception handler is running, and restored when it returns. Exception handlers for traps are called *trap handlers* and exception handlers for interrupts are called *interrupt handlers*. Normal processes which are not exception handlers are known as *user processes*.

The exception handler is a transient process. Each exception handler starts execution with a standard initial state, runs to completion and terminates with an empty workspace. When the triggering event occurs again, the handler is restarted from its standard initial state and again runs to completion and terminates.

Exception handlers may be nested to arbitrary depth, but they are not re-entrant, so care should be taken to ensure that the exception which caused the handler to run cannot occur while the handler is running, trapped or interrupted. The nesting of exceptions is illustrated in Figure 6.1.

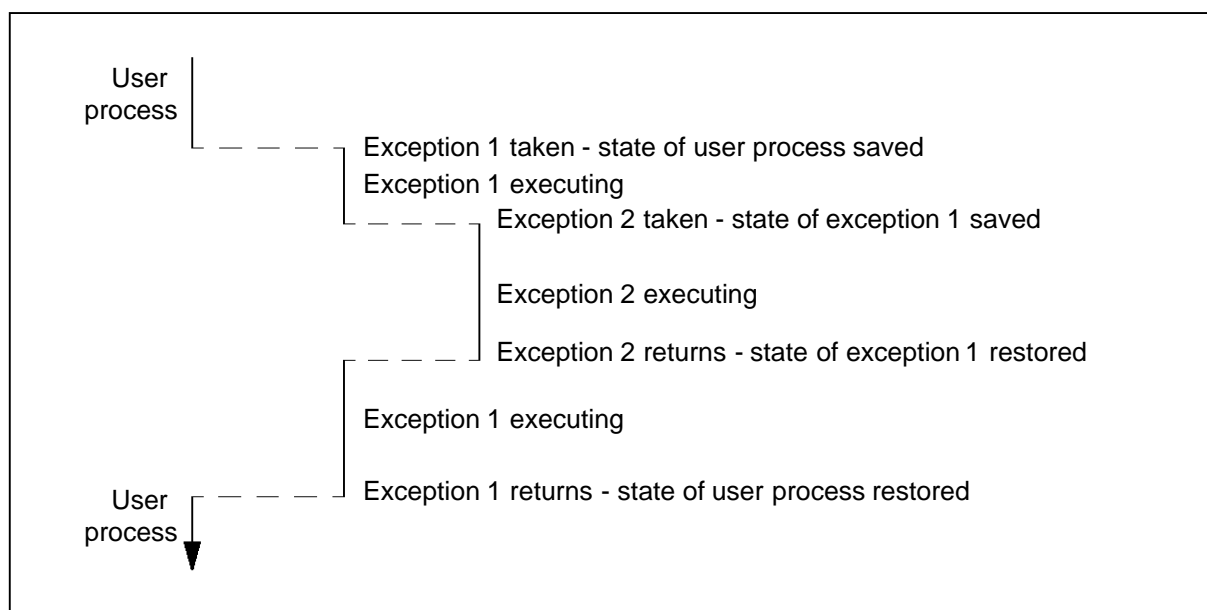


Figure 6.1 Nested exceptions

Exception handler code is completed by executing the *eret* instruction, which restores the state of the interrupted or trapped process. When an interrupt handler executes *eret*, it also signals to the interrupt controller that the interrupt has completed. This allows the interrupt handler to start a lower priority waiting interrupt if required.

The exception instructions are listed in Table 6.1.

Mnemonic	Name
<i>ecall</i>	exception call
<i>eret</i>	exception return
<i>breakpoint</i>	breakpoint

Table 6.1 Exception instructions

6.1 Exception levels

All exception handlers are identified by an integer called the *exception level*. Exception levels 0 to *HighestException* (255) are available for user-defined exceptions, while system exceptions have negative levels, as defined in Table 6.2.

Exception level	Name	Circumstances when taken if not null
0 - 255	-	Interrupt, system call, DMA user process.
-1	el_breakpoint_trap	<i>breakpoint</i> instruction executed or DCU breakpoint request.
-2	el_illegal_instr_trap	Illegal op-code encountered.
-3	el_idle_trap	CPU becomes idle.
-4	el_schedule_exception_trap	Schedule a user process as an exception.
-5	el_run_trap	Execute a <i>run</i> instruction.
-6	el_stop_trap	Execute a <i>stop</i> instruction.
-7	el_timeslice_trap	Take a timeslice.

Table 6.2 Exception levels

Any exception may be triggered from software with the *ecall* instruction. User-defined exceptions can be interrupt handlers, user processes waiting for DMA peripherals or trap handlers used as system calls by executing *ecall*.

System exceptions are traps which may be triggered automatically when the CPU is in certain states. They are intended mainly for operating system kernels to trap scheduling events and for debuggers to trap breakpoints. The circumstances in which each system trap is taken are as follows:

- **el_breakpoint_trap**

This trap is taken when either a *breakpoint* instruction is executed or a diagnostic controller (DCU) signals to the CPU requesting a breakpoint. If the trap is null then the process continues. This trap is used by debuggers.

- **el_illegal_instr_trap**

This trap is taken when the CPU encounters an instruction with an illegal opcode. If the trap is null then the instruction is treated as a *nop*.

- **el_idle_trap**

This trap is taken when the CPU becomes idle, i.e. the current process executes a *stop* when there are no active processes waiting for CPU time, or a *timeslice* is trapped or interrupted so that there are no active processes waiting when the CPU attempts to start the next process. If the trap is null then the CPU waits for an interrupt or for a process to be scheduled. This trap is used by software scheduling kernels.

- **el_schedule_exception_trap**

This trap is taken when an interrupt is received from a peripheral and the exception level is assigned to a user process, which will generally be descheduled waiting for the peripheral to complete a job. If the trap is null then the user process is queued. This trap is used by software scheduling kernels.

- **el_run_trap**

This trap is taken when the *run* instruction is executed. If the trap is null then the CPU adds the process to the back of the scheduling queue. This trap is used by software scheduling kernels.

- **el_stop_trap**

This trap is taken when the *stop* instruction is executed. If the trap is null then the current process is descheduled and the CPU starts executing the process on the front of the scheduling queue, or goes idle if there is none. This trap is used by software scheduling kernels.

- **el_timeslice_trap**

This trap is taken when a timeslice is due and enabled and the *timeslice* instruction is executed. If the trap is null then the current process is timesliced, i.e. placed on the back of the scheduling queue. This trap is used by software scheduling kernels.

Scheduling and timeslices are discussed in Chapter 7. Using user processes as exceptions to handle peripherals is described in section 4.11.

6.2 Exception vector table

The *exception vector table* maps each exception level to a user process or exception. The base of the exception vector table is at the fixed address *ExceptionBase* (#80000040) in on-chip memory, and the exception level is the word offset from the *ExceptionBase* to the vector. Thus the exception level is used as an index into the exception vector table.

The address of the user process or exception is always word aligned, and so has bits 0 and 1 set to zero. The entry in the exception vector table is a *descriptor*, which consists of the address of the user process or exception ORed with a type. The *type* is

bit 0 of the descriptor, and so can be either *ExceptionProcessType* (which has the value 1) or *UserProcessType* (which has the value 0). The type allows each exception level to be assigned to one of the following:

- an exception handler. The entry for an exception handler is the address of the exception control block (as described in section 6.3) bitwise ORed with *ExceptionProcessType* to indicate an exception.
- a user process waiting for a peripheral (as described in section 4.11). The entry for a user process is the task descriptor (i.e. the address of the process control block) bitwise ORed with *UserProcessType* to indicate a user process.
- the null entry *NotProcess*. The CPU treats null entries as disabled exceptions.

When the exception is triggered, the CPU looks in the table entry for the requested exception level. If the value in the table is *NotProcess* then no exception or trap is taken and the CPU continues with the default behavior. Otherwise, if bit 0 is *UserProcessType* then the address part of the value is assumed to be a valid task descriptor of a process. If bit 0 is *ExceptionProcessType*, then the address part is assumed to be a pointer to a valid exception control block.

Using user processes as exceptions to handle peripherals is described in section 4.11. The rest of this chapter refers only to exceptions.

Typically a separate descriptor is used for each interrupt level and system call, so that this scheme provides a system of vectored interrupts and system calls. The exception handler is then executed, and can itself be interrupted or trapped.

6.3 Exception control block and the saved state

When an exception is taken, the state is saved in the exception control block. The evaluation stack, status register, **lptr**, **Wptr** and **Tdesc** are automatically saved on taking the exception and restored on returning. This is done to enable any exception handler to have direct access to the state of the underlying process and is required for some software scheduler implementations.

The constants in Table 6.3 define the locations in the control block.

Word offset	Name	Purpose
7	ex.HandlerIptr	Exception handler instruction pointer.
6	ex.InterptdStatus	Interrupted or trapped process status register.
5	ex.InterptdTdesc	Interrupted or trapped process task descriptor.
4	ex.InterptdIptr	Interrupted or trapped process instruction pointer.
3	ex.InterptdWptr	Interrupted or trapped process workspace pointer.
2	ex.InterptdCreg	Interrupted or trapped process Creg .
1	ex.InterptdBreg	Interrupted or trapped process Breg .
0	ex.InterptdAreg	Interrupted or trapped process Areg .

Table 6.3 Exception control block

6.4 Initial exception handler state

The control block is at the initial **Wptr** for the exception handler, so the locations are word offsets from the initial **Wptr** of the exception handler. The initial **Wptr** is the address of the exception control block which is the address part of the entry in the exception vector table.

6.4 Initial exception handler state

When the exception handler starts, **Wptr** is set to the address of the exception control block. The work space of the exception handler is normally below the control block, so like a function or procedure call, one of the first actions of the exception handler code is to adjust the **Wptr** downwards to create space for local variables using *ajw*. The **Wptr** must be adjusted back up again before the handler returns.

The initial **lptr** of the exception handler is the value loaded from **ex.Handlerlptr** in the exception control block. The state of the interrupted or trapped process is saved in the exception control block. If the exception is an idle trap then the saved state is the state of the last descheduled process. Initially the status register is set to the values shown in Table 6.4.

Field or bit	Value
mac_count	As in the interrupted or trapped process.
mac_buffer	As in the interrupted or trapped process.
mac_scale	As in the interrupted or trapped process.
mac_mode	As in the interrupted or trapped process.
global_interrupt_enable	False if exception is a trap, otherwise preserved.
local_interrupt_enable	As in the interrupted or trapped process.
overflow	False.
underflow	False.
carry	False.
user_mode	False.
interrupt_mode	True if any interrupts are running, false otherwise.
trap_mode	True if exception is a trap, false otherwise.
sleep	False.
reserved	Undefined.
start_next_task	False.
timeslice_enable	False.
timeslice_count	As in the interrupted or trapped process.

Table 6.4 Exception handler initial status register

If the exception handler is interrupting a user process, then the address of the exception control block (i.e. the user process state) is left in **Tdesc**. If necessary, the exception handler can save this address. If a sequence of nested interrupts has

occurred then this is the only way that the nested interrupts can identify the state of the user process.

If the exception is a **schedule_exception** trap then the trap handler also needs to know the process due to be scheduled. The descriptor of the process (as held in the exception vector table) is saved at *SavedTaskDescriptor* near the bottom of the address space.

6.5 Restrictions on exception handlers

Exception handlers cannot be queued, so they must not deschedule. This means that the following are not permitted inside exception handlers:

- the *stop* instruction;
- the *timeslice* instruction.

Exception handlers may be nested to arbitrary depth, but they are not re-entrant, so care should be taken to ensure that the exception which caused the handler to run cannot occur while the handler is running, trapped or interrupted.

6.6 Interrupts

All interrupts, whether from on-chip peripherals or external pins, are routed through an interrupt controller, which is normally an on-chip peripheral. The interrupt controller is responsible for arbitration between multiple interrupt signals. The design of the interrupt controller varies between ST20 variants. Typically, interrupt priorities are managed by the interrupt controller, which will usually track the priority of the highest level task currently executed by the core, and will interrupt the ST20-C1 again if a higher priority interrupt occurs.

When an interrupt is requested by the interrupt controller, the ST20-C1 always changes context to the appropriate interrupt handler. An interrupt request is accompanied by an identifier for the interrupt handler, which is the exception level. The ST20-C1 scheduler uses the exception level from the interrupt controller to start the appropriate interrupt handler.

The CPU also sets the **interrupt_mode** bit in the status register and clears the **trap_mode** and **user_mode** bits. The **interrupt_mode** bit indicates that an interrupt handler is running, though it may have been trapped.

When the interrupt handler executes the *eret* instruction, the CPU signals to the interrupt controller that the handler has returned. This allows the interrupt controller to keep track of which interrupt handlers are running, so that it can start a low priority waiting interrupt when a higher priority handler completes.

If the interrupt controller requests an interrupt level which has a null interrupt handler then the CPU signals to the controller that the interrupt has completed.

6.7 Traps

The ST20-C1 has a system of traps which act as software generated interrupts. Any level of exception can be called as a user exception using *ecall*. This mechanism can be used for system calls to an operating system. In addition some special exception levels are reserved for system use to provide for the trapping of breakpoints, scheduling events, illegal operations and the machine becoming idle. The reserved 'system' exception levels are described in section 6.1.

If a system trap event occurs or a trap is called at an exception level which has a null trap handler then the CPU ignores the trap and continues.

When a non-null trap handler is started, the **trap_mode** bit of the status register is set and the **user_mode** bit is cleared. The **interrupt_mode** bit is not altered. The **trap_mode** bit indicates that a trap handler is currently executing, so it is cleared if the trap handler is interrupted.

6.8 Setting up the exception handler

To create an exception handler, an exception work space area must be created, with enough space for the exception handler's stack and 8 words at the top of the work space for the exception control block, as defined in Table 6.3. For minimum interrupt latency, interrupt handler control blocks should be in fast memory, preferably on-chip. The normal work space and control block of an exception handler is shown in Figure 6.2. In the control block, **ex.HandlerIp** must be initialized to point to the entry point of the exception handler code.

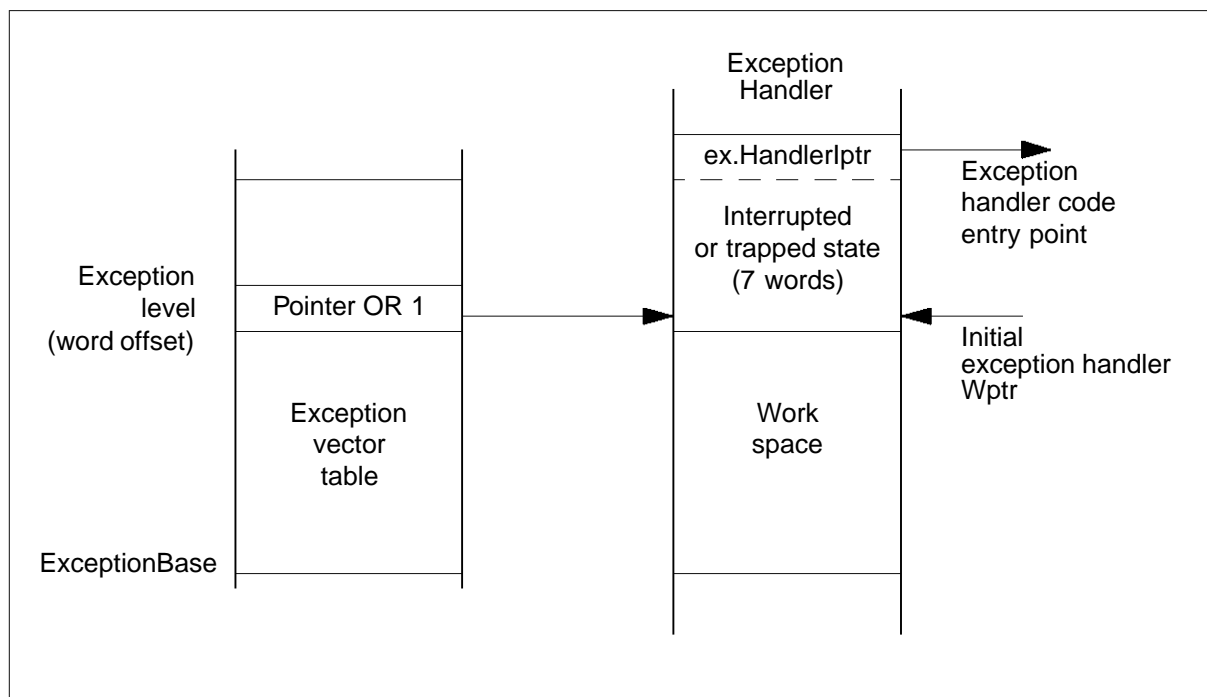


Figure 6.2 Exception handler

The code of the exception handler may access or modify the state of the interrupted or trapped process. The state of the interrupted or trapped process is stored in the exception control block, which is located at the initial **Wptr** of the exception handler.

An exception handler must use *eret* to return to the interrupted or trapped process.

6.8.1 Enabling and disabling exceptions

When the exception handler has been created and initialized, the exception can be enabled. The address of the exception handler control block, ORed with 1 to set the exception type bit, must be written in the exception vector table at the level for the exception. To write an entry *control_block* with type *ExceptionProcessType* in the exception vector table, the following code may be used:

```
ld control_block; ldc ExceptionProcessType; or;  
ld ExceptionBase;  
stnl exception_level;
```

For interrupts, both the *global_interrupt_enable* and *local_interrupt_enable* bits in the status register must be set. In addition the interrupt controller may need to be initialized, including any interrupt enable bits and masks.

A trap can be disabled by writing *NotProcess* into the exception vector table.

Interrupts can be disabled in four ways:

- 1 Clearing the status register bit **global_interrupt_enable** disables all interrupts until the bit is set by an explicit write to the status register.
- 2 Clearing the status register bit **local_interrupt_enable** disables all interrupts until the current process is descheduled.
- 3 A single exception level can be disabled by writing *NotProcess* into the exception vector table.
- 4 The interrupt controller will generally have a means of disabling interrupts individually or globally by writing to interrupt controller registers.

7 Multi-tasking

This chapter describes the features of the ST20-C1 core provided to support multi-tasking, and how to use them. The architecture of the ST20-C1, including the registers and memory arrangement, are described in Chapter 3. Interrupts and traps are described in Chapter 6. A full list of constants and data structures is given in Appendix A.

Support is provided in the instruction set for timeslicing, scheduling processes and manipulating queues of processes.

7.1 Processes

A process (also known as a task or thread) is an independent unit of software with a single thread of control, i.e. a sequential algorithm. Any number of processes may be run. A process which has been started but not terminated may be in one of several different states:

- *executing* on the CPU;
- *interrupted or trapped* by an exception;
- *inactive*, i.e. waiting for a peripheral or semaphore signal;
- waiting for CPU time.

A process that is not inactive is said to be active. A process that is not executing or interrupted is said to be descheduled. The states and main transitions are shown in Figure 7.1. The scheduling transitions can be trapped so that a software scheduling kernel can modify the transitions and so change the scheduling behavior, for example by providing a system of process priorities.

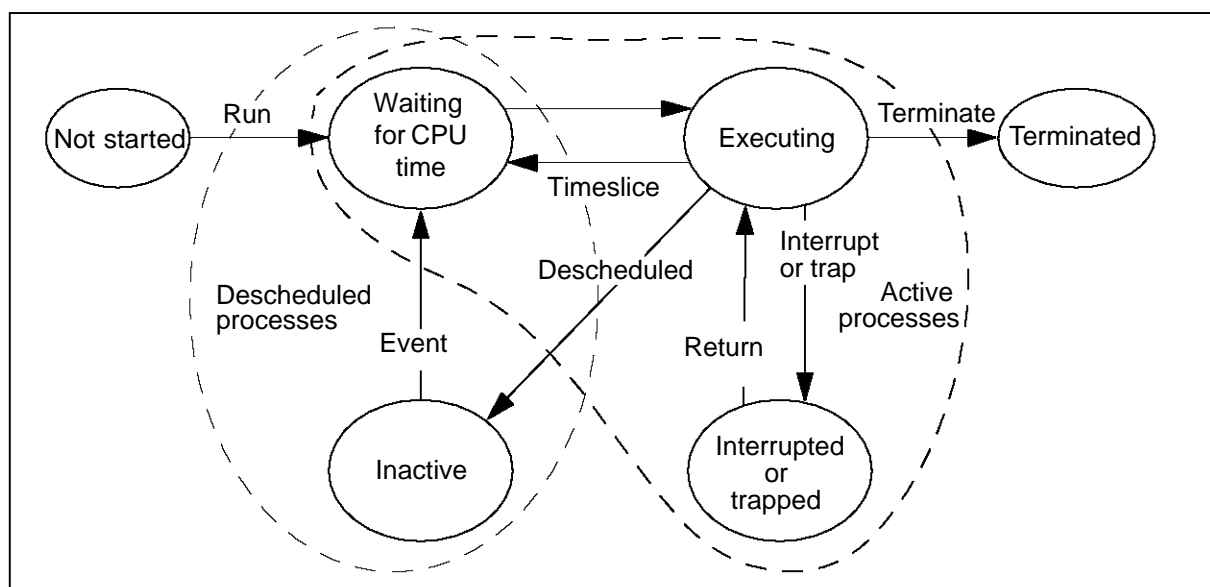


Figure 7.1 Process states and main transitions

A process state is held in memory and the CPU registers. Sufficient of the register state must be saved when the process is interrupted or a context switch occurs, so that the process can be reloaded and continue execution at a later time. The register state consists of:

- the instruction pointer register;
- the work space pointer register;
- the task descriptor register;
- the evaluation stack registers;
- the status register.

In order to save memory space and context switch time, processes are only descheduled when the evaluation stack and status register are empty. This is achieved by only allowing processes to deschedule at certain instructions, called deschedule instructions, after which the final values in the evaluation stack are undefined and the status register is reset to a default value. The deschedule instructions are *stop* and *timeslice*.

Table 7.2 lists the multi-tasking instructions.

Mnemonic	Name
<i>run</i>	Run process
<i>stop</i>	Stop process
<i>timeslice</i>	Timeslice
<i>ldtdesc</i>	Load task descriptor
<i>enqueue</i>	Enqueue a process
<i>dequeue</i>	Dequeue a process

Table 7.2 Multi-tasking instructions

7.2 Descheduled processes

If the process is waiting for a peripheral or a semaphore or descheduled by a timeslice then the evaluation stack is not saved. The instruction pointer and **Wptr** are saved in the *process descriptor block*. The task descriptor is the address of the process descriptor block. It therefore identifies the waiting process and points to its saved state. The task descriptor is a fixed address for each process, unlike the **Wptr** which changes as the code executes. When the process is running, the task descriptor is held in the **Tdesc** register.

Word offset	Slot name	Purpose
2	pw.lptr	The process saved instruction pointer.
1	pw.Wptr	The process saved work space pointer.
0	pw.Link	The link to the next process in the queue.

Table 7.1 Process descriptor block

7.3 Queues

The structure of the process descriptor block is shown in Table 7.1. When the process is not executing, it contains the saved work space pointer and instruction pointer of the process, plus a queue link if the process is in a queue.

Figure 7.3 illustrates a descheduled process.

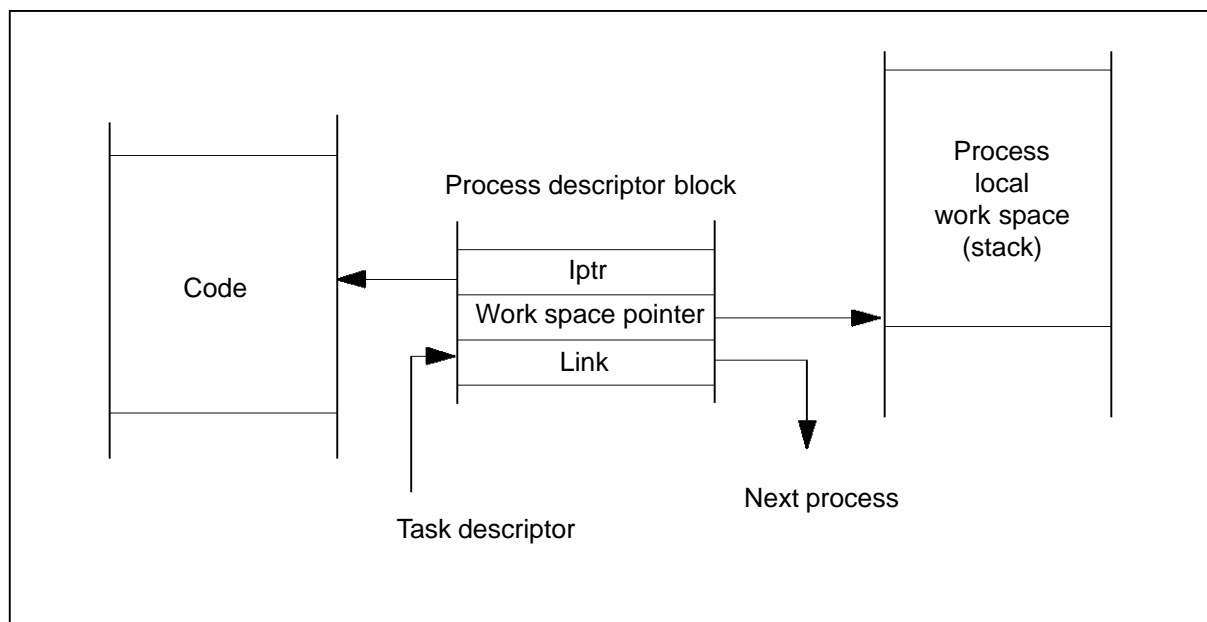


Figure 7.3 A descheduled process

7.3 Queues

There may be any number of processes waiting for execution, so a queue (i.e. a linked list) of waiting processes is formed, called the *scheduling queue*. This is an example of a general queue supported by the instruction set for queueing waiting processes.

A queue is a linked list of process control blocks, formed by links included in the process control blocks. Each link points to the control block of the next process in the queue unless it is the last in the queue, which is undefined.

The front and back pointers of a queue are held in a queue control block, as shown in Table 7.2. The queue control block is held in memory, and the address of the block is the identifier of the queue.

Word offset	Slot name	Purpose
1	q.BPtrLoc	The back of the queue.
0	q.FPtrLoc	The front of the queue.

Table 7.2 Queue control block

A complete queue is illustrated in Figure 7.4. In the case of the scheduling queue, the control block is stored at the reserved address called *SchedulerQptr* (which has the

value *MostNeg*) at the bottom of the memory space.

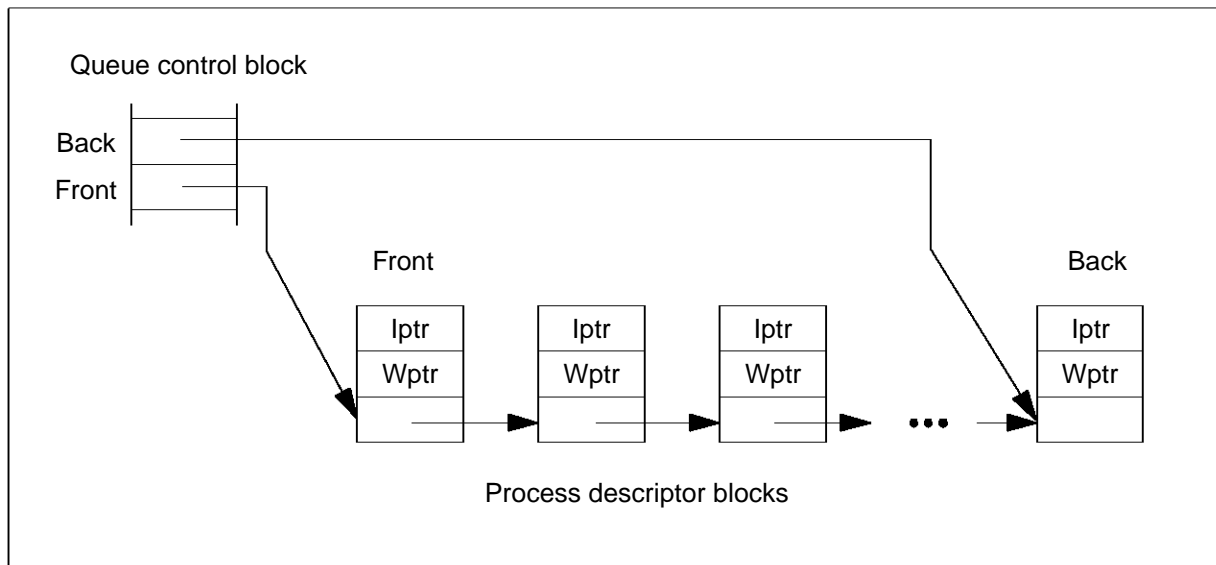


Figure 7.4 A process queue

7.4 Timeslicing

The ST20-C1 includes support for timeslicing. Timeslicing is a safeguard in a multi-tasking environment to prevent any one process from taking too much processor time. Exception handlers are not timesliced.

A timeslicing counter is provided as a field of the status register called the **timeslice_count**. It is reset to *MaxTimesliceCount* each time a user process is loaded from the scheduling queue into the CPU. The counter is decremented regularly until it reaches 0, and then stays at 0. A timeslice is due when the counter value is 0.

If a *timeslice* instruction is executed when a timeslice is due and timeslicing is enabled then:

- the timeslice trap will be taken if it is installed;
- otherwise the current process will be timesliced, i.e. the current process is placed on the back of the scheduling queue and the process on the front of the scheduling queue is loaded into the CPU for execution.

If an exception occurs, the value of the counter is saved with the status register and reloaded when the interrupted or trapped process is restarted. This ensures that a process will be executed for roughly the same time regardless of whether it was interrupted or trapped.

The **timeslice_enable** bit of the status register can be used to enable or disable timeslicing. Timeslicing is enabled when the bit is set. This bit is preserved when a process is descheduled, so it may be treated as global among user processes. Timeslicing must not be enabled in exception handlers.

7.5 Inactive processes

An inactive process is a process that is waiting for some event to occur, such as a process executing a *run* instruction or a peripheral completing a DMA. An inactive process cannot continue even if the CPU is idle. Inactive processes are not polled, but should be rescheduled by the event for which they are waiting.

A process becomes inactive by saving its own state and then executing the *stop* instruction. *stop* automatically saves the **Ip**tr and **Wp**tr in the process control block. The code should save the **Tdesc** in an appropriate place where the awaited event can find it. The **Tdesc** points to the process control block. The **Ip**tr is loaded onto the evaluation stack using *ldpi*, the **Wp**tr using *ldlp 0* and the **Tdesc** using *ldtdesc*. For example:

```
ldtdesc; ld tdesc_save_address; stnl 0;
stop;
```

Two mechanisms are provided for rescheduling inactive processes:

- Executing the *run* instruction. This allows other processes to reschedule an inactive process. For example, this is used when semaphore signalling to a waiting process and could be used by other communications, possibly implemented by an operating system.
- Scheduling an exception. This allows external devices to reschedule an inactive process. For example, this is used by DMA peripherals to wake a process waiting for the DMA to complete. Exception scheduling is described in section 4.11.

The *run* instruction takes a task descriptor in **Areg** and adds the process control block to the back of the scheduling queue. For example:

```
ld task_descriptor; run;
```

Inactive processes may need to be queued, for example while waiting for a semaphore. The semaphore queue handling is incorporated into the signal and wait instructions, but for other queues the instructions *enqueue* and *dequeue* are needed. *Enqueue* will add a process to the back of an arbitrary queue, and *dequeue* will take a process from the front of the queue.

7.6 Descheduled process state

When a process restarts after an interrupt or trap, the entire state is loaded from the saved state. However, when the process starts or restarts after being descheduled, the CPU makes assumptions about the state of the process, since not all the state was saved.

Wptr and **Ip**tr are set to the values saved in **pw.Wp**tr and **pw.Ip**tr of the process descriptor block.

The status register is set to the values shown in Table 7.3. The global interrupt enable and timeslice enable are global status bits, and are carried over from one process to the next when a context switch occurs.

Field or bit	Value
mac_count	As in previous process.
mac_buffer	As in previous process.
mac_scale	As in previous process.
mac_mode	As in previous process.
global_interrupt_enable	As in previous process.
local_interrupt_enable	Set.
overflow	False.
underflow	False.
carry	False.
user_mode	True.
interrupt_mode	False.
trap_mode	False.
sleep	False.
reserved	Undefined.
start_next_task	False.
timeslice_enable	As in previous process.
timeslice_count	MaxTimesliceCount.

Table 7.3 Restarted process status register

7.7 Initializing multi-tasking

7.7.1 Initializing the scheduling queue

Before multi-tasking operations can be performed, the scheduling queue must be initialized. The scheduling queue is described in section 7.3. The queue must be initialized by setting it to empty, which means setting the front pointer to empty:

```
ld MostNeg; ld SchedulerQptr; stnl q.FPtrLoc;
```

7.7.2 Creating and starting a process

A process consists of code, a work space area, a process control block and the values for the **Iptra**, **Wptr** and **Tdesc**. To create a process, a large enough work space is created, and a process control block of three words. It may be convenient to put all the process control blocks together in one area of memory. For fast context switches and minimal interrupt latency, process control blocks should be in on-chip memory.

The entry point for the process code is written into **pw.Iptr** of the process control block. The address of the top of the work space is written into **pw.Wptr** of the process control block, so the process code must adjust the initial **Wptr** down using *ajw* to create space for local variables.

7.8 Scheduling kernels

The following code would set up a process control block:

```
ld entry_point; ld control_block; stnl pw.lptr;
ld work_space_top; arot; stnl pw.Wptr;
```

The process is then inactive, so it can be added to the back of the scheduling queue using *run*, exactly as in section 7.5.

7.8 Scheduling kernels

A scheduling kernel can be written to override the default scheduling behavior of the ST20-C1, but still using the very fast micro-code scheduling. The basis of such a scheduler would be trap handlers to trap the system exceptions **el_idle_trap**, **el_run_trap**, **el_stop_trap**, **el_timeslice_trap** and **el_schedule_exception_trap**. When these traps are taken, a scheduling event is due or the processor has become idle. The trap replaces the default scheduling action. When the trap handler returns, the CPU will continue with the next instruction. Traps are described in Chapter 6.

Additional system calls may be implemented by user-defined trap handlers, called using *ecall*.

7.9 Semaphores

Semaphores are a mechanism for managing access to shared resources within a multi-tasking environment. The semaphore operations are provided by the instructions listed in Table 7.4.

Mnemonic	Name
<i>signal</i>	signal
<i>wait</i>	wait
<i>stop</i>	stop process
<i>run</i>	run process

Table 7.4 Semaphore instructions

The semaphore instructions act on a semaphore control block, defined in Table 7.5.

Word offset	Slot name	Purpose
2	s.Back	The back of the semaphore waiting list.
1	s.Front	The front of the semaphore waiting list.
0	s.Count	The unsigned number of extra processes that the semaphore will allow to continue running on a <i>wait</i> request.

Table 7.5 Semaphore control block

Each semaphore has a semaphore control block, which implements a linked list of waiting processes and a count of free resources. The count should be initialized to the total number of available resources (usually one), and the front list pointer should be initialized to the empty value *NotProcess*. For fast signalling and minimal interrupt latency, semaphore control blocks should be in fast memory, preferably on-chip.

7.9.1 Waiting for a resource

A process requesting a resource executes a wait (or P), which is the following code sequence:

```
ld semaphore; wait;
cj CONTINUE;
stop;
CONTINUE:
```

The *wait* instruction is executed, with a pointer to the semaphore control block in **Areg**. The action of *wait* depends on the count in the semaphore control block. If the count is not zero, then a resource is free, so the count of free resources is decremented and the value *false* is left in the **Areg** to indicate that the process can continue. If the count is zero then there are no free resources, and the process is added to the list of waiting processes, and the value *true* is left in **Areg** to indicate that the process must wait. A conditional jump then tests **Areg** and performs a *stop* if the process must wait. *stop* saves the **Ip**tr and **Wp**tr and deschedules the process, which will wait on the semaphore queue until another process performs a *signal* and subsequently restarts the process with a *run* instruction.

7.9.2 Freeing a resource

When a process finishes with a resource and can free it, it performs a signal (or V), which is the following code sequence:

```
ld semaphore; signal;
cj CONTINUE;
run;
CONTINUE:
```

The *signal* instruction is executed, with a pointer to the semaphore control block in **Areg**. The action of *signal* depends on whether a process is waiting or not. If the front pointer of the process waiting list is empty then there are no processes waiting, so the count is incremented and **Areg** is set to *false*. Otherwise, at least one process is waiting, so the first process is removed from the list and placed in the **Breg**, and **Areg** is set to *true*. A conditional jump tests **Areg** and performs a *run* to restart the process if there was one waiting.

7.10 Sleep

When the ST20-C1 becomes *idle* it disables counter distribution to its circuits and consumes a very small amount of electrical power; this is known as *sleep* mode. The counters are re-enabled and normal operation resumes automatically and instantly when either an interrupt or a software reset from the diagnostic controller is received.

Sleep mode may also be triggered directly from software by setting the sleep bit in the status register:

```
ldc sleep; bitmask; statusset;
```

In this case also, sleep mode persists until the next interrupt or software reset.

8 Instruction Set Reference

The following pages define the actions of each instruction in the ST20-C1 instruction set. The notation used is described in Chapter 2. The use of the instructions is described in Chapter 4, Chapter 6 and Chapter 7. The constants and data structures are listed in Appendix A.

adc n

add constant

Code: Function 8

Description: Add a constant to **Areg**.

Definition:

```
if (sum > MostPos)
{
    Areg' ← sum – 2BitsPerWord
    if (Status'underflow ≠ set )
        Status'overflow ← set
}
else if (sum < MostNeg)
{
    Areg' ← sum + 2BitsPerWord
    if (Status'overflow ≠ set )
        Status'underflow ← set
}
else
{
    Areg' ← sum
}
```

where sum = Areg + n
 – the value of sum is calculated to unlimited precision

Status Register:

Overflow or underflow bit may be set.

Comments:

Primary instruction.

See also: *add addc ldhlp*
section 4.4.

add

add

Code: F4**Description:** Add **Areg** and **Breg**.**Definition:**

```

if (sum > MostPos)
{
    Areg' ← sum – 2BitsPerWord
    if (Status'underflow ≠ set )
        Status'overflow ← set
}
else if (sum < MostNeg)
{
    Areg' ← sum + 2BitsPerWord
    if (Status'overflow ≠ set )
        Status'underflow ← set
}
else
{
    Areg' ← sum
}

Breg' ← Creg
Creg' ← Areg

where    sum = Areg + Breg
           – the value of sum is calculated to unlimited precision

```

Status Register:

Overflow or underflow bit may be set.

Comments:

Secondary instruction.

See also: *adc addc*
section 4.4.

addc

add with carry

Code: 21 F0

Description: Add **Areg** and **Breg**, unsigned, with carry propagation. This instruction is provided for long arithmetic; address calculations may be performed with *add*, *sub* and *adc* without affecting the carry flag.

Definition:

```
if (sum < 2BitsPerWord)
{
    Areg'unsigned ← sum
    Status'carry ← clear
}
else
{
    Areg'unsigned ← sum - 2BitsPerWord
    Status'carry ← set
}

Breg' ← Creg
Creg' ← Areg

where    sum = Aregunsigned + Bregunsigned + Statuscarry
           – the value of sum is calculated to unlimited precision
```

Status Register:

Carry bit is set or cleared.

Comments:

Secondary instruction.

See also: *adc add subc*
section 4.4.

ajw n

adjust work space

Code: Function B**Description:** Move the workspace (stack) pointer by the number of words specified in the operand, in order to allocate or de-allocate workspace stack slots.**Definition:**
$$Wptr' \leftarrow Wptr @ n$$
Status Register:

No effect

Comments:

Primary instruction.

See also: *fcall gajw*
section 4.10.

and

and

Code: F9

Description: Bitwise AND of **Areg** and **Breg**.

Definition:

$Areg' \leftarrow Breg \wedge Areg$

$Breg' \leftarrow Creg$

$Creg' \leftarrow Areg$

Status Register:

No effect

Comments:

Secondary instruction.

See also: *not or xor*
section 4.8.

arot

anti-rotate stack

Code: F3

Description: Rotate the evaluation stack downwards. This instruction may be used to recover values rotated onto the bottom of the stack, e.g. by *cj*.

Definition:
$$\begin{aligned} \text{Areg}' &\leftarrow \text{Creg} \\ \text{Breg}' &\leftarrow \text{Areg} \\ \text{Creg}' &\leftarrow \text{Breg} \end{aligned}$$
Status Register:

No effect

Comments:

Secondary instruction.

See also: *dup rev rot*
section 4.1.

ashr

arithmetic shift right

Code: 21 FF

Description: Perform an arithmetic shift right of **Breg** by **Areg** bits, copying the sign bit into the vacated bits. **Breg**, not **Areg**, is rotated into **Creg**, to preserve the value rather than the shift length.

Definition:

$$\text{Areg}' \leftarrow (\text{Breg} \gg_{\text{arith}} \text{Areg}_{\text{unsigned}})$$
$$\text{Breg}' \leftarrow \text{Creg}$$
$$\text{Creg}' \leftarrow \text{Breg}$$

Status Register:

No effect

Comments:

Secondary instruction.

If **Areg** is not in the range 0..31 then the result is undefined.

See also: *shl shr*

section 4.9.

biquad

biquad IIR filter step

Code: 21 F7

Description: Execute a step of a biquad IIR filter on vectors of 16-bit values. **Areg** points to the C coefficient vector, **Breg** points to the X input data vector and **Creg** points to the Y results vector. **Areg** must be word-aligned and **Breg** and **Creg** half-word aligned. *biquad* increments **Breg** and **Creg** by 2 bytes and performs the five multiply accumulates $Y[2] = X[0].C[0] + X[1].C[1] + X[2].C[2] + Y[0].C[3] + Y[1].C[4]$.

Definition:

```

if (overflow)
    if (Status'underflow ≠ set) Status'overflow ← set
else if (underflow)
    if (Status'overflow ≠ set) Status'underflow ← set
else
{
    sixteen'[Creg + 4] ← acc >>arith23
    Breg' ← Breg + 2
    Creg' ← Creg + 2
}
where    acc = 1 << 22 + ((sixteen[Areg] << shift) × sixteen[Breg])
          + ((sixteen[Areg+2] << shift) × sixteen[Breg+2])
          + ((sixteen[Areg+4] << shift) × sixteen[Breg+4])
          + ((sixteen[Areg+6] << shift) × sixteen[Creg])
          + ((sixteen[Areg+8] << shift) × sixteen[Creg+2])
          – the value of acc is calculated to 48-bit precision
if (Statusmac_scale = 3) shift = 9
else                      shift = 4 × Statusmac_scale

```

Status Register:

May set underflow or overflow.

Comments:

Areg must be word-aligned.

Breg and **Creg** must be half-word aligned, and must either be both word-aligned or neither word-aligned.

This instruction may take 8 memory accesses, which will affect interrupt latency.

See also: *mac, smacloop, umac*
Chapter 5.

bitld

load bit

Code: 22 F3

Description: Get a bit of **Breg**. The bit number is given by **Areg**. **Breg** is rotated into **Creg**, to preserve the value rather than the bit number.

Definition:

$Areg' \leftarrow (Breg \gg Areg) \wedge 1$

$Breg' \leftarrow Creg$

$Creg' \leftarrow Breg$

Status Register:

No effect

Comments:

Secondary instruction.

Areg must be in the range 0..31.

See also: *bitst bitmask*
section 4.8.

bitmask

create bit mask

Code: 22 F5

Description: Create a bit mask with a single bit set. The value in **Areg** indicates the bit that is to be set.

Definition:

$\text{Areg}' \leftarrow 1 \ll \text{Areg}$

Status Register:

No effect

Comments:

Secondary instruction.

Areg must be in the range 0..31.

See also: *bitld*
section 4.8.

bitst

store bit

Code: 22 F4

Description: Overwrite the bit position **Areg** of the value in **Creg** with a single bit with the value given by **Breg**. **Breg** is rotated into **Creg**, to preserve the value rather than the bit number.

Definition:

$$\text{Areg}' \leftarrow (\text{Creg} \wedge \sim(1 \ll \text{Areg})) \vee (\text{Breg} \ll \text{Areg})$$

$$\text{Breg}' \leftarrow \text{Creg}$$

$$\text{Creg}' \leftarrow \text{Breg}$$

Status Register:

No effect

Comments:

Secondary instruction.

Areg must be in the range 0..31.

Breg must be 0 or 1.

See also: *bitld bitmask*
section 4.8.

breakpoint

breakpoint

Code: FF**Description:** Take a breakpoint trap if a breakpoint trap is installed.**Definition:**

if (*breakpoint trap installed*)
 take breakpoint trap

Status Register:

If the trap is taken then the status register is saved and the trap handler status register is loaded.

Comments:

Secondary instruction.

This instruction is a short secondary instruction, encoded in a single byte.

See also:

Chapter 6.

cjn

conditional jump

Code: Function A

Description: Jump if **Areg** is 0 (i.e. jump if *false*). The destination of the jump is expressed as a byte offset from the instruction following the conditional jump.

Definition:

```
if (Areg = false)
    lptr' ← next instruction + n
else
{
    Areg' ← Breg
    Breg' ← Creg
    Creg' ← Areg
}
```

Status Register:

No effect

Comments:

Primary instruction.

The initial **Areg** can be recovered using *arot*.

See also: *eqc gt gtu j jab*
section 4.6.

dequeue

dequeue a process

Code: 23 F8

Description: Load into **Breg** a pointer to the first process from the queue control block pointed to by **Areg**. Load into **Areg** a boolean value indicating whether a process was found. The queue control block is defined in Chapter 7.

Definition:

```

if (frontptr = NotProcess)                                -- queue is empty
{
    Areg' ← false
    Breg' ← undefined
}
else                                                        -- queue is not empty
{
    Areg' ← true
    Breg' ← frontptr
    if (frontptr = backptr)                                  -- one process on queue
        word'[Areg @ q.FPtrLoc] ← NotProcess
    else                                                      -- many processes on queue
        word'[Areg @ q.FPtrLoc] ← word[frontptr @ pw.Link ]
    }

    Creg' ← undefined

where frontptr = word[Areg @ q.FPtrLoc]
        backptr = word[Areg @ q.BPtrLoc]

```

Status Register:

No effect

Comments:

Secondary instruction.

This instruction may require 4 memory accesses. If the queue structure is in off-chip memory then interrupt latency may be affected.

Areg must be word aligned (i.e. divisible by 4).

See also: *enqueue run*
Chapter 7.

divstep

divide step

Code: 21 F8

Description: Perform a step of an integer division to generate 4 bits of the quotient. **Areg** holds the positive divisor. **Breg** and **Creg** hold the non-negative dividend or partial remainder and the partial result. Before the first step, **Breg** holds the dividend and **Creg** must be 0. After performing *divstep* 8 times, **Breg** will hold the quotient and **Creg** the remainder.

Definition:

```
if (Areg > 0 and Breg ≥ 0 and Creg ≥ 0 )
{
    Breg' ← (Breg << 4) ∨ (part_div / Areg)
    Creg' ← part_div rem Areg
}
else
{
    Breg' ← undefined
    Creg' ← undefined
}

where    part_div = (Creg << 4) ∨ (Breg >> 28)
        – the value of part_div is calculated to unlimited precision
```

Status Register:

No effect.

Comments:

Secondary instruction.

Areg must be greater than zero.

Breg and **Creg** must be not less than zero.

See also: *unsign*
section 4.4.

dup

duplicate top of stack

Code: F1

Description: Duplicate the top of the integer stack.

Definition:

Breg' \leftarrow Areg
Creg' \leftarrow Breg

Status Register:

No effect

Comments:

Secondary instruction.

See also: *arot rev rot*
section 4.8.

ecall

exception call

Code: 23 F1

Description: Take a trap with exception level indicated by **Areg**. This instruction can be used to implement system calls to an operating system.

Definition:

Areg' ← Breg
Breg' ← Creg
Creg' ← Areg

if (trap level Areg is installed)
 take trap level Areg

Status Register:

The status register is saved and a new status register with set values is loaded.

Comments:

Secondary instruction.

Areg must be in the range *LowestException* to *HighestException*.

The exception type must be *ExceptionProcessType*.

See also: *eret*

Chapter 6.

enqueue

enqueue a process

Code: 23 F7**Description:** Add the process indicated by the process descriptor in **Breg** to the queue structure in **Areg**. The queue control block is defined in Chapter 7.**Definition:**

```

if (word[Areg @ q.FPtrLoc] = NotProcess)           -- queue is empty
{
    word'[Areg @ q.FPtrLoc]    ←  Breg
    word'[Areg @ q.BPtrLoc]    ←  Breg
}
else
{
    word'[word[Areg @ q.BPtrLoc] @ pw.Link] ←  Breg
    word'[Areg @ q.BPtrLoc]    ←  Breg
}

Areg' ←  undefined
Breg' ←  undefined
Creg' ←  undefined

```

Status Register:

No effect

Comments:

Secondary instruction.

This instruction may require 4 memory accesses. If the queue structure is in off-chip memory then interrupt latency may be affected.

Areg must be word aligned (i.e. divisible by 4).**See also:** *dequeue, stop*
Chapter 7.

eqc n

equals constant

Code: Function C

Description: Compare **Areg** to a constant.

Definition:

```
if (Areg = n)
    Areg' ← true
else
    Areg' ← false
```

Status Register:

No effect

Comments:

Primary instruction.

See also: *cj gt gtu*
section 4.6.

eret

exception return

Code: 23 F2**Description:** Return from an interrupt or trap.**Definition:**

if (Status_{trap_mode} = clear)
 signal interrupt return to interrupt controller

Areg' ← word[Wptr @ ex.InterptdAreg]
 Breg' ← word[Wptr @ ex.InterptdBreg]
 Creg' ← word[Wptr @ ex.InterptdCreg]
 Wptr' ← word[Wptr @ ex.InterptdWptr]
 lptr' ← word[Wptr @ ex.Interptdlptr]
 Tdesc' ← word[Wptr @ ex.InterptdTdesc]
 Status' ← word[Wptr @ ex.InterptdStatus]

Status Register:

Saved status register is restored.

Comments:

Secondary instruction.

The interrupted **Wptr** must be word aligned.

See also: *ecall*

Chapter 6.

fcall n

function call

Code: Function 9

Description: Call subroutine at the specified byte offset.

Definition:

$\text{word}'[\text{Wptr} @ 0] \leftarrow \text{lptr}$

$\text{lptr}' \leftarrow \text{next instruction} + n$

Status Register:

No effect

Comments:

Primary instruction.

See also: *ajw jab*

section 4.4.

gajw

general adjust workspace

Code: 23 FB**Description:** Set the workspace pointer to the word aligned address in **Areg**, saving the previous value in **Areg**.**Definition:**
$$\begin{aligned} \text{Wptr}' &\leftarrow \text{Areg} \\ \text{Areg}' &\leftarrow \text{Wptr} \end{aligned}$$
Status Register:

No effect

Comments:

Secondary instruction.

Areg should be word aligned (i.e. divisible by 4).**See also:** *ajw fcall*
section 4.4.

gt

greater than

Code: 21 FB**Description:** Compare the top two elements of the stack, returning *true* if **Breg** is greater than **Areg**.**Definition:**

```
if (Breg > Areg)
    Areg' ← true
else
    Areg' ← false
```

```
Breg' ← Creg
Creg' ← Areg
```

Status Register:

No effect

Comments:

Secondary instruction.

See also: *eqc gtu order*
section 4.6.

gtu

greater than unsigned

Code: 21 FC

Description: Compare the top two elements of the stack, treating both as unsigned integers, returning *true* if **Breg** is greater than **Areg**.

Definition:

if ($Breg_{\text{unsigned}} > Areg_{\text{unsigned}}$)

$Areg' \leftarrow true$

else

$Areg' \leftarrow false$

$Breg' \leftarrow Creg$

$Creg' \leftarrow Areg$

Status Register:

No effect

Comments:

Secondary instruction.

See also: *eqc gt orderu*
section 4.6.

Code: 23 FD

Description: Set, clear and test bits of the IO register. The bits set in the least significant half word of **Breg** are cleared in the IO register and then the bits set in the least significant half word of **Areg** are set in the IO register. The initial IO register is copied into the **Areg**.

Definition:

$$\text{IOreg}' \leftarrow (\text{IOreg} \wedge (\sim \text{Breg} \vee \#FFFF0000)) \vee (\text{Areg} \wedge \#FFFF)$$

$$\text{Areg}' \leftarrow \text{IOreg}$$

$$\text{Breg}' \leftarrow \text{Creg}$$

$$\text{Creg}' \leftarrow \text{Areg}$$

Status Register:

No effect

Comments:

Secondary instruction.

Some bits of the IO register may be reserved in some variants. See the datasheet for the variant.

See also: *bitmask*
section 4.11.

***j*n**

jump

Code: Function 0**Description:** Unconditional relative jump. The destination of the jump is expressed as a byte offset from the first byte after the current instruction.**Definition:**
$$Ip_{tr}' \leftarrow next\ instruction + n$$
Status Register:

No effect.

Comments:

Primary instruction.

See also: *cj jab*
section 4.6.

jab

jump absolute

Code: FD

Description: Jump to the address in **Areg**, saving the previous address in **Creg**. This instruction can be used for function and procedure returns and to jump to a run-time computed location.

Definition:

$\text{Iptr}' \leftarrow \text{Areg}$

$\text{Areg}' \leftarrow \text{Breg}$

$\text{Breg}' \leftarrow \text{Creg}$

$\text{Creg}' \leftarrow \text{Iptr}$

Status Register:

No effect

Comments:

Secondary instruction.

See also: *cj j*

section 4.4.

lbinc

load byte and increment

Code: 22 FA

Description: Load the unsigned byte addressed by **Areg** into **Areg** and increment the address by one byte.

Definition:

$$\begin{aligned} \text{Areg}'_{0..7} &\leftarrow \text{byte}[\text{Areg}] \\ \text{Areg}'_{8..\text{BitsPerWord}-1} &\leftarrow 0 \\ \text{Creg}' &\leftarrow \text{Areg} + 1 \end{aligned}$$

Status Register:

No effect

Comments:

Secondary instruction.

See also: *ldl ldnl lsinc lwinc sbinc*
section 4.2.

ldc n

load constant

Code: Function 4

Description: Load constant into **Areg**.

Definition:

$Areg' \leftarrow n$

$Breg' \leftarrow Areg$

$Creg' \leftarrow Breg$

Status Register:

No effect

Comments:

Primary instruction.

See also: *adc ldl*
section 4.2.

ldl n

load local

Code: Function 7**Description:** Load into **Areg** the local variable at the specified word offset in workspace. *ldl* cannot read from addresses reserved for peripheral registers.**Definition:**
$$\text{Areg}' \leftarrow \text{word}[\text{Wptr} @ n]$$
$$\text{Breg}' \leftarrow \text{Areg}$$
$$\text{Creg}' \leftarrow \text{Breg}$$
Status Register:

No effect

Comments:

Primary instruction.

Areg must be word aligned (i.e. divisible by 4).**See also:** *ldnl stl*
section 4.2.

ldlp n

load local pointer

Code: Function 1

Description: Load into **Areg** the address of the local variable at the specified offset in workspace.

Definition:

Areg' ← Wptr @ n

Breg' ← Areg

Creg' ← Breg

Status Register:

No effect

Comments:

Primary instruction.

See also: *ldl ldnlp*
section 4.5.

ldnl n

load non-local

Code: Function 3**Description:** Load into **Areg** the non-local variable at the specified word offset from **Areg**.**Definition:**
$$\text{Areg}' \leftarrow \text{word}[\text{Areg} @ n]$$
Status Register:

No effect

Comments:

Primary instruction.

Areg must be word aligned (i.e. divisible by 4).**See also:** *ldl ldnlp stnl*
section 4.2.

ldnlp n

load non-local pointer

Code: Function 5

Description: Load into **Areg** the address at the specified word offset from the address in **Areg**.

Definition:

$\text{Areg}' \leftarrow \text{Areg} @ n$

Status Register:

No effect

Comments:

Primary instruction.

See also: *ldlp ldnl wsub*
section 4.5.

ldpi

load pointer to instruction

Code: 23 FA

Description: Load into **Areg** an absolute address calculated from an offset relative to the current instruction pointer. **Areg** contains a byte offset which is added to the address of the first byte following this instruction.

Definition:

$Areg' \leftarrow next\ instruction + Areg$

Status Register:

No effect

Comments:

Secondary instruction.

See also: *jab*
section 4.5.

ldprodid

load product identity

Code: 23 FC

Description: Load a value indicating the product identity into Areg. Each product in the ST20 family has a unique product identity code.

Definition:

$Areg' \leftarrow ProductId$

$Breg' \leftarrow Areg$

$Creg' \leftarrow Breg$

Status Register:

No effect

Comments:

Secondary instruction.

Different ST20 products may use the same processor type, but return different product identity codes. However a product identity code uniquely defines the processor type used in that product. For specific product identities in the ST20 family refer to SGS-THOMSON.

See also:

section 4.2.

ltdesc

load task descriptor

Code: 23 F9

Description: Load the task descriptor of the current task.

Definition:

Areg' ← Tdesc

Breg' ← Areg

Creg' ← Breg

Status Register:

No effect

Comments:

Secondary instruction.

See also: *ldlp ldpi*

Chapter 6.

lsinc

load sixteen and increment

Code: 22 FC

Description: Load the unsigned 16-bit object addressed by **Areg** into **Areg** and increment the address by two bytes.

Definition:

$Areg'_{0..15} \leftarrow sixteen[Areg]$

$Areg'_{16..BitsPerWord-1} \leftarrow 0$

$Creg' \leftarrow Areg + 2$

Status Register:

No effect

Comments:

Secondary instruction.

Areg must be half-word aligned (i.e. divisible by 2).

See also: *lbinc ldl lsxinc lwinc ssinc*
section 4.2.

lsxinc load sixteen sign extended and increment

Code: 22 FD

Description: Load the signed 16-bit object addressed by **Areg** into **Areg**, sign extend to a word, and increment the address by two bytes.

Definition:

$$\begin{aligned} \text{Areg}'_{0..15} &\leftarrow \text{sixteen}[\text{Areg}] \\ \text{Areg}'_{16..\text{BitsPerWord}-1} &\leftarrow \text{sixteen}[\text{Areg}]_{15} \\ \text{Creg}' &\leftarrow \text{Areg} + 2 \end{aligned}$$

Status Register:

No effect

Comments:

Secondary instruction.

Areg must be half-word aligned (i.e. divisible by 2).

See also: *ldl lsinc lwinc ssinc xsword*
section 4.2.

lwinc

load word and increment

Code: 22 FF

Description: Load a variable from the address given in **Areg** and increment the address by one word. This instruction may be used with *stinc* as a step in a block move, and **Breg** is not modified so that it can be used to hold the store address.

Definition:

$Areg' \leftarrow word[Areg]$
 $Creg' \leftarrow Areg @ 1$

Status Register:

No effect

Comments:

Secondary instruction.

Areg must be word aligned (i.e. divisible by 4).

See also: *lbinc lsinc swinc*
section 4.2.

mac

multiply accumulate

Code: 21 F2

Description: Multiply **Areg** and **Breg**, and add in **Creg**. The most significant word of the result is placed in **Breg** and the least significant in **Areg**. This can be used in forming the double length product of **Areg** and **Breg**, with **Creg** as carry in, treating the **Areg** and **Breg** values as signed but **Creg** as unsigned.

Definition:

Areg' \leftarrow low_word (accum64)

Breg' \leftarrow high_word (accum64)

Creg' \leftarrow Areg

where $\text{accum64} = (\text{Breg} \times \text{Areg}) + \text{Creg}_{\text{unsigned}}$
 – the value of accum64 is calculated as a 64-bit value

Status Register:

No effect

Comments:

Secondary instruction.

See also: *ldiv mul umac*

Chapter 5.

mul

multiply

Code: F6

Description: Multiply **Areg** by **Breg**, with checking for overflow but no carry.

Definition:

```
Areg' ← low_word (prod)

if (prod > MostPos)
{
    if (Status'underflow ≠ set )
        Status'overflow ← set
}
else if (prod < MostNeg)
{
    if (Status'overflow ≠ set )
        Status'underflow ← set
}
```

```
Breg' ← Creg
Creg' ← Areg
```

where $\text{prod} = \text{Areg} \times \text{Breg}$
– *the value of prod is calculated to unlimited precision*

Status Register:

Overflow or underflow bit may be set.

Comments:

Secondary instruction.

See also: *add mac*
section 4.4.

nop

no operation

Code: 23 FF

Description: Perform no operation.

Definition:

no change

Status Register:

No effect

Comments:

Secondary instruction.

not

bitwise not

Code: F8

Description: Complement bits in **Areg**.

Definition:

$$\text{Areg}' \leftarrow \sim\text{Areg}$$

Status Register:

No effect

Comments:

Secondary instruction.

See also: *and or xor*
section 4.8.

or**or****Code:** FA**Description:** Bitwise OR of **Areg** and **Breg**.**Definition:**
$$\text{Areg}' \leftarrow \text{Breg} \vee \text{Areg}$$
$$\text{Breg}' \leftarrow \text{Creg}$$
$$\text{Creg}' \leftarrow \text{Areg}$$
Status Register:

No effect

Comments:

Secondary instruction.

See also: *and not xor*
section 4.8.

order

order

Code: 21 FD

Description: Order **Areg** and **Breg** into signed ascending order, so that **Breg** is greater than or equal to **Areg**. This can be used to implement maximum, minimum, absolute value and saturation at less than 32 bits.

Definition:

```
if (Areg > Breg)
{
    Areg' ← Breg
    Breg' ← Areg
}
```

Status Register:

No effect

Comments:

Secondary instruction.

See also: *gt orderu rev*
section 4.6.

orderu

unsigned order

Code: 21 FE

Description: Order **Areg** and **Breg** into unsigned ascending order, so that **Breg** is greater than or equal to **Areg**. This can be used to implement unsigned maximum and minimum.

Definition:

```
if (Aregunsigned > Bregunsigned)
{
    Areg' ← Breg
    Breg' ← Areg
}
```

Status Register:

No effect

Comments:

Secondary instruction.

See also: *gtu order rev*
section 4.6.

rev

reverse

Code: F0**Description:** Swap the values in **Areg** and **Breg**.**Definition:**
$$\begin{array}{lcl} \text{Areg}' & \leftarrow & \text{Breg} \\ \text{Breg}' & \leftarrow & \text{Areg} \end{array}$$
Status Register:

No effect

Comments:

Secondary instruction.

See also: *dup order orderu rot*
section 4.8.

rmw

memory read modify write

Code: 22 F9**Description:** Clear and set bits of the memory word at address **Areg** using the set mask in **Breg** and clear mask in **Creg**.**Definition:**

$$\begin{aligned} \text{word}'[\text{Areg}] &\leftarrow (\text{word}[\text{Areg}] \wedge \sim \text{Creg}) \vee \text{Breg} \\ \text{Areg}' &\leftarrow \text{word}[\text{Areg}] \end{aligned}$$

$$\begin{aligned} \text{Breg}' &\leftarrow \text{Areg} \\ \text{Creg}' &\leftarrow \text{Breg} \end{aligned}$$

Status Register:

No effect

Comments:

Secondary instruction.

Areg must be word aligned (i.e. divisible by 4).**See also:** *bitmask*
section 4.8.

rot

rotate stack

Code: F2

Definition: Rotate the evaluation stack upwards.

Definition:

$$\begin{aligned} \text{Areg}' &\leftarrow \text{Breg} \\ \text{Breg}' &\leftarrow \text{Creg} \\ \text{Creg}' &\leftarrow \text{Areg} \end{aligned}$$

Status Register:

No effect

Comments:

Secondary instruction.

See also: *arot dup rev*
section 4.8.

run

run process

Code: 23 F3

Description: Spawn a new process. Add the new process to the back of the process queue defined by the control block at *SchedulerQptr*. The task descriptor of the process is in **Areg**; this identifies the process descriptor block. The instruction pointer and work space pointer must have been written in the process descriptor block of the process at offsets **pw.Iptr** and **pw.Wptr** words.

Definition:

```

    if (run trap handler installed)
        take run trap
    else
    {
        if (word [SchedulerQptr @ q.FPtrLoc] = NotProcess)    -- queue is empty
        {
            word'[SchedulerQptr @ q.FPtrLoc] ← Areg
            word'[SchedulerQptr @ q.BPtrLoc] ← Areg
        }
        else
        {
            word'[ word [SchedulerQptr @ q.BPtrLoc] @ pw.Link] ← Areg
            word'[SchedulerQptr @ q.BPtrLoc] ← Areg
        }

        Areg' ← undefined
        Breg' ← undefined
        Creg' ← undefined
    }

```

Status Register:

No effect

Comments:

Secondary instruction.

Areg must be word aligned (i.e. divisible by 4).

This instruction may require up to three on-chip memory accesses and one off-chip. This may affect interrupt latency.

See also: *dequeue enqueue signal stop*
Chapter 7.

saturate

saturate arithmetic result

Code: 21 FA

Description: Saturate the value in **Areg** to *MostPos* or *MostNeg* if overflow or underflow respectively has occurred, and clear the overflow or underflow.

Definition:

```
if (Statusoverflow)  
    Areg' ← MostPos  
else if (Statusunderflow)  
    Areg' ← MostNeg
```

```
Status'overflow ← clear  
Status'underflow ← clear
```

Status Register:

Overflow and underflow bits are cleared.

Comments:

Secondary instruction.

See also: *adc add mul smul sub*
section 4.4.

sbinc

store byte and increment

Code: 22 FB

Description: Store the least significant byte of **Breg** into the byte of memory addressed by **Areg** and increment the address by one byte.

Definition:

$\text{byte}'[\text{Areg}] \leftarrow \text{Breg}_{0..7}$

$\text{Areg}' \leftarrow \text{Creg}$

$\text{Breg}' \leftarrow \text{Areg} + 1$

$\text{Creg}' \leftarrow \text{Breg}$

Status Register:

No effect

Comments:

Secondary instruction.

See also: *lbinc ssinc swinc stnl*
section 4.2.

shl

shift left

Code: FB

Description: Logical shift of **Breg** left by **Areg** bits, filling with zero bits. If the initial **Areg** is not between 0 and 31 inclusive then the result is undefined.

Definition:

$\text{Areg}' \leftarrow \text{Breg} \ll \text{Areg}$

$\text{Breg}' \leftarrow \text{Creg}$

$\text{Creg}' \leftarrow \text{Breg}$

Status Register:

No effect

Comments:

Secondary instruction.

Areg must be in the range 0..31.

See also: *shr*

section 4.9.

shr

shift right

Code: FC

Description: Logical shift of **Breg** right by **Areg** places, filling with zero bits. If the initial **Areg** is not between 0 and 31 inclusive then the result is undefined.

Definition:
$$\text{Areg}' \leftarrow \text{Breg} \gg \text{Areg}$$
$$\text{Breg}' \leftarrow \text{Creg}$$
$$\text{Creg}' \leftarrow \text{Breg}$$
Status Register:

No effect

Comments:

Secondary instruction.

Areg must be in the range 0..31.

See also: *ashr shl*
section 4.9.

Code: 23 F5

Description: Perform the first part of a semaphore signal (or V) on the semaphore pointed to by **Areg**. If no process is waiting then the unsigned count is incremented. If processes are waiting then the first process is removed from the queue and its identifier placed in **Breg** ready to be run. The instruction returns a boolean value in **Areg** stating whether a run process is required.

This instruction should be used in the following sequence: *ld semptr; signal; cj continue; run; continue*:

Definition:

```
if (frontptr = NotProcess)                -- queue empty
{
    word'[Areg @ s.Count] ← word[Areg @ s.Count] + 1
    Areg' ← false
    Breg' ← undefined
}
else                                       -- queue not empty
{
    if (frontptr = backptr)                -- one process on queue
        word'[Areg @ s.Front] ← NotProcess
    else                                   -- many processes on queue
        word'[Areg @ s.Front] ← word[ frontptr @ pw.Link ]

    Areg' ← true
    Breg' ← frontptr
}

Creg' ← undefined

where frontptr = word[Areg @ s.Front]
      backptr = word[Areg @ s.Back]
```

Status Register:

No effect

Comments:

Secondary instruction.

The increment of the count is unchecked.

This instruction may require 4 accesses to memory. The semaphore structure should be placed in on-chip memory if interrupt latency is critical.

See also: *run wait*

Chapter 7.

smacinit initialize short multiply accumulate loop

Code: 21 F5

Description: Set the loop count, buffer size, scaling and data format for the *smacloop* instruction.

Definition:

$$\begin{aligned}\text{Status}'_{\text{mac_count}} &\leftarrow \text{Areg} \wedge \text{\#FF} \\ \text{Status}'_{\text{mac_buffer}} &\leftarrow (\text{Areg} \gg 8) \wedge \text{\#7} \\ \text{Status}'_{\text{mac_scale}} &\leftarrow (\text{Areg} \gg 11) \wedge \text{\#3} \\ \text{Status}'_{\text{mac_mode}} &\leftarrow (\text{Areg} \gg 13) \wedge \text{\#1}\end{aligned}$$

$$\begin{aligned}\text{Areg}' &\leftarrow \text{Breg} \\ \text{Breg}' &\leftarrow \text{Creg} \\ \text{Creg}' &\leftarrow 0\end{aligned}$$

Status Register:

No effect.

Comments:

Secondary instruction.

See also: *smacloop*
Chapter 5.

smacloop

short multiply accumulate loop

Code: 21 F6

Description: Multiply pairs of signed 16-bit values and accumulate the products. On entry **Areg** and **Breg** contain the addresses of arrays X and Y of 16-bit values and **Creg** holds the initial 32-bit accumulator. On exit the accumulator is in **Areg**, while **Breg** and **Creg** contain the next addresses in the X and Y arrays respectively. The X values can be in a circular buffer whose address is exactly divisible by the size. The loop count, buffer size, mode and scaling codes, held in the status register, may be set by the *smacinit* instruction.

Warning: This instruction is not interruptible. If interrupt latency is critical then a similar result may be achieved by executing the instruction many times with small counts. The result may not be the same, since rounding occurs at the end of the instruction from the internal 48-bit accumulator to a 32-bit result.

Definition:

```
if (Statusoverflow)
    Areg' ← max
else if (Statusunderflow)
    Areg' ← min
else if (overflow)
{
    Status'overflow ← set
    Areg' ← max
}
else if (underflow)
{
    Status'underflow ← set
    Areg' ← min
}
else
{
    Areg' ← acc >>arith shift1
}
```

```
Breg' ← xaddress (Statusmac_count)
Creg' ← Breg + (2 × Statusmac_count)
```

where

$$\begin{aligned} \text{acc} = & (\text{Creg} \ll \text{shift1}) + (1 \ll (\text{shift1}-1)) \\ & + \sum_{i=0..n-1} [\text{sixteen}[\text{xaddress}(i)] \times (\text{sixteen}[\text{Breg} + 2.i] \ll \text{shift2})] \\ & - \text{the value of acc is calculated to 48-bit precision} \end{aligned}$$

if (Status_{mac_count} = 0) n=256


```

else                                n = Statusmac_count

if (Statusmac_buffer ≠ 0)
{
    buff_size = 4 << Statusmac_buffer
    xaddress (i) = Areg - (Areg rem buff_size) + ((Areg + 2.i) rem buff_size)
}
else    xaddress(i) = Areg + (2 × i)

if (Statusmac_mode = LongMode)
{
    shift1 = 7
    max = MostPos
    min = MostNeg
}
else
{
    shift1 = 23
    max = #7FFF
    min = #8000
}

if (Statusmac_scale = 3)  shift2 = 9
else                    shift2 = 4 × Statusmac_scale

```

Status Register:

Overflow or underflow may be set.

Comments:

Secondary instruction.

Areg must be half-word aligned. **Breg** must be word aligned.

See also: *biquad mac mul smacinit*

Chapter 5.

smul

short multiply

Code: 21 F4

Description: Multiply **Areg** by **Breg**, treated as sixteen bit signed integers.

Definition:

$\text{Areg}' \leftarrow (\text{int16})\text{Areg} \times (\text{int16})\text{Breg}$

$\text{Breg}' \leftarrow \text{Creg}$

$\text{Creg}' \leftarrow \text{Areg}$

Status Register:

No effect

Comments:

Secondary instruction.

Areg and **Breg** must be in the range of 16-bit signed integers.

See also: *mul smac*
section 4.4.

ssinc

store sixteen and increment

Code: 22 FE

Description: Store the least significant 16 bits of **Breg** into the half word of memory addressed by **Areg** and increment the address by two bytes.

Definition:

sixteen'[Areg] \leftarrow Breg_{0..15}

Areg' \leftarrow Creg

Breg' \leftarrow Areg + 2

Creg' \leftarrow Breg

Status Register:

No effect

Comments:

Secondary instruction.

Areg must be half-word aligned (i.e. divisible by 2).

See also: *lsinc lxxinc sbinc stl stl swinc*
section 4.2.

statusclr

clear bits in status register

Code: 22 F7

Description: Clear the bits of the status register which are set in the **Areg**. The initial status register value is returned in the **Areg**.

Definition:

$Areg' \leftarrow Status$
 $Status' \leftarrow Status \wedge \sim Areg$

Status Register:

The bits of **Areg** which are set will be cleared in the Status Register.

Comments:

Secondary instruction.

See also: *statusset statustst*
section 4.12.

statusset

set bits in status register

Code: 22 F6

Description: Set the bits of the Status Register which are set in **Areg**. The initial Status Register value is returned in the **Areg**.

Definition:

$Areg' \leftarrow Status$
 $Status' \leftarrow Status \vee Areg$

Status Register:

The bits of **Areg** which are set will be set in the Status Register.

Comments:

Secondary instruction.

See also: *statusclr statustst*
section 4.12.

statustst

test status register

Code: 22 F8

Description: Test the bits of the Status Register which are set in the **Areg**.

Definition:

$Areg' \leftarrow Status \wedge Areg$

Status Register:

No effect

Comments:

Secondary instruction.

See also: *bitmask statusclr statusset*
section 4.12.

stl n

store local

Code: Function D

Description: Store the contents of **Areg** into the local variable at the specified word offset in workspace. This instruction cannot write to addresses reserved for peripheral registers.

Definition:
$$\text{word}'[\text{Wptr @ n}] \leftarrow \text{Areg}$$
$$\text{Areg}' \leftarrow \text{Breg}$$
$$\text{Breg}' \leftarrow \text{Creg}$$
$$\text{Creg}' \leftarrow \text{Areg}$$
Status Register:

No effect

Comments:

Primary instruction.

See also: *ldl sbinc ssinc stnl*
section 4.2.

stnl n

store non-local

Code: Function E

Description: Store the contents of **Breg** into the non-local variable at the specified word offset from **Areg**.

Definition:

$\text{word}[\text{Areg} @ n] \leftarrow \text{Breg}$

$\text{Areg}' \leftarrow \text{Creg}$

$\text{Breg}' \leftarrow \text{Areg}$

$\text{Creg}' \leftarrow \text{Breg}$

Status Register:

No effect

Comments:

Primary instruction.

Areg must be word aligned (i.e. divisible by 4).

See also: *ldnl sbinc ssinc swinc stl*
section 4.2.

stop

stop process

Code: 23 F4

Description: Terminate the current process, saving the current **Ip**tr and **Wp**tr for later use, and start the next process from the scheduling queue. If the scheduling queue is empty then the CPU will become idle. This instruction is implemented in two stages to protect interrupt latency. The first stage deschedules the current process and sets the **start_next_task** status bit. The second stage clears the **start_next_task** status bit and starts the next process. An interrupt may occur between the two stages.

Definition:

```
if (stop trap handler installed)
    take stop trap
```

```
else
```

```
{
```

```
    word'[Tdesc @ pw.Iptr]    ← next instruction
    word'[Tdesc @ pw.Wptr]    ← Wptr
    Status'local_interrupt_enable ← set
    Status'overflow           ← clear
    Status'underflow          ← clear
    Status'carry               ← clear
    Status'interrupt_mode      ← clear
    Status'trap_mode           ← clear
    Status'start_next_task     ← clear
    Status'timeslice_count     ← MaxTimesliceCount
```

```
-- start next process on scheduling queue
```

```
if (frontptr = NotProcess)          -- queue is empty, so become idle
```

```
{
```

```
    Status'sleep              ← set
    Status'user_mode           ← clear
```

```
    if (idle trap handler installed)
        take idle trap
```

```
    else
        set idle
```

```
}
```

```
else
```

```
-- processes waiting, so start next process
```

```
{
```

```
    Status'sleep              ← clear
    Status'user_mode           ← set
    Tdesc'                     ← frontptr
    Wptr'                       ← word[frontptr @ pw.Wptr]
```

```

    lptr'          ← word [frontptr @ pw.lptr]
    if (frontptr = backptr)      --one process on queue, so make it empty
        word'[SchedulerQptr @ q.FPtrLoc] ← NotProcess
    else                        --many processes, so remove first
        word'[SchedulerQptr @ q.FPtrLoc] ← word[frontptr @ pw.Link]
}

Areg' ← undefined
Breg' ← undefined
Creg' ← undefined
}

where frontptr = word[SchedulerQptr @ q.FPtrLoc]
        backptr = word[SchedulerQptr @ q.BPtrLoc]

```

Status Register:

Global interrupt enable and timeslice enable are preserved.

Start_next_task is set when the current process is descheduled and cleared when the new process is started. An interrupt may occur between these two stages.

Other bits are set to the values listed above.

Comments:

Secondary instruction.

Deschedules the current process.

This instruction must not be used in an exception handler.

See also: *dequeue enqueue run wait*
Chapter 7.

sub

subtract

Code: F5**Description:** Subtract **Areg** from **Breg**, with checking for overflow.**Definition:**

```

if (diff > MostPos)
{
    Areg'      ← diff – 2BitsPerWord
    if (Status'underflow ≠ set )
        Status'overflow ← set
}
else if (diff < MostNeg)
{
    Areg'      ← diff + 2BitsPerWord
    if (Status'overflow ≠ set )
        Status'underflow ← set
}
else
{
    Areg'      ← diff
}

Breg ← Creg
Creg ← Areg

where    diff = Breg – Areg
           – the value of diff is calculated to unlimited precision

```

Status Register:

Overflow or underflow bit may be set.

Comments:

Secondary instruction.

See also: *add subc*
section 4.4.

subc

subtract with carry

Code: 21 F1

Description: Subtract **Areg** from **Breg**, unsigned with carry propagation. This instruction is provided for long arithmetic; address calculations may be performed with *add*, *sub* and *adc* without affecting the carry flag.

Definition:

```
if ( diff ≥ 0 )
{
    Areg'unsigned    ← diff
    Status'carry    ← clear
}
else
{
    Areg'unsigned    ← diff + 2BitsPerWord
    Status'carry    ← set
}

Breg' ← Creg
Creg' ← Areg

where    diff = (Bregunsigned – Aregunsigned) - Statuscarry
           – the value of diff is calculated to unlimited precision
```

Status Register:

Carry bit is set or cleared.

Comments:

Secondary instruction.

See also: *addc sub*
section 4.4.

swap32

byte swap 32

Code: 23 FE**Description:** Reverse the order of the bytes within **Areg**.**Definition:**

$$\text{Areg}' \leftarrow (\text{Areg}_{0..7} \ll 24) \vee (\text{Areg}_{8..15} \ll 16) \vee (\text{Areg}_{16..23} \ll 8) \vee \text{Areg}_{24..31}$$

Status Register:

No effect

Comments:

Secondary instruction.

See also:

section 4.9.

swinc

store word and increment

Code: 23 F0

Description: Store the value from the **Breg** at the memory address given in the initial **Areg** and increment the address by one word. This instruction may be used with *lwinc* as a step in a block move. The value in **Creg** is copied into the **Areg**.

Definition:

word'[Areg] ← Breg

Areg' ← Creg

Breg' ← Areg @ 1

Creg' ← Breg

Status Register:

No effect

Comments:

Areg must be word aligned (i.e. divisible by 4).
Secondary instruction.

See also: *ldinc sbinc ssinc stnl*
section 4.2.

timeslice

timeslice

Code: FE

Description: Deschedules the current process and starts the next task if a timeslice is due and timeslicing is enabled. This instruction is implemented in two stages to protect interrupt latency. The first stage deschedules the current process (even if the queue is empty), adds it to the back of the scheduling queue and sets the **start_next_task** status bit. The second stage clears the **start_next_task** status bit and starts the next process from the scheduling queue. An interrupt may occur between the two stages.

Definition:

```

if ( Statustimeslicing enabled and Statustimeslice_count=0 )
{
    if (timeslice trap installed)
        take timeslice trap
    else
    {
        -- save state of current process
        word'[Tdesc @ pw.lptr]    ←  next instruction
        word'[Tdesc @ pw.Wptr]    ←  Wptr

        if (frontptr ≠ NotProcess)          --queue not empty
        {
            -- add current process to scheduling queue
            word'[backptr @ pw.Link]        ←  Tdesc
            word'[SchedulerQptr @ q.BPtrLoc] ←  Tdesc
            -- remove front process from queue
            word'[SchedulerQptr @ q.FPtrLoc] ←  word[frontptr @ pw.Link]
            -- start next process from scheduling queue
            Tdesc'          ←  frontptr
            Wptr'           ←  word [frontptr @ pw.Wptr]
            lptr'           ←  word [frontptr @ pw.lptr]
        }

        Status'local_interrupt_enable    ←  set
        Status'overflow                  ←  clear
        Status'undeflow                  ←  clear
        Status'carry                      ←  clear
        Status'user_mode                  ←  set
        Status'interrupt_mode             ←  clear
        Status'trap_mode                  ←  clear
        Status'sleep                      ←  clear
        Status'start_next_task            ←  clear
    }
}

```

```

        Status' timeslice_count      ←  MaxTimesliceCount

        Areg'  ←  undefined
        Breg'  ←  undefined
        Creg'  ←  undefined
    }
}

where frontptr = word[SchedulerQptr @ q.FPtrLoc]
        backptr  = word[SchedulerQptr @ q.BPtrLoc]

```

Status Register:

Global interrupt enable and timeslice enable preserved.

Start_next_task is set when the current process is descheduled and cleared when the new process is started. An interrupt may occur between these two stages.

Other bits are set to the values listed above.

Comments:

Secondary instruction.

This instruction must not be used in an exception handler.

See also: *stop run*

Chapter 7.

umac

unsigned multiply accumulate

Code: 21 F3

Description: Multiply Areg by Breg, adding in Creg, to form a double word result, treating the initial values as unsigned. This can be used in forming the double length product of **Areg** and **Breg**, with **Creg** as carry in.

Definition:

$$\text{Areg}'_{\text{unsigned}} \leftarrow \text{low_word}(\text{prod})$$

$$\text{Breg}'_{\text{unsigned}} \leftarrow \text{high_word}(\text{prod})$$

$$\text{Creg}' \leftarrow \text{Areg}$$

where $\text{prod} = (\text{Breg}_{\text{unsigned}} \times \text{Areg}_{\text{unsigned}}) + \text{Creg}_{\text{unsigned}}$
– the value of prod is calculated to unlimited precision

Status Register:

No effect

Comments:

Secondary instruction.

See also: *mac*

Chapter 5.

unsign

unsign argument

Code: 21 F9

Description: Remove the sign of the value in **Areg** and exclusive-or it with a sign flag in **Breg**.

Definition:

```
if (Areg ≥ 0)
{
    Areg' ← Breg
    Breg' ← Areg
}
else
{
    Areg' ← 1 - Breg
    Breg' ← - Areg
}
```

Status Register:

No effect

Comments:

Secondary instruction

To aid the implementation of signed division.

Breg must be 0 or 1.

See also: *divstep*
section 4.4.

wait

wait

Code: 23 F6

Description: Perform the first part of a semaphore wait (or P) on the semaphore pointed to by **Areg**. If the semaphore count is non-zero then the count is decremented and the process continues; otherwise the current process is added to the back of the semaphore list ready for the current process to be descheduled. The instruction returns a boolean value in **Areg** stating whether a process deschedule is required.

The instruction should be used in the following sequence:

ld semptr; wait; cj continue; stop; continue:

Definition:

```

if (word[Areg @ s.Count] ≠ 0)
{
    word'[Areg @ s.Count] ← word[Areg @ s.Count] – 1
    Areg' ← false
}
else
{
    if (frontptr = NotProcess)                -- queue empty so add to queue
    {
        word'[Areg @ s.Front] ← Tdesc
        word'[Areg @ s.Back] ← Tdesc
    }
    else                                     -- queue not empty so add to back of queue
    {
        word'[backptr @ pw.Link] ← Tdesc
        word'[Areg @ s.Back] ← Tdesc
    }
    Areg' ← true
}
Breg' ← undefined
Creg' ← undefined

where frontptr = word[Areg @ s.Front]
        backptr = word[Areg @ s.Back]

```

Status Register: No effect

Comments:

Secondary instruction.

Areg must be word-aligned.

The semaphore structure should be in fast memory if interrupt latency is critical.

See also: *signal stop*
Chapter 7.

wsub

word subscript

Code: F7

Description: Generate the address of the element which is indexed by **Breg** in the word array pointed to by **Areg**.

Definition:

$Areg' \leftarrow Areg @ Breg$

$Breg' \leftarrow Creg$

$Creg' \leftarrow Areg$

Status Register:

No effect

Comments:

Secondary instruction.

See also: *ldlp ldhlp*
section 4.5.

xbword

sign extend byte to word

Code: 22 F1

Description: Sign-extend the value in the least significant byte of **Areg** into a signed word.

Definition:

$Areg'_{0..7}$	\leftarrow	$Areg_{0..7}$
$Areg'_{8..BitsPerWord-1}$	\leftarrow	$Areg_7$

Status Register:

No effect

Comments:

Secondary instruction.

See also: *xsword*
section 4.4.6.

xor

exclusive or

Code: 21 F0

Description: Bitwise exclusive **or** of **Areg** and **Breg**.

Definition:

$\text{Areg}' \leftarrow \text{Breg} \otimes \text{Areg}$

$\text{Breg}' \leftarrow \text{Creg}$

$\text{Creg}' \leftarrow \text{Areg}$

Status Register:

No effect

Comments:

Secondary instruction.

See also: *and not or*
section 4.8.

xsword

sign extend sixteen to word

Code: 22 F2

Description: Sign extend the value in the least significant 16 bits of **Areg** to a signed word.

Definition:

$Areg'_{0..15}$	\leftarrow	$Areg_{0..15}$
$Areg'_{16..BitsPerWord-1}$	\leftarrow	$Areg_{15}$

Status Register:

No effect

Comments:

Secondary instruction.

See also: *xbword*
section 4.4.6.

Appendices

A Constants and data structures

This appendix gives a full listing of all the constants and data structures used in this document, with brief descriptions and values.

A.1 Status Register

Full name	Bit numbers	Meaning when set or meaning of value
mac_count	0 - 7	Multiply-accumulate number of steps.
mac_buffer	8 - 10	Multiply-accumulate data buffer size code.
mac_scale	11 - 12	Multiply-accumulate scaling code.
mac_mode	13	Multiply-accumulate accumulator format code.
global_interrupt_disable	14	Disable external interrupts until explicitly enabled.
local_interrupt_disable	15	Disable external interrupts until process descheduled.
overflow	16	An arithmetic operation gave a positive overflow.
underflow	17	An arithmetic operation gave a negative overflow.
carry	18	An arithmetic operation produced a carry.
user_mode	19	A user process is executing.
interrupt_mode	20	An interrupt handler is executing.
trap_mode	21	A trap handler is executing.
sleep	22	The processor is due to go to sleep.
reserved	23	Reserved.
start_next_task	24	The CPU must start executing a new process.
timeslice_enable	25	Timeslicing is enabled.
timeslice_count	26 - 31	Timeslice clock.

Table A.1 Status register bits (see section 3.3.2)

A.2 Exception levels

Exception level	Name	Description
0 - 255	User exception	User exception levels.
-1	el_breakpoint_trap	Breakpoint trap.
-2	el_illegal_instr_trap	Illegal op-code trap.
-3	el_idle_trap	CPU idle trap.
-4	el_schedule_exception_trap	Schedule user process as exception trap.
-5	el_run_trap	Run process trap.
-6	el_stop_trap	Stop process trap.
-7	el_timeslice_trap	Timeslice trap.

Table A.2 Exception levels (see Chapter 6)

A.3 Data structures

This section defines the data structures used by the ST20-C1 core.

Name	Word offset	Purpose
ex.HandlerIptra	7	Exception handler instruction pointer.
ex.InterptdStatus	6	Interrupted process status register.
ex.InterptdTdesc	5	Interrupted process task descriptor.
ex.InterptdIptra	4	Interrupted process instruction pointer.
ex.InterptdWptra	3	Interrupted process stack pointer.
ex.InterptdCreg	2	Interrupted process Creg.
ex.InterptdBreg	1	Interrupted process Breg.
ex.InterptdAreg	0	Interrupted process Areg.

Table A.3 Exception handler (see Chapter 6)

Name	Word offset	Purpose
pw.Iptra	2	The instruction pointer.
pw.Wptra	1	The instruction pointer.
pw.Link	0	The link to the next process in the list.

Table A.4 Process descriptor block (see section 3.3.4)

Name	Word offset	Purpose
q.BPtraLoc	1	The last process in the waiting process list.
q.FPtraLoc	0	The first process in the waiting process list.

Table A.5 Waiting process list data structure (see section 7.1)

Name	Word offset	Purpose
s.Back	2	The last process in the semaphore waiting list.
s.Front	1	The first process in the semaphore waiting list.
s.Count	0	The unsigned number of extra processes that the semaphore will allow to continue running on a <i>wait</i> request.

Table A.6 Semaphore data structure (see section 7.9)

A.4 Miscellaneous constants

This section lists the constants which are used in this manual.

Name	Value	Meaning
<i>BitsPerWord</i>	32	The number of bits in a word.
<i>BytesPerWord</i>	4	The number of bytes in a word.
<i>ExceptionBase</i>	#80000040	The base of the exception descriptor area.
<i>ExceptionProcessType</i>	#1	Bit 0 of exception vector table entry for an exception.
<i>false</i>	0	The boolean value 'false'.
<i>HighestException</i>	255	Highest permitted exception level.
<i>LongMode</i>	1	<i>smacloop</i> Q15 data format.
<i>LowestException</i>	-7	Lowest permitted exception level.
<i>MaxTimesliceCount</i>	63	Maximum value of timeslice count.
<i>MostNeg</i>	#80000000	The most negative integer value.
<i>MostPos</i>	#7FFFFFFF	The most positive signed integer value.
<i>NotProcess</i>	<i>MostNeg</i> #80000000	Used, wherever a process descriptor is expected, to indicate that there is no process.
<i>ProductId</i>	Depends on processor type.	A value used to identify the type and revision of processor, returned by the <i>Idprodid</i> instruction.
<i>SavedTaskDescriptor</i>	#800000008	Address where user process exception is saved when a schedule exception trap occurs.
<i>SchedulerQptr</i>	<i>MostNeg</i> #80000000	The address of the process list data structure for the scheduler list.
<i>ShortMode</i>	0	<i>smacloop</i> Q31 data format.
<i>true</i>	1	The boolean value 'true'.
<i>UserProcessType</i>	#0	Bit 0 of exception vector table entry for a user process.

Table A.7 Constants

B Instruction set summary

This appendix gives a full listing of all the instructions and instruction components, both in function code and alphabetical order. The 'Memory' column gives, in hexadecimal, the bytes that appear in memory. For primary instructions, this includes a data value, which is represented by *n*.

B.1 Function code order

B.1.1 Primary instructions

Code	Memory	Mnemonic	Name
0	0 <i>n</i>	<i>j n</i>	jump
1	1 <i>n</i>	<i>ldlp n</i>	load local pointer
2	2 <i>n</i>	<i>pfix n</i>	prefix
3	3 <i>n</i>	<i>ldnl n</i>	load non-local
4	4 <i>n</i>	<i>ldc n</i>	load constant
5	5 <i>n</i>	<i>ldnlp n</i>	load non-local pointer
6	6 <i>n</i>	<i>nfix n</i>	negative prefix
7	7 <i>n</i>	<i>ldl n</i>	load local
8	8 <i>n</i>	<i>adc n</i>	add constant
9	9 <i>n</i>	<i>fcall n</i>	function call
A	A <i>n</i>	<i>cj n</i>	conditional jump
B	B <i>n</i>	<i>ajw n</i>	adjust work space
C	C <i>n</i>	<i>eqc n</i>	equals constant
D	D <i>n</i>	<i>stl n</i>	store local
E	E <i>n</i>	<i>stnl n</i>	store non-local
F	F <i>n</i>	<i>opr n</i>	operate

B.1.2 Secondary instructions

Code	Memory	Mnemonic	Name
00	F0	<i>rev</i>	reverse
01	F1	<i>dup</i>	duplicate
02	F2	<i>rot</i>	rotate stack
03	F3	<i>arot</i>	anti-rotate stack
04	F4	<i>add</i>	add
05	F5	<i>sub</i>	subtract
06	F6	<i>mul</i>	multiply
07	F7	<i>wsb</i>	word subscript
08	F8	<i>not</i>	bitwise not
09	F9	<i>and</i>	and
0A	FA	<i>or</i>	or
0B	FB	<i>shl</i>	shift left
0C	FC	<i>shr</i>	shift right
0D	FD	<i>jab</i>	jump absolute
0E	FE	<i>timeslice</i>	timeslice
0F	FF	<i>breakpoint</i>	breakpoint

10	21 F0	<i>addc</i>	add with carry
11	21 F1	<i>subc</i>	subtract with carry
12	21 F2	<i>mac</i>	multiply accumulate
13	21 F3	<i>umac</i>	unsigned multiply accumulate
14	21 F4	<i>smul</i>	short multiply
15	21 F5	<i>smacinit</i>	initialize short multiply accumulate loop
16	21 F6	<i>smacloop</i>	short multiply accumulate loop
17	21 F7	<i>biquad</i>	biquad IIR filter step
18	21 F8	<i>divstep</i>	divide step
19	21 F9	<i>unsign</i>	unsign argument
1A	21 FA	<i>saturate</i>	saturate
1B	21 FB	<i>gt</i>	greater than
1C	21 FC	<i>gtu</i>	greater than unsigned
1D	21 FD	<i>order</i>	order
1E	21 FE	<i>orderu</i>	unsigned order
1F	21 FF	<i>ashr</i>	arithmetic shift right
20	22 F0	<i>xor</i>	exclusive or
21	22 F1	<i>xbword</i>	sign extend byte to word
22	22 F2	<i>xsword</i>	sign extend sixteen to word
23	22 F3	<i>bitld</i>	load bit
24	22 F4	<i>bitst</i>	store bit
25	22 F5	<i>bitmask</i>	create bit mask
26	22 F6	<i>statusset</i>	set bits in status register
27	22 F7	<i>statusclr</i>	clear bits in status register
28	22 F8	<i>statustst</i>	test status register
29	22 F9	<i>rmw</i>	read modify write
2A	22 FA	<i>lbinc</i>	load byte and increment
2B	22 FB	<i>sbinc</i>	store byte and increment
2C	22 FC	<i>lsinc</i>	load sixteen and increment
2D	22 FD	<i>lsxinc</i>	load sixteen sign extended and increment
2E	22 FE	<i>ssinc</i>	store sixteen and increment
2F	22 FF	<i>lwinc</i>	load word and increment
30	23 F0	<i>swinc</i>	store word and increment
31	23 F1	<i>ecall</i>	exception call
32	23 F2	<i>eret</i>	exception return
33	23 F3	<i>run</i>	run process
34	23 F4	<i>stop</i>	stop process
35	23 F5	<i>signal</i>	signal
36	23 F6	<i>wait</i>	wait
37	23 F7	<i>enqueue</i>	enqueue a process
38	23 F8	<i>dequeue</i>	dequeue a process
39	23 F9	<i>ldtdesc</i>	load task descriptor
3A	23 FA	<i>ldpi</i>	load pointer to instruction
3B	23 FB	<i>gajw</i>	general adjust workspace
3C	23 FC	<i>ldprodid</i>	load product identity
3D	23 FD	<i>io</i>	input/output
3E	23 FE	<i>swap32</i>	byte swap 32
3F	23 FF	<i>nop</i>	no operation

B.2 Alphabetical order

Mnemonic	Code	Memory	Name
<i>adc n</i>	primary 8	8n	add constant
<i>add</i>	secondary 04	F4	add
<i>addc</i>	secondary 10	21 F0	add with carry
<i>ajw n</i>	primary B	Bn	adjust work space
<i>and</i>	secondary 09	F9	and
<i>arot</i>	secondary 03	F3	anti-rotate stack
<i>ashr</i>	secondary 1F	21 FF	arithmetic shift right
<i>biquad</i>	secondary 17	21 F7	biquad IIR filter step
<i>bitld</i>	secondary 23	22 F3	load bit
<i>bitmask</i>	secondary 25	22 F5	create bit mask
<i>bitst</i>	secondary 24	22 F4	store bit
<i>breakpoint</i>	secondary 0F	FF	breakpoint
<i>cj n</i>	primary A	An	conditional jump
<i>dequeue</i>	secondary 38	23 F8	dequeue a process
<i>divstep</i>	secondary 18	21 F8	divide step
<i>dup</i>	secondary 01	F1	duplicate
<i>ecall</i>	secondary 31	23 F1	exception call
<i>enqueue</i>	secondary 37	23 F7	enqueue a process
<i>eqc n</i>	primary C	Cn	equals constant
<i>eret</i>	secondary 32	23 F2	exception return
<i>fcall n</i>	primary 9	9n	function call
<i>gajw</i>	secondary 3B	23 FB	general adjust workspace
<i>gt</i>	secondary 1B	21 FB	greater than
<i>gtu</i>	secondary 1C	21 FC	greater than unsigned
<i>io</i>	secondary 3D	23 FD	input/output
<i>j n</i>	primary 0	0n	jump
<i>jab</i>	secondary 0D	FD	jump absolute
<i>lbinc</i>	secondary 2A	22 FA	load byte and increment
<i>ldc n</i>	primary 4	4n	load constant
<i>ldl n</i>	primary 7	7n	load local
<i>ldlp n</i>	primary 1	1n	load local pointer
<i>ldnl n</i>	primary 3	3n	load non-local
<i>ldnlp n</i>	primary 5	5n	load non-local pointer
<i>ldpi</i>	secondary 3A	23 FA	load pointer to instruction
<i>ldprodid</i>	secondary 3C	23 FC	load product identity
<i>ldtdesc</i>	secondary 39	23 F9	load task descriptor
<i>lsinc</i>	secondary 2C	22 FC	load sixteen and increment
<i>lsxinc</i>	secondary 2D	22 FD	load sixteen sign extended and increment
<i>lwinc</i>	secondary 2F	22 FF	load word and increment
<i>mac</i>	secondary 12	21 F2	multiply accumulate
<i>mul</i>	secondary 06	F6	multiply
<i>nfix n</i>	primary 4	4n	negative prefix
<i>nop</i>	secondary 3F	23 FF	no operation
<i>not</i>	secondary 08	F8	bitwise not
<i>opr n</i>	primary F	Fn	operate
<i>or</i>	secondary 0A	FA	or
<i>order</i>	secondary 1D	21 FD	order
<i>orderu</i>	secondary 1E	21 FE	unsigned order

<i>prefix n</i>	primary 2	2n	prefix
<i>rev</i>	secondary 00	F0	reverse
<i>rmw</i>	secondary 29	22 F9	read modify write
<i>rot</i>	secondary 02	F2	rotate stack
<i>run</i>	secondary 33	23 F3	run process
<i>saturate</i>	secondary 1A	21 FA	saturate
<i>sbinc</i>	secondary 2B	22 FB	store byte and increment
<i>shl</i>	secondary 0B	FB	shift left
<i>shr</i>	secondary 0C	FC	shift right
<i>signal</i>	secondary 35	23 F5	signal
<i>smacinit</i>	secondary 15	21 F5	initialize short multiply accumulate loop
<i>smacloop</i>	secondary 16	21 F6	short multiply accumulate loop
<i>smul</i>	secondary 14	21 F4	short multiply
<i>ssinc</i>	secondary 2E	22 FE	store sixteen and increment
<i>statusclr</i>	secondary 27	22 F7	clear bits in status register
<i>statusset</i>	secondary 26	22 F6	set bits in status register
<i>statustst</i>	secondary 28	22 F8	test status register
<i>stl n</i>	primary D	Dn	store local
<i>stnl n</i>	primary E	En	store non-local
<i>stop</i>	secondary 34	23 F4	stop process
<i>sub</i>	secondary 05	F5	subtract
<i>subc</i>	secondary 11	21 F1	subtract with carry
<i>swap32</i>	secondary 3E	23 FE	byte swap 32
<i>swinc</i>	secondary 30	23 F0	store word and increment
<i>timeslice</i>	secondary 0E	FE	timeslice
<i>umac</i>	secondary 13	21 F3	unsigned multiply accumulate
<i>unsign</i>	secondary 19	21 F9	unsign argument
<i>wait</i>	secondary 36	23 F6	wait
<i>wsub</i>	secondary 07	F7	word subscript
<i>xbword</i>	secondary 21	22 F1	sign extend byte to word
<i>xor</i>	secondary 20	22 F0	exclusive or
<i>xsword</i>	secondary 22	22 F2	sign extend sixteen to word

C Compiling for the ST20-C1

C.1 Generating prefix sequences

Prefixing is intended to be performed by a compiler or assembler. Prefixing by hand is not advised.

Normally a value can be loaded into the instruction data value by a variety of different prefix sequences. It is important to use the shortest possible sequence as this enhances both code compaction and execution speed. The best method of optimizing object code so as to minimize the number of prefix instructions needed is shown below.

C.1.1 Prefixing a constant

The algorithm to generate a constant instruction data value e for a function op is described by the following recursive function.

$$\begin{aligned} \text{prefix}(op, e) = & \text{if } (e < 16 \text{ AND } e \geq 0) \\ & \quad op(e) \\ & \text{else if } (e \geq 16) \\ & \quad \{ \text{prefix}(pfix, e \gg 4); op(e \wedge \#F) \} \\ & \text{else if } (e < 0) \\ & \quad \{ \text{prefix}(nfix, (\sim e) \gg 4); op(e \wedge \#F) \} \end{aligned}$$

where (op, e) is the instruction component with function code op and data field e , \sim is a bitwise NOT, and \gg is a logical right shift.

C.1.2 Evaluating minimal symbol offsets

Several primary instructions have an operand that is an offset between the current value of the instruction pointer and some other part of the code. Generating the optimal prefix sequence to create the instruction data value for one of these instructions is more complicated. This is because two, or more, instructions with offset operands can interlock so that the minimal prefix sequences for each instruction is dependent on the prefixing sequences used for the others.

For example consider the interlocking jumps below which can be prefixed in two distinct ways. The instructions j and cj are respectively *jump* and *conditional jump*. These are explained in more detail later. The sequence:

$$cj + 16; j - 257$$

can be coded as

$$pfix\ 1; cj\ 0; pfix\ 1; nfix\ 0; j\ 15$$

but this can be optimized to be

$$cj\ 15; nfix\ 15; j\ 1$$

which is the encoding for the sequence

$$cj + 15; j - 255$$

This is because when the two offsets are reduced, their prefixing sequences take 1 byte less so that the two interlocking jumps will still transfer control to the same instructions as before. This compaction of non-optimal prefix sequences is difficult to perform and a better method is to slowly build up the prefix sequences so that the optimal solution is achieved. The following algorithm performs this.

- 5 Associate with each jump instruction or offset load an 'estimate' of the number of bytes required to code it and initially set them all to 0.
- 6 Evaluate all jump and load offsets under the current assumptions of the size of prefix sequences to the jumps and offset loads
- 7 For each jump or load offset set the number of bytes needed to the number in the shortest sequence that will build up the current offset.
- 8 If any change was made to the number of bytes required then go back to 2 otherwise the code has reached a stable state.

The stable state that is achieved will be the optimal state.

Where the code being analyzed has alignment directives, then it is possible that this algorithm will not reach a stable state. One solution to this, is to allow the algorithm to increase the instruction size but not allow it to reduce the size. This is achieved by modifying stage 7 to choose the larger of: the currently calculated length, and the previously calculated length. This approach does not always lead to minimal sized code, but it guarantees termination of the algorithm.

Steps 2 and 3 can be combined so that the number of bytes required by each jump is updated as the offset is calculated. This does mean that if an estimate is increased then some previously calculated offsets may have been invalidated, but step 4 forces another loop to be performed when those offsets can be corrected.

By initially setting the estimated size of offsets to zero, all jumps whose destination is the next instruction are optimized out.

Knowledge of the structure of code generated by the compiler allows this process to be performed on individual blocks of code rather than on the whole program. For example it is often possible to optimize the prefixing in the code for the sub-components of a programming language construct before the code for the construct is optimized. When optimizing the construct it is known that the sub-components are already optimal so they can each be considered as a fixed block of code which cannot be reduced.

This algorithm may not be efficient for long sections of code whose underlying structure is not known. If no knowledge of the structure is available (e.g. in an assembler), all the code must be processed at once. In this case a code shrinking algorithm where in step one the initial number of bytes is set to twice the number of bytes per word is used. The prefix sequences then shrink on each iteration of the loop. 1 or 2 iterations produce fairly good code although this method will not always produce optimal code as it will not correctly prefix the pathological example given above.

C.2 Compiling switch statements

The `switch` statement is a special form of conditional transfer where the transfer is determined by comparing an expression to a number of constants.

When compiling the process

```
switch x
...
```

the expression *x* is evaluated and stored in a local variable by

```
x; stl selector
```

Then each branch of the `switch` statement

```
c1, ..., cn
    P
```

can be compiled by

```
    ldl selector; ldc c1; sub; cj L;
    ldl selector; ldc c2; sub; cj L;
    ...
    ldl selector; eqc cn; cj M;
L:    P; j END;
M:
```

where the label *END*: is placed at the end of the `switch` statement.

C.2.3 Optimal compilation of switch

The compilation method given above will produce inefficient code for large `switch` statements. To produce more efficient code the following rules can be used.

First build up a set of pairs of selector values and processes, consisting of every selector value in the `switch` statement along with its associated process — the process part of each pair can be represented by the offset to the start of the compiled code for that process. Then the following rules can be used.

- 1 If there are 3 entries or less then use the method as described above.
- 2 If there are 12 entries or less then use a binary search to limit the number of comparisons required.
- 3 For more than 12 entries attempt to use a jump table. The offset of the start of each selected process is placed in the table against each selector value. Entries that do not match a selector in the `switch` statement must contain the offset of an error handler process. This jump table should be the largest table such that about one third of the entries are filled. This compilation strategy is then recursively called to handle the two ends. The *jab* (section 4.10) and *ldpi* (section 4.5) instructions, can be used to jump to the selected piece of code.

The choice of 3 or less entries, 12 or less entries and one third filled table are the values used in current ST20 compilers.

Consider compiling the following `switch` expression:

```
switch X
  c1
    P1
  ...
  cn
    Pn
```

where, for brevity, it is assumed that all the switch selectors are already in increasing order.

Three entries or less

This switch is compiled as:

```
if (X = c1)
  P1
else if (X = c2)
  P2
...
else if (X = cn)
  Pn
```

Four to twelve entries

This switch is compiled as:

```
if (X <= cn/2 )
{
  if (X <= cn/4 )
    ...etc.
  else
    ... etc.
}
else
  ... etc.
```

Using a jump table

Assume that $c_1 \dots c_m$ form a third filled jump table. Then the switch is compiled as:

```
if ( X < ci )
{
  switch X
    c1
      P1
    ...
    ci-1
      Pi-1
```

```

}
else if (X > cm)
    ... similar
else
    ... jump table code

```

where *jump table code* is

```

X; ldc ci; sub; ldc jump_size; mul; ldc (jump_table-M); ldpi
M:      add; jab;
jump_table:
j case_0; j case_1; ... ; j case_k
ERROR:  ... error code
Li:     ... code for Pi

...
Lm:     ... code for Pm

```

The code at *jump_table* consists of a sequence of jump instructions which transfer control to the relevant branch *Li* . . . *Lm* or to *ERROR*. The destination, *case_x*, of each of these jumps is *Lj* if *c_i* is equal to (*c_i* + *x*) and is *ERROR* otherwise.

The code at *ERROR* should be the same code as used at the end of an IF statement where all the conditionals have been *false*. The *add*, *ldpi* and *jab* instructions are explained in other sections.

All the jumps in the *jump_table* code must be encoded to the same length (*jump_size* bytes) to enable them to be accessed as a byte array. *nop*, which is a two byte instruction that performs no operation, can be used to ensure this where different operands require a different amount of prefixing.

Also note that in the special case where *jump_size* is 1, '*ldc jump_size; mul*' can be removed from the sequence, and where *jump_size* is 4, '*ldc jump_size; mul*' can be removed provided *add* is replaced with *wsb*.

C.3 Expression evaluation order

Let *depth(e)* be the number of stack locations needed for the evaluation of expression *e*, defined by

```

depth(constant)      = 1
depth(variable)      = 1
depth(function call) = 'infinite'
depth(e1 op e2)      = if (depth(e1) > depth(e2))
                        depth(e1)
                        else if (depth(e1) < depth(e2))
                        depth(e2)
                        else
                        depth(e1) + 1

```

That is, if the depth required for each expression is the same, then one extra stack location is required to store the result of the first expression, while the second expression is being evaluated. If the stack requirements for each expression are different,

then the total stack requirement is the larger stack requirement of the individual expressions. This is only the case if care is taken over the order of evaluation. Note that 'infinite' should be taken as meaning greater than any finite depth - because a function call does not preserve values on the stack.

Let the function $eval(e, r)$ evaluate expression e where there are r registers available to perform the evaluation. Where this expression is an operation on two sub expressions - $e1 \text{ op } e2$ - it is efficiently evaluated by the following algorithm, where op commutes is *true* if $a \text{ op } b$ is equal to $b \text{ op } a$ and *false* otherwise:

```

if (depth(e1) < r AND depth(e2) < r) /* i.e. depth of both expressions is less
                                     than the number of registers available */
{
  if (depth(e2) > depth(e1)) {
    if (op commutes) {
      eval(e2, r); eval(e1, r-1); op
    }
    else {
      eval(e2, r); eval(e1, r-1); rev; op      (*)
    }
  }
  else if (depth(e2) <= depth(e1)) {
    eval(e1, r); eval(e2, r-1); op
  }
}
else if (depth(e1) >= r OR depth(e2) >= r) {
  if (depth(e2) >= depth(e1)) {
    if (depth(e1) >= r) { /* i.e. both depths >= r */
      eval(e2, r); stl temp; eval(e1, r); ldl temp; op
    }
    else { /* i.e. depth(e1) < r AND depth(e2) >= r */
      if (op commutes) {
        eval(e2, r); eval(e1, r-1); op
      }
      else { /* i.e. operation doesn't commute */
        eval(e2, r); eval(e1, r-1); rev; op
      }
    }
  }
  else if (depth(e2) < depth(e1)) {
    if (depth(e2) >= r) { /* i.e. both depths >= r */
      if (op commutes) {
        eval(e1, r); stl temp; eval(e2, r); ldl temp; op
      }
      else {
        eval(e2, r); stl temp; eval(e1, r); ldl temp; op
      }
    }
  }
}
else { /* i.e. (depth(e2) < r) AND depth(e1) >= r */
  eval(e1, r); eval(e2, r-1); op
}

```

```

    }
  }
}

```

The justification of this is as follows. If the depth of both expressions is less than the number of registers available (r), then there is no need to store the result of the first evaluation in a temporary variable. The deeper expression is evaluated first to ensure that the operation evaluates in the least number of stack registers. If this were not done then the total depth requirement would have to be incremented due to the extra location for storing the result of the first evaluation. If both expression depths are as great as r , then a local variable (*temp*) must be used to store the result of the first expression. It is again better to evaluate the expression with the larger depth if possible, because this minimizes the number of local variables required. If only one of the expressions is as great as r , then provided that expression is evaluated first, there is no need to store its result in a local variable.

In the cases where a temporary variable *temp* is required to hold the value of the first expression in the evaluation of $e1 \text{ op } e2$, then that variable can be used as a temporary variable in the evaluation of the first expression. Also a temporary variable used in the evaluation of the first expression and not used to hold its result can be used in the evaluation of second expression.

The code sequence

($e2; e1; rev; op$)

at (*) in the above algorithm, can be optimized further to

($e1; e2; op$)

removing the execution of the *rev* instruction. But be aware that the latter uses an extra stack register, and so there is trade-off here between evaluation depth and code size.

C.4 Loading sequence

The three registers of the evaluation stack are used to hold operands of instructions. Evaluation of an operand or parameter may involve the use of more than one register. Care is needed when evaluating such operands to ensure that the first operand to be loaded is not pushed off the bottom of the evaluation stack by the evaluation of later operands. The processor does not detect evaluation stack overflow.

Three registers are available for loading the first operand, two registers for the second and one for the third. Consequently, the instructions are designed so that **Creg** holds the operand which, on average, is the most complex, and **Areg** the operand which is the least complex.

In some cases, it is necessary to evaluate the **Areg** and **Breg** operands in advance, and to store the results in temporary variables. This can sometimes be avoided using the reverse instruction. Any of the following sequences may be used to load the operands *A*, *B* and *C* into **Areg**, **Breg** and **Creg** respectively.

- 1 *C; B; A*
- 2 *C; A; B; rev*
- 3 *B; C; rev; A*
- 4 *A; C; rev; B; rev*

The choice of loading sequence, and of which operands should be evaluated in advance is determined by the number of registers required to evaluate each of the operands.

In particular, if *C* requires more than two registers it must be loaded before *A* and *B*. If *A* or *B* requires more than two registers it must be evaluated before *C* and may need to be stored in a temporary variable if *C* requires more than two registers.

registers required			temp		load sequence	instructions
C	B	A	b	a		
≤ 2	1	1			1	<i>C; B; A</i>
	1	2			2	<i>C; A; B; rev</i>
	1	>2			4	<i>A; C; rev; B; rev</i>
	2	1			1	<i>C; B; A</i>
	2	2		*	1	<i>A; stl a; C; B; ldl a</i>
	2	>2		*	1	<i>A; stl a; C; B; ldl a</i>
	>2	1			3	<i>B; C; rev; A</i>
	>2	2		*	3	<i>A; stl a; B; C; rev; ldl a</i>
	>2	>2		*	3	<i>A; stl a; B; C; rev; ldl a</i>
> 2	1	1			1	<i>C; B; A</i>
	1	2			2	<i>C; A; B; rev</i>
	1	>2		*	1	<i>A; stl a; C; B; ldl a</i>
	2	1			1	<i>C; B; A</i>
	2	2		*	1	<i>A; stl a; C; B; ldl a</i>
	2	>2		*	1	<i>A; stl a; C; B; ldl a</i>
	>2	1	*		1	<i>B; stl b; C; ldl b; A</i>
	>2	2	*		2	<i>B; stl b; C; A; ldl b; rev</i>
	>2	>2	*	*	1	<i>A; stl a; B; stl b; C; ldl b; ldl a</i>

Table 8.1 Register loading sequences

Table 8.1 gives the instruction sequences needed for loading three operands into the evaluation stack, where the number in the 'load sequence' column refers to the list above. The columns labelled 'temp' indicate where a local variable is needed to temporarily save an operand value in the load sequence. The variable 'a' is used to store operand 'A' and the variable 'b' is used to store operand 'B'.

D Glossary

Active process

An active *process* in a *multi-tasking* program is one that is either *executing* or waiting for CPU time. It may have been *interrupted* or *trapped* or it may be waiting on a *scheduling queue*.

Address

A 32-bit integer used to identify one byte of memory or memory-mapped device. ST20 addresses are signed, i.e. the address range is the same as the range of a signed 32-bit integer. A *pointer*.

Address space

The range of valid addresses.

Areg

The 32-bit register at the top of the *evaluation stack*.

Arithmetic shift

A shift in which the sign bit is preserved. An arithmetic right shift copies the sign bit into the vacated bits.

Array

A data structure in which all the elements are of the same type and are identified by a non-negative integer subscript. A *vector*.

Bit

A binary digit, which can take the value 1 (also called *set* or *true*) or 0 (also called *clear* or *false*).

Bitwise operation

An operation on multi-bit values which treats each bit as a true (if *set*) or false (if *clear*).

Boolean

Logic values and calculations using AND, OR, and NOT. A multi-bit value is treated as Boolean if it represents a single true or false value, as opposed to *bitwise*.

Breg

The 32-bit register in the middle of the *evaluation stack*.

Byte

An 8-bit value or location.

Cache

Fast memory used to temporarily hold copies of slow memory locations in order to provide high performance memory access with slow memory.

Channel

A synchronous, one-way, point-to-point, unbuffered communications mechanism.

Clear

A bit is clear if it has the value 0.

Creg

The 32-bit register at the bottom of the *evaluation stack*.

Current process

The process being either being executed by the CPU, interrupted or trapped.

DCU

Diagnostic Controller Unit.

Descriptor

Identifier of a *process* or *exception*.

Deschedule

A *process* is descheduled when it is not being executed by the CPU and has not been *interrupted* or *trapped* by an *exception handler* but it is capable of being restarted. To deschedule a process is to make it descheduled.

Diagnostic Controller Unit

ST20 hardware macro-cell which provides fully non-intrusive breakpoints, watch-points and code tracing.

DMA

Direct memory access. A *peripheral* or external device performs DMA when it reads directly from or writes directly to memory.

DMA peripheral

A *peripheral* with a *DMA* engine.

Double word

A 64-bit value, occupying two words.

Evaluation stack

The **Areg**, **Breg** and **Creg**, which together behave as a stack. The evaluation stack is used for expression evaluation and to hold operands for instructions. The registers cannot be individually addressed, but values can be pushed onto the stack or popped off it.

Exception

A mechanism for handling events outside the normal program flow. An exception may be an *interrupt* or a *trap*.

Exception handler

The code and data which defines the action when an *exception* is taken. An exception handler may be a *trap handler* or an *interrupt handler*.

Exception level

The signed integer specifying which *exception* will be taken, which is the word offset from the base of the *exception vector table* to the *descriptor* of the *process* or *exception*.

Exception vector table

The table mapping *exception levels* to user *processes* and *exception descriptors*.

Executing

A *process* is executing if the CPU is currently modifying its *state*. Only one process can be executing at any one time.

External interrupt

An *interrupt* generated by a *peripheral* or external device, routed through the *interrupt controller* and handled by an *interrupt handler*.

Function

A named section of code which may have parameters and may return one or more values.

Function

A primary instruction.

Function code

The most significant 4-bits of an *instruction component*, which defines the action of the component.

Half-word

A 16-bit value or location.

Half-word aligned

An address is half-word aligned if it is divisible by two, i.e. if bit 0 is 0.

Idle

The CPU is idle when there is no *process* or *exception handler* executing.

Inactive

A process is inactive if it is capable of being restarted but it is currently waiting for some event to occur before it can continue. The event might be a semaphore signal or a peripheral completing a DMA.

Instruction

A code element corresponding to a single instruction set mnemonic, consisting of zero or more *prefixes* followed by a non-prefix *instruction component*.

Instruction component

A single byte of code, consisting of a function code and 4 bits of data.

Instruction data value

The operand of a *primary instruction*, which is built from the least significant 4 bits of each of the *instruction components* of the instruction.

Instruction pointer

The 32-bit CPU register which holds the address of the next *instruction* to be executed by the *current process*.

Integer

A whole number, which may be positive, negative or zero.

Internal memory

On-chip memory.

Interrupt

A signal from outside the CPU to switch from normal execution to execution of an *interrupt handler*. If the interrupt causes a scheduling event, then that may cause an immediate *trap*.

Interrupt controller

An on-chip *peripheral* which arbitrates between *interrupt* requests and signals to the CPU when an interrupt is required.

Interrupt handler

The code and data which defines the action when an *interrupt* is taken.

Interrupt latency

The time taken from receiving an *interrupt* signal to starting execution of the *interrupt handler*.

Interrupted

A *process* is in an interrupted state if it is waiting for an *interrupt handler* to complete and it will continue as soon as the interrupt handler has completed.

IReg

The input and output register, whose bits are directly connected to *peripherals*.

Iptr

The *instruction pointer*.

Linked list

A linked list is an ordered data structure where each element includes a pointer to the next element. A list is identified by a front pointer, which points to the first element in the list. Linked lists are also called *queues*.

Little-endian

Little-endian ordering of data, used by all ST20s, means that less significant data is always held in lower addresses. This applies to bits in bytes, bytes in words and words in memory.

List

See *linked list*.

Local

A local variable is one that is addressed by means of an offset from the workspace pointer. Local variables are usually held on a stack and are defined within a function or procedure. *cf. non-local*.

Logical shift

A shift in which the sign bit is shifted by the same amount as any other bit.

Memory

Addressed devices which can be read and possibly written. Reading read/write memory at a particular address gives the value that was last written to that address. Reading read-only memory always gives the same value.

Memory-mapped

Addressed devices which are read and written as if they were *memory*, but do not necessarily return the last written value.

Microcode

The on-chip hardware mechanism for implementing instructions.

Micro-kernel

The ST20 on-chip hardware support for multi-tasking.

MostNeg

The most negative 32-bit integer, #80000000. The address of the base of memory. The value of the null process pointer *NotProcess*.

Multi-tasking

Running several communicating *processes* on the same processor using time sharing.

Non-local

A non-local variable is one that is addressed by means of an offset from the base address held in **Areg**. Non-local variables are usually not held on a stack and are defined globally. *cf. local*.

NotProcess

The null *process* pointer, which has the value *MostNeg*.

On-chip emulator

Diagnostic controller unit.

On-chip memory

Very fast *memory* included in the ST20 chip. On-chip memory is usually located at the bottom of the *address space*.

Operate

An *instruction component*, with function code #F and mnemonic *opr*, used to encode *secondary instructions*. The *instruction data value* identifies the secondary instruction.

Operating system

Code controlling multi-tasking, interrupts and certain system operations, which may be called by a program. The ST20 supports many of these facilities in micro-code without an operating system.

Operation

A secondary instruction.

Peripheral

A device connected to and controlled by the CPU.

Pointer

An *address*.

Prefix

An *instruction component* used to extend the data value of the following instruction component. There are two prefixes - *prefix* (function code #2) which preserves the sign and *ntfix* (function code #6) which complements the data value and so changes the sign.

Primary instruction

One of thirteen directly coded instructions which therefore do not require an *operate* instruction component. A primary instruction has a data part associated with it, called the operand. *cf. secondary instruction*.

Priority

A value associated with each process or interrupt which is used to make choices when conflicts occur.

Procedure

A named section of code which may have parameters.

Process

A sequential algorithm and data which can run in parallel with other processes. Processes are also known as tasks or threads. Each process is identified by a task descriptor which points to a process descriptor block.

Process descriptor block

A data structure defining a *process*, including the saved *instruction pointer*, *workspace pointer* and a link used for *queueing* processes.

Process pointer

A pointer to a *process*.

Queue

A linked list.

Real time operating system

An *operating system* providing facilities for real-time programming, including *multi-tasking* and *interrupt* handling.

Register cache

Registers used to *cache* the lowest words of the *work space* in order to accelerate access to local variables. Also called *work space cache* or *register file*.

Register file

See *register cache*.

RTOS

Real-time operating system.

Scheduler

Code or hardware which controls how processes share CPU time.

Scheduling kernel

Code to override the ST20 built-in *microcode scheduler*, consisting mainly of *trap handlers* which are taken when a scheduling event occurs.

Scheduling queue

The *queue* of *processes* waiting for CPU time.

Secondary instruction

An instruction encoded using an *operate* instruction component. A secondary instruction has no data part associated with it, as the instruction data value is used to identify the instruction. *cf. primary instruction*.

Semaphore

A mechanism for synchronizing and signalling between processes.

Set

A bit is set when its value is 1.

Sign extension

A signed value can be extended to occupy more bits by filling the most significant bits with copies of the sign bit.

Sign bit

The most significant bit of a *signed value*, used to indicate the sign of the value.

Signed value

Value which could be positive or negative depending on the *sign bit*.

Sleep

The CPU is asleep when its clocks are turned off to reduce power consumption. The CPU automatically goes to sleep after it becomes idle. It is woken by an external *interrupt*.

Slot

A slot is a field of a data structure for holding a value. A slot normally occupies one *word*.

Stack

A stack is a data storage structure that uses a 'first in, last out' access model. Adding an item to a stack is called pushing on, and extracting a value is called popping. A program stack is normally implemented using the Wptr as the stack pointer. The **Areg**, **Breg** and **Creg** form the *evaluation stack*.

Stack, evaluation

See *evaluation stack*.

State

The state of a process is the data needed for the process to continue execution. This may include the evaluation stack, **Ip**tr, **Wp**tr, status register and local data.

Status register

The status register holds CPU status information which must be saved if an *interrupt* occurs but is not saved if the process deschedules. Some status register

bits are global and are preserved when a process is descheduled and others are local to a *process* and are lost when the process is *descheduled*.

Task

A *process*.

Task descriptor

The identifier of a *process*, which is a pointer to the *process descriptor block*. When a process is executing, the **Tdesc** holds the task descriptor of the process.

Tdesc

A 32-bit register which holds the *task descriptor* of the current *process*.

Thread

A dynamically generated *process*.

Timeslice

Deschedule a *process* to allow other processes access to the CPU. Timeslicing prevents any one process from occupying too much CPU time. A timesliced process is normally added to the back of the *scheduling queue* and the process on the front of the queue is restarted.

Trap

An *exception* caused by software, or an interrupt initiating a scheduling event.

Trap handler

The code and data which define what happens when a *trap* is taken.

Trapped

A process is trapped when it is waiting for a *trap handler* to complete.

Unsigned value

Value which could only be positive, so the most significant bit is not used as a *sign bit*.

Vector

An *array*.

Waiting

A process is waiting for an event if it cannot continue execution until that event has occurred. The event could be for example a communication or a *semaphore* signal.

Word

A word is a four byte (32-bit) location for data.

Word-aligned

An *address* is word-aligned if it is divisible by 4, i.e. if the least significant two bits are zero.

Work space

Space in memory for *local* variables and values.

Work space cache

Registers used to *cache* the lowest words of the *work space* in order to accelerate access to local variables. Also called *register cache* or *register file*

Workspace pointer

A 32-bit register pointing to the *work space* for the *current process*.

Wptr

The workspace pointer.

Index

Symbols

+ (plus), 11
 .. (ellipsis), 9
 / (divide), 11
 < (less than), 11
 = (equals), 11
 > (greater than), 11
 >> (logical shift right), 11
 @ (word offset), 10
 { } (braces), 13
 ' (prime), 10
 ≤ (less than or equal to), 11
 ≥ (greater than or equal to), 11
 ≠ (is not equal to), 11

Numerics

16-bit, 32
 assignment, 35
 load, 31, 124
 signed, 31, 125
 multiply, 146
 multiply accumulate, 56, 143, 144
 sign extension, 40, 167
 store, 31, 147

A

active, 78, 187
adc, 36, 88
 add, 36, 89
 constant, 36, 88
 with carry, 38, 90
addc, 38, 90
 address, 187
 calculation, 10, 41–43
 load local, 41
 load non-local, 41
 space, 18, 187
 subscript, 41
 addressing, 16
 adjust work space, 91
ajw, 48, 91
 alignment, 18
 and
 bitwise, 11, 47
 boolean, 11, 46
 instruction, 92
 anti-rotate stack, 30, 93
 architecture, 16–28
Areg, 20, 21, 30, 187
 arithmetic, 35–40

add, 89
 boolean, 45–47
 divide, 102
 modulo, 11
 multiply accumulate, 56–67
 saturated, 138
 shift, 187
 shift right, 94
 subtract, 155
 subtract with carry, 156
 unchecked, 11
 unsign, 162
arot, 30, 93
 arrays, 41, 187
 word subscript, 164
ashr, 48, 94
 assignment
 byte, 35
 single word, 35

B

biquad, 56, 58, 95
 bit, 187
 load, 47, 96
 mask, 47
 read modify write, 47, 135
 store, 47, 98
bitld, 47, 96
bitmask, 47, 52, 97
BitsPerWord, 172
bitst, 47, 98
 bitwise, 187
 and, 11
 exclusive or, 166
 not, 11, 130
 operations, 47
 or, 11, 131
 xor, 11
 block copy, 31, 33
 load, 126
 store, 158
 boolean, 187
 expressions, 45–47
 branch, 43, 100
 breakpoint, 71, 99
 trap, 71
Breg, 20, 21, 30, 187
 buffer size
 multiply accumulate, 57
 byte, 32, 187
 addressing, 19

- arrays, 43
- assignment, 35
- load, 31
- load and increment, 115
- ordering
 - compatibility, 17
- selector, 18
- sign extension, 40, 165
- store, 31, 139
- swap, 48, 157
- BytesPerWord*, 172

C

- cache, 187
- call
 - exception, 104
 - function, 48–52, 108
 - procedure, 48–52
- carry, 22, 38
- channel, 187
- channel-type peripherals, 53
- checked arithmetic, 36
- cj*, 43, 100
- clear, 188
 - status bit, 55
 - status bits, 148
- clock
 - timeslicing, 81
- code
 - in instruction descriptions, 6
- coding of instruction, 6, 24
- comments
 - in instruction descriptions, 8, 9
- comparison, 43–45
 - equals constant, 43, 106
 - greater than, 43, 110
 - unsigned, 43, 111
 - order, 132
 - unsigned, 133
- compatibility
 - byte ordering, 17
- complement, 47
- conditional jump, 43, 100
- conditionals, 43–45
- conditions
 - in instruction descriptions, 13
- constants, 172
 - equals, 43
 - loading, 31, 116
 - from table, 34
 - machine constant definitions, 14, 169
 - used in instruction descriptions, 14, 172
- control block
 - exception, 73

- process queue, 80
- conversion
 - object length, 17, 40
 - type, 12
- create bit mask, 97
- Creg**, 20, 21, 30, 188
- current process, 188

D

- data format
 - multiply accumulate, 58, 65
- data structures, 171
- data types, 9
- DCU, 188
- decrement, 36, 88
- definition
 - in instruction descriptions, 7, 8
- depth
 - of stack needed, 182
- dequeue a process, 79, 101
- descheduled process, 79, 188
 - state, 80, 82
- description
 - in instruction descriptions, 7
- descriptor, 188
 - in exception vector table, 72
 - of process, 79
 - task, 24
- diagnostic controller unit, 188
- direct memory access, 53
- disabling
 - exceptions, 77
 - timeslicing, 81
- division, 36, 102
- divstep*, 36, 102
- DMA, 188
 - peripherals, 53
- double word, 19, 188
 - arithmetic, 38
 - multiply, 39
- DSP, 56–67
- dup*, 30, 103
- duplicate top of stack, 30, 103
- dynamic allocation of workspace, 52

E

- ecall*, 71, 104
- ellipsis, 9
- else, 13
- emulator, 191
- enabling
 - exceptions, 77
 - timeslicing, 81
- encoding of instructions, 6, 24

enqueue a process, 79, 105
eqc, 43, 106
 equals constant, 43, 106
eret, 71, 107
 error signals
 in instruction descriptions, 8
 evaluation
 arithmetic, 35–40
 boolean, 45–47
 function, 51
 of addresses, 41–43
 of expressions, 33–35, 182–185
 stack, 19, 21, 30, 33–35, 182–185, 188
 loading sequences, 34
 subscripts, 42
 exception, 70–77, 188
 call, 104
 control block, 73
 handler, 188
 initial state, 74
 levels, 71, 170, 188
 return, 107
 type, 72
 vector table, 72, 189
ExceptionBase, 72, 172
ExceptionProcessType, 73, 172
 exclusive or, 166
 bitwise, 47
 executing, 78, 189
 expression
 depth, 182–184
 evaluation, 33–35, 182–185
 using functions, 51
 extend sign, 40, 165
 external interrupt, 189

F

false, 16, 43, 172
fcall, 48, 108
 filter
 biquad IIR, 58
 IIR step, 95
 function, 189
 call, 48–52, 108
 code, 7, 25, 189
 evaluation, 51
 functions
 in instruction descriptions, 12

G

gajw, 48, 109
 general adjust workspace, 109
global_interrupt_enable, 22, 77

greater than, 43, 110
 unsigned, 43, 111
gt, 43, 110
gtu, 43, 111

H

half-word, 16, 32, 189
 aligned, 19, 189
 assignment, 35
 load, 31, 124
 signed, 31, 125
 multiply, 36, 146
 multiply accumulate, 56, 143, 144
 sign extension, 40, 167
 store, 31, 147
high_word, 12
HighestException, 172

I

i/o, 52–55
 identity
 of device, 14, 122
 idle, 85, 189
 trap, 72
 if
 compiling, 43–45
 in definitions, 13
 IIR filter, 58, 95
 inactive, 78, 82, 189
 process restarting, 82
 increment, 36, 88
 indirection of exceptions, 72
 initializing
 exception handlers, 76
 multi-tasking, 83
 smacloop, 143
 input/output, 52–55, 112
 instruction, 14, 189
 address, 41
 component, 25, 189
 data value, 25, 189
 definitions, 87
 encoding, 6, 24
 pointer, 20, 189
 load, 41, 121
 int16, 12
 integer, 189
 length conversion, 40
 internal memory, 190
 interrupt, 70–77, 190
 controller, 190
 enable, 22
 handler, 190

latency, 190
interrupt_mode, 22, 23
interrupted, 78, 190
io, 52, 112
IOreg, 20, 24, 52, 190
lptr, 20, 190
load, 41

J

j, 43, 113
jab, 43, 48, 114
jump, 43–45, 112, 113
absolute, 43, 114
conditional, 43, 100
table, 181

L

lbinc, 31, 115
ldc, 31, 116
ldl, 31, 117
ldlp, 41, 118
ldnl, 31, 119
ldnlp, 41, 120
ldpi, 41, 121
ldprodid, 14, 122
ldtdesc, 52, 79, 123
length conversion, 17
link
in queue, 80
linked list, 80, 190
list, 190
little-endian, 16, 19, 157, 190
load, 31–33
bit, 47, 96
byte, 31, 115
constant, 31, 116
instruction pointer, 41
local, 31, 117
local pointer, 41, 118
non-local, 31, 119
non-local pointer, 41, 120
operands, 34
parameters, 50
pointer to instruction, 121
product identity, 122
sequence
integer stack, 34
sequences, 184
sixteen bit, 31, 124
signed, 125
sixteen bit signed, 125
task descriptor, 123
word, 31, 126

local, 19, 23, 31, 190
load pointer, 118
load variable, 31, 117
store variable, 31, 151
local_interrupt_enable, 22, 77
logic, 45–47
bitwise instructions, 47
logical shift, 190
long arithmetic, 38
multiplication, 127
subtract, 156
LongMode, 58, 172
low_word, 12
LowestException, 172
lsinc, 31, 124
lsxinc, 31, 125
lwinc, 31, 126

M

mac, 56, 127
mac_buffer, 22, 57
mac_count, 22, 57
mac_mode, 22, 57
mac_scale, 22, 57
mask, 47
create, 97
maximum, 45, 132
unsigned, 133
MaxTimesliceCount, 81, 172
memory, 18, 191
block copy, 33
mapped, 191
mapped peripherals, 53
read modify write, 47, 135
representation of, 10
microcode, 191
micro-kernel, 191
minimum, 45, 132
unsigned, 133
minus, 36, 38
modulo arithmetic, 11
MostNeg, 16, 172, 191
MostPos, 16, 172
move, 33
mul, 36, 128
multiply, 36, 128
half-word, 36
multiple length, 39
multiply accumulate, 56–67, 127
short loop, 144
initialize, 143
unsigned, 38, 161
multi-tasking, 78–85, 191

N

negation, 38
next instruction, 9
nfix, 25
 no operation, 129
 non-local, 31, 191
 load pointer, 120
 load variable, 31, 119
 store variable, 152
nop, 129
 not
 bitwise, 47
 boolean, 11, 46
 instruction, 130
 notation, 6–14
NotProcess, 18, 172, 191

O

object length conversion, 40
 objects, 9
 On-chip emulator, 191
 on-chip memory, 191
 operands
 in descriptions, 6
 primary instructions, 26
 secondary instructions, 34
 operate, 25, 191
 operating system, 191
 operation, 191
 code, 6, 25, 27
 operations
 bitwise, 47
 operators
 in instruction definitions, 11
 in instruction descriptions, 11
opr, 8, 27
 or
 bitwise, 11, 47
 boolean, 11, 46
 instruction, 131
 order, 43
 of loads, 184
 unsigned, 43
order, 132
 ordering
 of data, 16
 values, 43–45
orderu, 133
 output, 52–55, 112
 overflow, 22, 36, 37
 saturate, 138

P

parallel programming, 78–85
 parameters
 procedure, 50
 peripherals, 52–55, 191
pfix, 25
 plus, 36
 pointer, 192
 load local, 41
 load non-local, 41
 reserved values, 18
 pop, 21
 prefixing, 25, 178, 192
 primary instructions, 8, 25, 26, 192
 prime notation
 in instruction descriptions, 10
 priority, 78, 170, 192
 procedure, 192
 call, 48–52
 process, 78–85, 192
 creation, 83
 current, 188
 dequeue, 101
 descriptor, 24, 170
 descriptor block, 79, 192
 enqueue, 105
 pointer, 192
 queues, 80
 run, 137
 state, 9, 24, 79
 in descriptions, 7
 states, 78
 stop, 153
 timeslice, 159
 transitions, 78
 user, 22
 waiting, 80
 product identity, 14, 122
ProductId, 172
 push, 21
pw.lptr, 82
pw.Wptr, 82

Q

q.BptrLoc, 80, 171
q.FPtrLoc, 80, 171
 Q15, 58
 Q31, 58, 65
 queue, 192
 control block, 80
 scheduling, 80

R

- read modify write, 47, 135
- real time operating system, 192
- register, 20
 - cache, 20, 192
 - file, 192
 - in instruction descriptions, 7
 - memory mapped, 53
- rem, 11
- remainder, 36
- repeat loop, 45
- representing memory
 - in instruction descriptions, 10
- rescheduling a process, 82
- reserved memory, 18
- return
 - exception, 107
 - from procedure, 49
- rev, 30, 134
- reverse stack, 30, 134
- rmw, 47, 135
- rot, 30, 136
- rotate stack, 30, 136
- RTOS, 192
- run, 79, 82, 84, 137
 - trap, 72

S

- s.Back**, 79, 84, 171
- s.Count**, 84, 171
- s.Front**, 84, 171
- saturate, 37, 138
- SavedTaskDescriptor, 172
- sbinc, 31, 139
- scaling
 - multiply accumulate, 59
- scheduler, 192
- SchedulerQptr, 80, 172
- scheduling, 78–85
 - kernel, 193
 - kernels, 84
 - queue, 80, 193
- secondary instructions, 8, 25, 27, 193
- semaphore, 84–85, 193
 - signal (V), 84, 85, 142
 - wait (P), 85, 163
- set, 193
 - status bits, 55, 149
- setting up
 - exception handler, 76
 - multi-tasking, 83
 - process, 83
 - semaphore, 84
- shift, 48
 - instructions, 47
 - logical left, 140
 - right arithmetic, 94
 - right logical, 141
- shl, 48, 140
- short
 - multiply, 36, 146
 - multiply accumulate loop, 144
 - store, 147
- ShortMode, 58, 172
- shr, 48, 141
- sign
 - bit, 193
 - extension, 17, 40, 193
 - byte, 165
 - half-word, 31, 167
- signal, 84, 142
- signal processing, 56–67
- signed
 - arithmetic, 17
 - shift right, 48
 - value, 193
- sixteen bit, 16, 32
 - assignment, 35
 - load, 31, 124
 - signed, 31, 125
 - multiply, 36, 146
 - multiply accumulate, 56, 143, 144
 - sign extension, 40, 167
 - store, 31, 147
- sleep, 22, 23, 85, 193
- slot, 16, 193
- smacinit, 56, 57, 143
- smacloop, 56, 144
 - initialize, 143
- smul, 36, 146
- ssinc, 31, 147
- stack, 193
 - anti-rotate, 30
 - duplicate, 30, 103
 - evaluation, 19, 21, 193
 - memory, 19
 - reverse, 30, 134
 - rotate, 30, 136
- start_next_task**, 22, 23
- state, 193
- status register, 8, 20, 21, 55, 170, 193
 - clear bits, 148
 - exception handler start state, 74
 - restart value, 83
 - restarted process state, 83
 - set bits, 149
 - test, 150
- statusclr, 55, 148

statusset, 55, 149
statustst, 55, 150
 steps
 multiply accumulate, 57
stl, 31, 151
stnl, 31, 152
stop, 52, 79, 84, 153
 trap, 72
 store, 31–33
 bit, 47, 98
 byte, 31, 139
 local, 31, 151
 non-local, 31, 152
 sixteen bit, 31
 word, 31, 158
sub, 36, 155
subc, 38, 156
 subprogram
 call, 48–52
 subscript, 42
 address, 41
 evaluation, 42
 in instruction definitions, 9
 word, 164
 subtract, 36, 155
 with carry, 38, 156
 swap bytes, 48, 157
swap32, 17, 48, 157
swinc, 31, 158
 switch statements, 180
 system traps, 71, 84

T

table of constants
 loading from, 34
 task, 78, 194
 descriptor, 79, 194
 load, 123
Tdesc, 20, 24, 194
 test status bit, 55, 150
 threads, 78, 194
 timeslice, 45, 79, 81, 159, 194
 trap, 72
timeslice_count, 22, 81
timeslice_enable, 22, 81
trap_mode, 22, 23
 trapped, 78, 194
 traps, 70–77, 194
 handlers, 194
true, 16, 43, 172
 type
 conversion, 12
 of data, 9
 of exception, 72

U

umac, 38, 56, 161
 unary minus, 38
 unchecked arithmetic, 11
 undefined, 9
 underflow, 22, 36, 37
 saturate, 138
unsign, 36, 162
 unsigned, 10
 greater than, 43, 111
 multiply accumulate, 38, 161
 order, 133
 value, 194
 user process, 22
user_mode, 22
UserProcessType, 73, 172

V

values, 16
 variable
 address, 41
 allocating work space for, 49
 assignment, 35
 vector, 194
 exception table, 72

W

wait, 84, 163
 waiting process, 79, 80, 194
 while loop, 45
 word, 16, 194
 address, 18
 aligned, 18, 194
 arrays, 43
 assignment, 35
 load, 126
 multiply accumulate, 56
 store, 158
 subscript, 41, 164
 work space, 19, 23, 170, 194
 address, 23
 adjust, 49, 91
 cache, 20, 195
 dynamic allocation, 52
 general adjust, 109
 pointer, 19, 23, 195
Wptr, 19, 20, 23, 195
wsub, 41, 164

X


x (multiply), 11
xbword, 40, 165
 xor

bitwise, 11, 47, 166
xsword, 40, 167

Information furnished is believed to be accurate and reliable. However, SGS-THOMSON Microelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of SGS-THOMSON Microelectronics. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. SGS-THOMSON Microelectronics products are not authorized for use as critical components in life support devices or systems without express written approval of SGS-THOMSON Microelectronics.

© 1997 SGS-THOMSON Microelectronics - All Rights Reserved

IMS and DS-Link® are trademarks of SGS-THOMSON Microelectronics Limited.

 is a registered trademark of the SGS-THOMSON Microelectronics Group.

Document number: 72-TRN-274-01