ST20C2/C4 Core Instruction Set Reference Manual

72-TRN-273-01 January 1996

SGS-THOMSON MICROELECTRONICS

Contents

1 Introduction

This manual provides a summary and reference to the ST20 instruction set for C2 and C4 cores. 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 complete description of the instruction, using the notation described below in section 1.7;
- **Error signals:** a list of errors and other signals which can occur;
- **Comments:** a list of other important features of the instruction;
- **See also:** for some instructions, a cross reference is provided to other instructions with a related function.

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

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.

1.2 Code

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 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 ‘Function 8’.

Example

The entry for the *add* instruction is:

Code: F5

1.3 Description

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

Example

The *add* instruction contains the following description:

> **Description:** Add **Areg** and **Breg**, with checking for overflow.

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 values in registers and memory before and after the instruction has executed).

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

> register′ ← *expression involving registers, etc.*

Primed names (e.g. Areg′) represent values after instruction execution, while unprimed names represent values when 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 the register on the left hand side becomes equal to the value of the expression on the right hand side after the instruction has been executed.

The description is written with the main function of the instruction stated first (e.g. 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 (e.g. popping the stack). There is no temporal ordering implied by the order in which the statements are written.

The notation is described more fully in section 1.7.

Example

The *add* instruction contains the following description:

Definition:

> Areg′ ← Breg $+_{checked}$ Areg
>
> Breg′ ← Creg
> Creg′ ← *undefined*

This says that the integer stack is popped and **Areg** assigned the sum 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** is undefined.

1.5 Error signals

This section lists the errors and other exceptional conditions that can be signalled by the instruction. This only indicates the error signal, not the action that will be taken by the processor - this will depend on the trap enable bits which are set, the value in the trap handler location, etc.

The order of the error signals listed is significant in that if a particular error is signalled then errors later in the list may not be signalled. The errors that may be signalled are as follows:

IntegerError indicates a variety of general errors such as a value out of range.

IntegerOverflow indicates that an overflow occurred during an integer arithmetic operation.

LoadTrap indicates that an attempt has been made to load a new trap handler. This provides a basic mechanism for a supervisor kernel to manage user processes installing trap handlers.

StoreTrap, analogous to *LoadTrap*, indicates that an attempt has been made to store a trap handler so that it can be inspected. Again this allows a supervisor kernel to manage the trap system used by user processes.

Example

As an example, the error signals listed for the *add* instruction are:

Error signals:

IntegerOverflow can be signalled by $+_{checked}$

So, the only error that can be caused by *add* is an integer overflow during the addition of **Areg** and **Breg**.

1.6 Comments

This section is used for listing other information about the instructions that may be of interest. Firstly, there is an indication of the type of the instruction. These are:

“Primary instruction” — indicates one of the 13 functions which are directly encoded in a single byte instruction.

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

Then there is information concerning the scheduling of the process:

“Instruction is a descheduling point” — a process may be descheduled after executing this instruction.

“Instruction is a timeslicing point” — a process may be timesliced after executing this instruction.

“Instruction is interruptible” — the execution of this instruction may be interrupted by a high priority process.

This section also describes any situations where the operation of the instruction is undefined or invalid.

Example

Using the *add* instruction as an example again, the comments listed are:

Comments:

Secondary instruction.

This says that *add* is a secondary instruction.

1.7 Notation

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

1.7.1 The processor state

The processor state consists of the registers (mainly **Areg**, **Breg**, **Creg**, **Iptr**, and **Wptr**), the contents of memory, and various flags and special registers (such as the error flags, process queue pointers, clock registers, etc.).

The **Wptr** register is used for the address of the workspace of the current process. This address is word aligned and therefore has the two least significant bits set to zero. **Wdesc** is used for the 'process descriptor' — the value that is held in memory as an identifier of the process when the process is not being executed. This value is composed of the top 31 bits of the **Wptr** plus the process priority stored in bit 0 of the word. Bit 0 is set to 0 for high priority processes and 1 for low priority processes. Bit 1 of the process descriptor is always 0.

1.7.2 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 block move instructions are described in terms of a sequence of *byte* reads and writes even though the instructions are implemented to perform the minimum number of *word* reads and writes.

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. *start next process*. These actions may be performed in another subsystem in the device, such as the communications subsystem, and so any changes to machine state are not necessarily completely synchronized with the execution of the instruction (as the different subsystems work independently and in parallel).

Ellipses are 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.

Generally, 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.

One exception to this general rule is **Iptr**, which is assigned the address of the next instruction in the code *before* every instruction execution starts. The **Iptr** is included in the description only when it is *directly* affected by the instruction (e.g. in the *jump* instruction). In these cases the address of the next instruction is indicated by the comment “*next instruction*”.

Scheduling operations

Some registers, such as the timer and scheduling list pointers and some special workspace locations, can be changed at any time by scheduling operations. Changes to these are included in the description only when they are *directly* caused by the instruction, and not just as an effect of any scheduling operation which might take place.

1.7.3 Undefined values

Many instructions 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 the integer stack is popped, **Creg** becomes undefined, i.e. it does 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 of the ST20 family of processors.

1.7.4 Data types

The instruction set includes operations on four sizes of data: 8, 16, 32 and 64-bit objects. 8-bit and 16-bit data can represent signed or unsigned integers; 32-bit data can represent addresses, signed or unsigned integers, or single length floating point numbers; and 64-bit data can represent signed or unsigned integers, or double length floating point values. Normally it is clear from the context (e.g. from the operators used) whether a particular object represents a signed, unsigned or floating point number. A subscripted label is added (e.g. $Areg_{unsigned}$) to clarify where necessary.

1.7.5 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 four data types is represented in the instruction descriptions in the following way:

byte [*address*] references a byte in memory at the given address

sixteen [*address*] references a 16-bit object in memory 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. word[]), and that *after* the instruction if the function is used with a prime (e.g. word′[]).

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

word′[*addr*] $\leftarrow$ *expr*

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

Areg′ $\leftarrow$ 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, e.g. writing a byte at address 0 will modify the least significant byte of the word at address 0.

Reading and writing in this way cannot be used to access on-chip peripherals. Reading or writing to memory addresses between *PeripheralStart* and *PeripheralEnd* will have undefined effects.

Data alignment

Each of these 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 and 64-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, words or double words depending on the data type being accessed. In order to calculate the address of the data, the 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 in the particular units being used. So, for example, a word offset is converted to a byte offset by multiplying it by the number of bytes in a word (4 in the case of the ST20).

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

Areg′ $\leftarrow$ word[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. **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.

1.7.6 On-chip peripherals

On-chip peripherals may have memory-mapped registers in the address range *PeripheralStart* to *PeripheralEnd*. Access to these registers is represented in the following way:

PeripheralByte[*address*] references an 8-bit peripheral register

PeripheralSixteen[*address*] references a 16-bit peripheral register

PeripheralWord[*address*] references a 32-bit peripheral register

For all of these, the state of the peripheral referenced is that *before* the instruction if the function is used without a prime (e.g. PeripheralWord[ ]), and that *after* the instruction if the function is used with a prime (e.g. PeripheralWord′[ ]).

For example, writing a value given by an expression, *expr*, to the register at address *addr* is represented by:

PeripheralWord′[*addr*] $\leftarrow$ *expr*

and reading a word from a peripheral is achieved by:

Areg′ $\leftarrow$ PeripheralWord[*addr*]

1.8 Block move registers

A group of registers is used in the implementation of block moves. These are referred to as the ‘block move registers’ and include *Move2dBlockLength*, *Move2dDestStride*, and *Move2dSourceStride*.

1.9 Constants

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

These data structures is listed in Table 1.2 to Table 1.4.

word offset	slot name	purpose
0	**pw.Temp**	slot used by some instructions for storing temporary values
-1	**pw.Iptr**	the instruction pointer of a descheduled process
-2	**pw.Link**	the address of the workspace of the next process in scheduling list
-3	**pw.Pointer**	saved pointer to communication data area
-3	**pw.State**	saved alternative state
-4	**pw.TLink**	address of the workspace of the next process on the timer list
-5	**pw.Time**	time that a process on a timer list is waiting for

Table 1.1 Process workspace data structure

word offset	slot name	purpose
0	**le.Index**	contains the loop control variable
1	**le.Count**	contains number of iterations left to perform

Table 1.2 Loop end data structure

word offset	slot name	purpose
1	**pp.Count**	contains unsigned count of parallel processes
0	**pp.IptrSucc**	contains pointer to first instruction of successor process

Table 1.3 Parallel process data structure

word offset	slot name	purpose
2	**s.Back**	back of waiting queue
1	**s.Front**	front of waiting queue
0	**s.Count**	number of extra processes that the semaphore will allow to continue running on a *wait* request

Table 1.4 Semaphore data structure

In addition, a number of constants are used to identify word length related values etc.; These are listed in Table 1.5 .

Name	Value	Meaning
BitsPerByte	8	The number of bits in a byte.
BitsPerWord	32	The number of bits in a word.
ByteSelectMask	#00000003	Used to select the byte select bits of an address.
WordSelectMask	#FFFFFFFC	Used to select the byte select bits of an address.
BytesPerWord	4	The number of bytes in a word.
MostNeg	#80000000	The most negative integer value.
MostPos	#7FFFFFFF	The most positive signed integer value.
MostPosUnsigned	#FFFFFFFF	The most positive unsigned integer value.
PeripheralStart	#20000000	The lowest address reserved for memory-mapped on-chip peripherals.
PeripheralEnd	#3FFFFFFF	The highest address reserved for memory-mapped on-chip peripherals.

Table 1.5 Constants used in the instruction descriptions

A number of values are used by the ST20 to indicate the state of a process and other conditions. These are listed in Table 1.6.

Name	Value	Meaning
DeviceId	Depends on processor type. See below.	A value used to identify the type and revision of processor. Returned by the *ldprodid* and *lddevid* instructions.
Disabling.p	*MostNeg* + #03 #80000003	Stored in the **pw.State** location while an alternative is being disabled.
Enabling.p	*MostNeg* + #01 #80000001	Stored in the **pw.State** location while an alternative is being enabled.
false	0	The boolean value 'false'.
NoneSelected.o	-1 #FFFFFFFF	Stored in the **pw.Temp** slot of a process' workspace while no branch of an alternative has yet been selected during the waiting and disabling phases.
NotProcess.p	*MostNeg* #80000000	Used, wherever a process descriptor is expected, to indicate that there is no process.
Ready.p	*MostNeg* + #03 #80000003	Stored in the **pw.State** location during the enabling phase of an alternative, to indicate that a guard is ready.
TimeNotSet.p	*MostNeg* + #02 #80000002	Stored in **pw.TLink** location during enabling of a timer alternative after a time to wait for has been encountered.
TimeSet.p	*MostNeg* + #01 #80000001	Stored in **pw.TLink** location during enabling of a timer alternative after a time to wait for has been encountered.
true	1	The boolean value 'true'.
Waiting.p	*MostNeg* + #02 #80000002	Stored in the **pw.State** location by *altwt* and *taltwt* to indicate that the alternative is waiting.
HighPriority	0	High priority
LowPriority	1	Low priority

Table 1.6 Constants used within the ST20

Product identity values

These are values returned by the *lddevid* and *ldprodid* instructions. For specific product ids in the ST20 family refer to SGS-THOMSON.

1.10 Operators used in the definitions

Modulo operators

Arithmetic on addresses 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 address map produces the address of the byte at the bottom of the address map. There is also a number of instructions for performing modulo arithmetic, such as *sum*, *prod*, etc. 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 *div* (*divide*) instruction indicates the cases that can cause overflow, independently of the actual division:

> **if** (Areg = 0) **or** ((Breg = MostNeg) **and** (Areg = -1))
> {
>   Areg′ ← *undefined*
>   *IntegerOverflow*
> }
> **else**
>   Areg′ ← Breg / Areg
>
> Breg′ ← Creg
> Creg′ ← *undefined*

Checked operators

To simplify the description of *checked* arithmetic, the operators '$+_{checked}$', '$-_{checked}$', etc. are used to indicate operations that perform checked arithmetic on signed integers. These operators signal an *IntegerOverflow* if an overflow, divide by zero, or other arithmetic error occurs. If no trap is taken, the operators also deliver the modulo result.

A number of comparison operators are also used and there are versions of some of these that treat the operands as unsigned integers. A full list of the operators used in the instruction definitions is given in Table 1.7.

1.11 Functions

Type conversions

The following function is used to indicate a type conversion:

`unsign(x)` causes the bit-pattern in x to be interpreted as an unsigned integer.

1.12 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

Symbol	Meaning
Integer arithmetic with overflow checking	
$+_{checked}$ $-_{checked}$ $\times_{checked}$	Add, subtract, and multiply of signed integers. If the computation overflows an *IntegerOverflow* is signalled and the result of the operation is truncated to the word length.
Unchecked (modulo) integer arithmetic	
$+$ $-$ $\times$ $/$ **rem**	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. The sign of the remainder is the same as the sign of the dividend.
Signed comparison operators	
$<$ $>$ $\leq$ $\geq$ $=$ $\neq$	Comparisons of signed integer and floating point values: 'less than', 'greater than', 'less than or equal', 'greater than or equal', 'equal' and 'not equal'.
Unsigned comparison operators	
$<_{unsigned}$ $>_{unsigned}$ $\geq_{unsigned}$ **after**	Comparisons of unsigned integer values: 'less than', 'greater than', 'greater than or equal', and 'after' (for comparison of times).
Logical bitwise operations	
$\sim$ (or BITNOT) $\wedge$ (or BITAND) $\vee$ (or BITOR) $\otimes$ (or BITXOR) >> <<	'Not' (1's complement), 'and', 'or', 'exclusive or', and logical left and right shift operations on bits in words.
Boolean operators	
not **and** **or**	Boolean combination in conditionals.

Table 1.7 Operators used in the instruction descriptions

- **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)
IptrReg′ $\leftarrow$ *next instruction* + n

else
{
IptrReg′ ← *next instruction*

Areg′ ← Breg
Breg′ ← Creg
Creg′ ← *undefined*
}

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 Addressing and data representation

The ST20 processor is a 32-bit word machine, with byte addressing and a 4 Gbyte address space. This chapter explains how data is loaded from and stored into that address space, explains how signed arithmetic is represented, and defines the arithmetic significance of ordering of data items.

2.1 Word address and byte selector

A machine address 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 comparison functions to be used on pointer (address) 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. In this text, names are used to represent these and other values (e.g. *NotProcess.p*, *Disabling.p*). A full list of names and values of constants used in this book is given in section 1.9.

2.2 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 the larger of the two. Figure 2.1 shows the ordering of bytes in words and memory for the ST20. Note that this ordering is compatible with Intel processors, but not Motorola or SPARC.

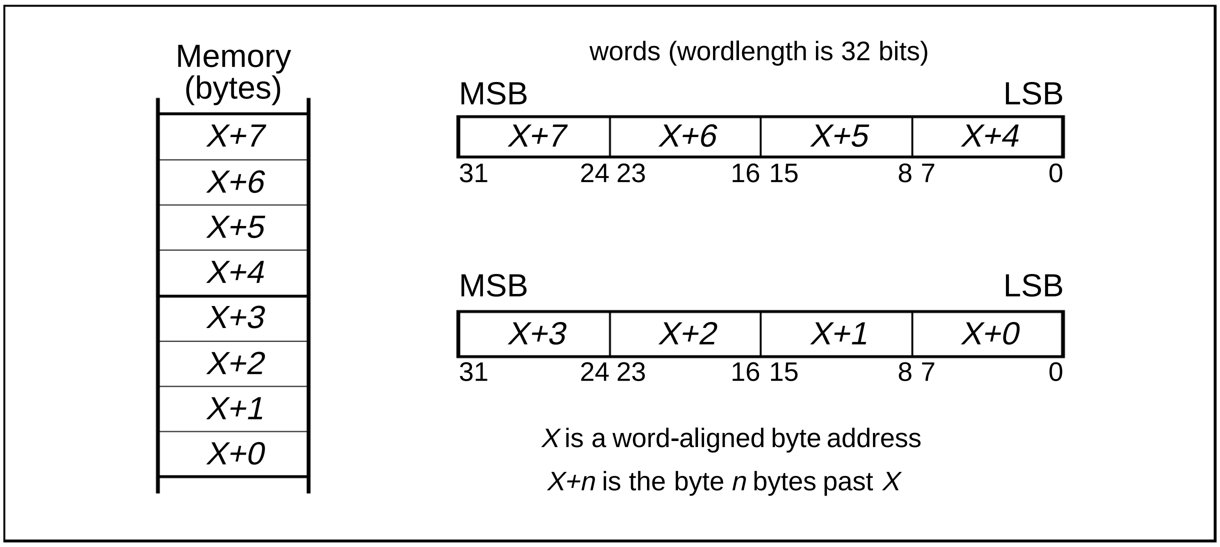

Figure 2.1 Bytes in memory and words

Most instructions that involve fetching data from or storing data into memory, use word aligned addresses (i.e. bits 1 and 0 are set to 0) and load or store four contiguous bytes. However, there are some instructions that can manipulate part of the bit pattern in a word, and a few that use double words.

A data item that is represented in two contiguous bytes, is referred to as a 16-bit object. This can be stored, either in the least significant 16-bits of a word location or in the most significant 16 bits, hence addresses of such locations are 16-bit aligned (i.e. bit 0 is set to 0).

A data item that is represented in two contiguous words is referred to as a 64-bit object or a double word.

Similarly, a data item represented in a single byte is sometimes referred to as an 8-bit object.

2.3 Signed integers and sign extension

A signed integer is stored in twos-complement format and may be represented by an N-bit object. Most commonly a signed integer is represented by a single word (32-bit object), 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.

Consider the example shown in Figure 2.2, which shows how the value -10 is stored in a 32-bit register, firstly as an 8-bit object and secondly as a 32-bit object. Observe that bits 31 to 8 are meaningful for a 32-bit object but not for an 8-bit object. These bits are set to 1 in the 32-bit object to preserve the negative sign of the integer being represented.

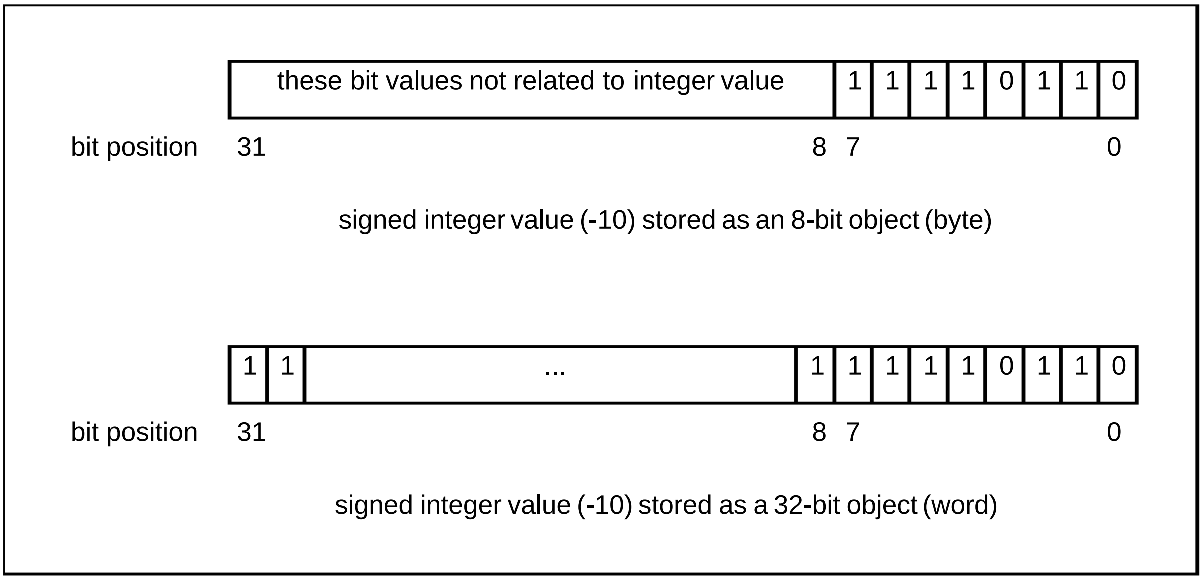

Figure 2.2 Storing a signed integer in different length objects

The length of the object that stores a signed integer can be increased (i.e. the object size can be increased). This operation is known as ‘sign extension’. 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 in fact the same as the value of the most significant bit - i.e. the sign bit - of the smaller object. The ST20 provides instructions that sign extend byte and half-word to word, and 32 bits to 64 bits.

3 Registers

3.1 Machine registers

This section introduces the ST20 processor registers that are visible to the programmer. Firstly the set of registers known as state registers are presented and discussed. These fully define the state of the executing process. Secondly the other registers of interest to the programmer, are presented.

3.1.1 Process state registers

The state of a executing process at any instant is defined by the contents of the machine registers listed in Table 3.1. The 'register' column gives the abbreviated name of the register. The 'full name / description' column provides the full textual name which is usually used when referencing a register in this manual; and where unclear, a brief description of the information contained in this register.

register	full name / description process modes
Status	status register
Wptr	workspace pointer - contains the address of the workspace of the currently executing process
Iptr	instruction pointer register - pointer to next instruction to be executed
Areg	integer stack register A
Breg	integer stack register B
Creg	integer stack register C

Table 3.1 Process state registers

In addition there is a small number of registers used to implement block moves.

Status register

The Status register contains status bits which describe the current state of the process and any errors which may have occurred. The contents of the Status register are shown in Table 3.2.

'Shadow' registers

When a high priority process interrupts a low priority process, the state of the currently executing process needs to be saved. For this purpose, two sets of process state registers are provided, one each for high and low priority. On interrupt, the processor switches to using the high priority registers, leaving the low priority registers to preserve the low priority state.

A high priority process may manipulate the low priority 'shadow' registers with the instructions *ldshadow* and *stshadow*. In the definitions of these instructions, the process state registers have a subscript (e.g. Areg[LowPriority]) indicating the priority.

bit number	full name / description
0	breakpoint trap status bit
1	integer error trap status bit
2	integer overflow trap status bit
3	illegal opcode trap status bit
4	load trap trap status bit
5	store trap trap status bit
6	internal channel trap status bit
7	external channel trap status bit
8	timer trap status bit
9	timeslice trap status bit
10	run trap status bit
11	signal trap status bit
12	process interrupt trap status bit
13	queue empty trap status bit
14	reserved
15	causeerror status bit
17-16	**Scheduler trap return priority status bits:** 00 - high priority 01 - low priority
19-18	**Trap group status bits**: 00 - Breakpoint 01 - Error 10 - System 11 - Scheduler
20	timeslice enable bit
25-21	reserved
30-26	**Interrupted operation status bits**: 00000 - None 00001 - *move* 00010 - *devmove* 00011 - *move2dall* 00100 - *move2dzero* 00101 - *move2dnonzero* 00110 - *in* 00111 - *out* 01000 - *tin* 01001 - *tin* restart 01010 - *taltwt* 01011 - *taltwt* restart 01100 - *dist* 01101 - *dist* restart 01110 - *enbc* 01111 - *disc* 10000 - *resetch*
31	status valid

Table 3.2 Status register

If the process state registers are referred to without subscripts then the current priority is implied.

3.1.2 Other machine registers

There are several other registers which the programmer should know about, but which are not part of the process state. These are presented in Table 3.3.

register	full name / description
ProcQueueFPtr[0]	high priority front pointer register - contains pointer to first process on the high priority scheduling list
ProcQueueFPtr[1]	low priority front pointer register - contains pointer to first process on the low priority scheduling list
ProcQueueBPtr[0]	high priority back pointer register - contains pointer to last process on the high priority scheduling list
ProcQueueBPtr[1]	low priority back pointer register - contains pointer to last process on the low priority scheduling list
ClockReg[0]	high priority clock register - contains current value of high priority clock
ClockReg[1]	low priority clock register - contains current value of low priority clock
TptrReg[0]	high priority timer list pointer register - contains pointer to the first process on the high priority timer list
TptrReg[1]	low priority timer list pointer register - contains pointer to the first process on the low priority timer list
TnextReg[0]	high priority alarm register - contains the time of the first process on the high priority timer queue
TnextReg[1]	low priority alarm register - contains the time of the first process on the low priority timer queue
Enables	trap and global interrupt enables register

Table 3.3 Other machine registers

Enables register

The Enables register contains:

- *TrapEnables* bits (0..15) which can be used to control the taking of traps;
- *GlobalInterruptEnables* bits (16..31) which are used to control timeslicing and interruptibility. These are normally set to 1.

Bits of *TrapEnables* may be set using the *trapenb* instruction and cleared using *trapdis*. Bits of *GlobalInterruptEnables* may be set using the instruction *gintenb* and disabled using *gintdis*.

The contents of the Enables register are shown in Table 3.4.

ClockEnables

ClockEnables is a pair of flags which enable the timers **ClockReg** to tick. Bit zero of *ClockEnables* controls **ClockReg[0]** and bit 1 controls **ClockReg[1]**. In each case,

bit number	full name / description
0	breakpoint trap enable bit
1	integer error trap enable bit
2	integer overflow trap enable bit
3	illegal opcode trap enable bit
4	load trap trap enable bit
5	store trap trap enable bit
6	internal channel trap enable bit
7	external channel trap enable bit
8	timer trap enable bit
9	timeslice trap enable bit
10	run trap enable bit
11	signal trap enable bit
12	process interrupt trap enable bit
13	queue empty trap enable bit
15-14	reserved
16	low priority process interrupt enable bit
17	low priority timeslice enable bit
18	low priority external event enable bit
19	low priority timer alarm enable bit
20	high priority process interrupt enable bit
21	high priority timeslice enable bit
22	high priority external event enable bit
23	high priority timer alarm enable bit
31-24	reserved

Table 3.4 Enables register

the timer will tick if the *ClockEnables* bit is set to1. *ClockEnables* can be set using the *clockenb* instruction and cleared using *clockdis*.

Error flags

The other machine flags referred to in the instruction definitions are listed in Table 3.4.

flag name	description
ErrorFlag	Untrapped arithmetic error flags
HaltOnErrorFlag	Halt the processor if the *ErrorFlag* is set

Table 3.5 Error flags

ErrorFlag is a pair of flags, one for each priority, set by the processor if an integer error or integer overflow error occurs and the corresponding trap is not enabled. The processor will immediately halt if the *HaltOnError* flag is also set, or will continue otherwise. The *ErrorFlag*s may also be set by the *seterr* instruction or tested and cleared by the *testerr* instruction. The *stoperr* instruction stops the current process if the *ErrorFlag* is set. The low priority *ErrorFlag* is copied to the high priority when the processor switches from low to high priority. The *HaltOnError* flag may be set by the *sethalterr* instruction, cleared by *clrhalterr* and tested by *testhalterr*.

3.2 The process descriptor and its associated register fields

In order to identify a process completely it is necessary to know: its workspace address (in which the byte selector is always 0), and its priority (high or low). This information is contained in the process descriptor. The workspace address of the currently executing process is held in the workspace pointer register (**Wptr**) and the priority is held in the flag **Priority**.

Wptr points to the current process workspace, which is always word-aligned. **Priority** is the priority of the currently executing process where the value 1 indicates low priority and 0 indicates high priority.

The process descriptor is formed from a pointer to the process workspace **or**-ed with the priority flag at bit 0. Bit 1 is always set to 0.

Wdesc is defined so that the following invariants are obeyed:

$$\textbf{Wptr} = \textbf{Wdesc} \wedge \textit{WordSelectMask}$$

$$\textbf{Priority} = \textbf{Wdesc} \wedge 1$$

Workspace address	0	Priority
31 … 2	1	0

Figure 3.1 Constituents of a process descriptor

4 Instruction representation

The instruction encoding is designed so that the most commonly executed instructions occupy the least number of bytes. This chapter describes the encoding mechanism and explains how it achieves this.

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 users (working at the level of microprocessor assembly language or high-level languages) need not be aware of the existence of instruction components and do not need to think about the encoding. The first section (4.1) has been included to provide a background. The following section (4.2) need only concern the reader that wants to implement a code generator.

4.1 Instruction encoding

4.1.1 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 are a *function code*, and the four least significant bits are used to build an *instruction data value* as shown in Figure 4.1.

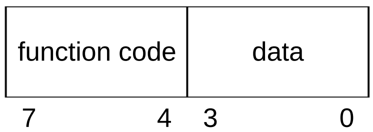

Figure 4.1 Instruction format

The representation provides for sixteen instruction components (one for each function), each with a data field ranging from 0 to 15.

There are three categories of instruction component. Firstly there are those that specify the instruction directly in the function field. These are used to implement *primary instructions*. Secondly there are the instruction components that are used to extend the instruction data value - this process of extension is referred to as *prefixing*. Thirdly there is the instruction component *operate* (*opr*) which specifies the instruction indirectly using the *instruction data value*. *opr* is used to implement *secondary instructions*.

4.1.2 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.

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. However two instruction components are used to extend the range of the instruction data value. Hence one or more prefixing components may be needed to create the correct instruction data value. These are shown in Table 4.1 and explained below.

mnemonic	**name**
pfix n	prefix
nfix n	negative prefix

Table 4.1 Prefixing instruction components

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

pfix loads its four data bits into the instruction data value, and then shifts this value up four places. *nfix* is similar, except that it complements the instruction data value† before shifting it up. Consequently, a sequence of one or more prefixes can be included to extend the value. Instruction data values in the range -256 to 255 can be represented using one prefix instruction.

When the processor encounters an instruction component other than *pfix* or *nfix*, it loads the data field into the instruction data value. The instruction encoding is now complete and the instruction can be executed. When the processor is ready to fetch the next instruction component, it starts to create a new instruction data value.

4.1.3 Primary Instructions

Research has shown that computers spend most of the time executing 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 therefore, 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 will therefore normally include a this operand - n - when referenced.

The mnemonics and names for the primary instructions are listed in Table 4.2.

mnemonic	**name**
adc n	add constant

Table 4.2 Primary instructions

† Note that it inverts *all* 32 bits of the instruction data value.

mnemonic	name
ajw n	adjust workspace
call n	call
cj n	conditional jump
eqc n	equals constant
j n	jump
ldc n	load constant
ldl n	load local
ldlp n	load local pointer
ldnl n	load non-local
ldnlp n	load non-local pointer
stl n	store local
*stn*l n	store non-local

Table 4.2 Primary instructions

4.1.4 Secondary instructions

The ST20 encodes all other instructions (secondary instructions) indirectly using the instruction data value.

mnemonic	name
opr	operate

Table 4.3 Operate instruction

The instruction component *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 integer stack. This allows a further 16 operations to be encoded in a single byte instruction. However the prefix instructions can be used to extend the instruction data value, allowing any number of operations to be performed.

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 frequent secondary instructions are represented without using prefix instructions.

4.1.5 Summary of encoding

The encoding mechanism has important consequences.

- Firstly, 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.
- Secondly, it allows these operands to be represented in a form independent of

the word-length of the processor.

- Thirdly, it enables any number of secondary instructions to be implemented.

The following provides some simple examples of encoding:

- The instruction *ldc* 17 is encoded with the sequence:

  pfix 1; *ldc* 1

- The instruction *add* is encoded by:

  opr 5

- The instruction *and* is encoded by:

  opr 70

  which is in turn encoded with the sequence:

  pfix 4; *opr* 6

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 in the above examples, these are just shown as: *ldc 17*, *add*, and *and*. (Also, where appropriate, an expression may be placed in a code sequence to represent the code needed to evaluate that expression.

4.2 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.

4.2.1 Prefixing a constant

The algorithm to generate a constant instruction data value *e* for a function *op* is described by the following recursive function.

```
prefix( op, e ) = IF
                    e < 16 AND e    0
                       op( e )
                    e    16
                        prefix( pfix, e >> 4 ); op( e ∧  # F )
                    e <  0
                        prefix( nfix, (~e)  >>  4 ); op( e ∧  # F )
```

where *(op, e)* is the instruction component with function code *op* and data field *e*, ~ is a bitwise NOT, and >> is a logical right shift.

4.2.2 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.

1. 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.
2. Evaluate all jump and load offsets under the current assumptions of the size of prefix sequences to the jumps and offset loads
3. 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.†
4. 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.

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.

† 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 3 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.

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 be considered as an unshrinkable block of code.

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.

5 Instruction Set Reference

adc n — add constant

Code: Function 8

Description: Add a constant to **Areg,** with checking for overflow.

Definition:

Areg′ $\leftarrow$ Areg $+_{\text{checked}}$ n

Error signals:

IntegerOverflow can be signalled by $+_{\text{checked}}$

Comments:

Primary instruction.

See also: *add ldnlp sum*

add add

Code: F5

Description: Add **Areg** and **Breg**, with checking for overflow.

Definition:

Areg′ ← Breg $+_{\text{checked}}$ Areg

Breg′ ← Creg
Creg′ ← *undefined*

Error signals:

IntegerOverflow can be signalled by $+_{\text{checked}}$

Comments:

Secondary instruction.

See also: *adc sum*

ajw n　　adjust workspace

Code: Function B

Description: Move the workspace pointer by the number of words specified in the operand, in order to allocate or de-allocate the workspace stack.

Definition:

Wptr′ $\leftarrow$ Wptr @ n

Error signals: none

Comments:

Primary instruction.

See also: *call gajw*

alt alt start

Code: 24 F3

Description: Start of a non-timer alternative sequence. The **pw.State** location of the workspace is set to *Enabling.p.*

Definition:

word′[Wptr @ pw.State] ← Enabling.p
Enter alternative sequence

Error signals: none

Comments:

Secondary instruction.

See also: *altend altwt disc diss dist enbc enbs enbt talt taltwt*

altend alt end

Code: 24 F5

Description: End of alternative sequence. Jump to start of selected process.

Definition:

Terminate alternative sequence
Iptr′ ← *next instruction* + word [Wptr @ pw.Temp]

Error signals: none

Comments:

Secondary instruction.
Uses the **pw.Temp** slot in the process workspace.

See also: *alt altwt disc diss dist enbc enbs enbt talt taltwt*

altwt alt wait

Code: 24 F4

Description: Wait until one of the enabled guards of an alternative has become ready, and initialize workspace for use during the disabling sequence.

Definition:

if (word[Wptr @ pw.State] ≠ Ready.p)
{
word′[Wptr @ pw.State] ← Waiting.p
Deschedule process and wait for one of the guards to become ready
}

word′[Wptr @ pw.Temp] ← NoneSelected.o

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.
Instruction is a descheduling point.
Uses the **pw.Temp** slot in the process workspace.

See also: *alt altend disc diss dist enbc enbs enbt talt taltwt*

and and

Code: 24 F6

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

Definition:

Areg′ ← Breg $\wedge$ Areg

Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *not or xor*

bcnt byte count

Code: 23 F4

Description: Produce the length, in bytes, of a multiword data object. Converts the value in **Areg**, representing a number of words, to the equivalent number of bytes.

Definition:

$\text{Areg}' \leftarrow \text{Areg} \times \text{BytesPerWord}$

Error signals: none

Comments:

Secondary instruction.

bitcnt count bits set in word

Code: 27 F6

Description: Count the number of bits set in **Areg** and add this to the value in **Breg**.

Definition:

Areg′ ← Breg + *number of bits set to 1 in* Areg

Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

bitrevnbits reverse bottom n bits in word

Code: 27 F8

Description: Reverse the order of the bottom **Areg** bits of **Breg**.

Definition:

if (0 ≤ Areg) **and** (Areg ≤ BitsPerWord)
{
$\quad Areg'_{0..Areg-1} \leftarrow$ *reversed* $Breg_{0..Areg-1}$
$\quad Areg'_{Areg..BitsPerWord-1} \leftarrow 0$
}
else
$\quad$ *Undefined effect*

Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.
The effect of the instruction is undefined if the number of bits specified is more than the word length.

See also: *bitrevword*

bitrevword reverse bits in word

Code: 27 F7

Description: Reverse the order of all the bits in **Areg**.

Definition:

Areg′ $\leftarrow$ *reversed* Areg

Error signals: none

Comments:

Secondary instruction.

See also: *bitcnt bitrevnbits*

bsub byte subscript

Code: F2

Description: Generate the address of the element which is indexed by **Breg**, in the byte array pointed to by **Areg**.

Definition:

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

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

Error signals: none

Comments:
Secondary instruction.

See also: *ssub sum wsub wsubdb*

call n call

Code: Function 9

Description: Adjust workspace pointer, save evaluation stack, and call subroutine at specified byte offset.

Definition:

Wptr′ $\leftarrow$ Wptr @ -4

word′[Wptr′ @ 0] $\leftarrow$ Iptr
word′[Wptr′ @ 1] $\leftarrow$ Areg
word′[Wptr′ @ 2] $\leftarrow$ Breg
word′[Wptr′ @ 3] $\leftarrow$ Creg

Iptr′ $\leftarrow$ *next instruction* + n

Areg′ $\leftarrow$ Iptr
Breg′ $\leftarrow$ *undefined*
Creg′ $\leftarrow$ *undefined*

Error signals: none

Comments:

Primary instruction.

See also: *ajw gcall ret*

causeerror cause error

Code: 62 FF

Description: Take a trap with the trap type set to the value in **Areg**. Only one bit in **Areg** should be set; the position of the bit indicates the trap to be signalled. When the trap is taken the causeerror bit in the status register is set to indicate a user generated trap.

In the case of a scheduler trap being set then the bottom bit of **Breg** is used to determine the priority of the scheduler trap. In addition when the scheduler trap is trapping a process scheduling operation (e.g. the *run* trap) the **Breg** is interpreted as the process descriptor to be scheduled.

Definition:

if (Areg=2^i) **and** (*trap type* i *is enabled*)
{
set causeerror bit in Status
cause trap type i
}
else
Undefined effect

Error signals: The causeerror bit is set and the indicated trap signalled if enabled.

Comments:

Secondary instruction.
Sets traps independently of trap enables state.

See also: *tret sttraph ldtrap*

cb check byte

Code: 2B FA

Description: Check that the value in **Areg** can be represented as an 8-bit signed integer.

Definition:

if (Areg $< -2^7$) **or** (Areg $\geq 2^7$)
IntegerError

Error signals:

IntegerError signalled if **Areg** is not in range.

Comments:

Secondary instruction.

See also: *cbu cir ciru cs csu*

cbu check byte unsigned

Code: 2B FB

Description: Check that the value in **Areg** can be represented as an 8-bit unsigned integer.

Definition:

if (Areg < 0) **or** (Areg $\geq 2^8$)
IntegerError

Error signals:
IntegerError signalled if **Areg** is not in range.

Comments:
Secondary instruction.

See also: *cb cir ciru cs csu*

ccnt1 check count from 1

Code: 24 FD

Description: Check that **Breg** is in the range 1..**Areg** inclusive, interpreting **Areg** and **Breg** as unsigned numbers.

Definition:

if (Breg = 0) **or** ($\text{Breg}_{\text{unsigned}} > \text{Areg}_{\text{unsigned}}$)
    IntegerError

Areg′ ← Breg
Breg′ ← Creg
Creg′ ← *undefined*

Error signals:
IntegerError signalled if **Areg** is not in range.

Comments:
Secondary instruction.

See also: *csub0*

cflerr check floating point error

Code: 27 F3

Description: Checks if **Areg** represents an Inf or NaN.

Definition:

if (Areg $\wedge$ #7F800000 = #7F800000)
IntegerError

Error signals:

IntegerError signalled if Areg represents an Inf or NaN.

Comments:

Secondary instruction.

See also: *unpacksn roundsn postnormsn ldinf*

cir check in range

Code: 2C F7

Description: Check that **Creg** is in the range **Areg**..**Breg** inclusive.

Definition:

if (Creg < Areg) **or** (Creg > Breg)
IntegerError

Areg′ ← Creg
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals:

IntegerError signalled if **Creg** is not in range.

Comments:

Secondary instruction.

See also: *ciru*

ciru check in range unsigned

Code: 2C FC

Description: Check that **Creg** is the range **Areg**..**Breg** inclusive, treating all as unsigned values.

Definition:

if ($Creg_{unsigned}$ < $Areg_{unsigned}$) **or** ($Creg_{unsigned}$ > $Breg_{unsigned}$)
  IntegerError

$Areg' \leftarrow Creg$
$Breg' \leftarrow$ *undefined*
$Creg' \leftarrow$ *undefined*

Error signals: *IntegerError* signalled if **Creg** is not in range.

Comments:
Secondary instruction.

See also: *cir*

cj n 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.

Definition:

if (Areg = 0)
    Iptr′ ← *next instruction* + n
else
{
    Iptr′ ← *next instruction*
    Areg′ ← Breg
    Breg′ ← Creg
    Creg′ ← *undefined*
}

Error signals: none

Comments:

Primary instruction.

See also: *j lend*

clockdis clock disable

Code: 64 FE

Description: Stops the clocks specified in bits 0 and 1 of **Areg** where bit 0 indicates the high priority clock and bit 1 the low priority clock. The original values of these two clock enable bits are returned in **Areg**.

Definition:

$Areg'_{1..0} \leftarrow ClockEnables$
$Areg'_{31..2} \leftarrow 0$
$ClockEnables' \leftarrow ClockEnables \wedge \sim Areg$

Error signals: none

Comments:

Secondary instruction.

See also: *clockenb*

clockenb clock enable

Code: 64 FF

Description: Starts or restarts the clocks specified in bits 0 and 1 of **Areg**, where bit 0 indicates the high priority clock and bit 1 indicates the low priority clock. The original values of these two clock enable bits are returned in **Areg**.

Definition:

$Areg'_{1..0}$	$\leftarrow$	ClockEnables
$Areg'_{31..2}$	$\leftarrow$	0
ClockEnables′	$\leftarrow$	ClockEnables $\vee$ Areg

Error signals: none

Comments:

Secondary instruction.

See also: *clockdis*

clrhalterr clear halt-on-error

Code: 25 F7

Description: Clear the HaltOnError flag.

Definition:

HaltOnErrorFlag′ $\leftarrow$ *clear*

Error signals: none

Comments:

Secondary instruction.

See also: *sethalterr testhalterr*

crcbyte calculate CRC on byte

Code: 27 F5

Description: Generate a CRC (cyclic redundancy check) checksum from the most significant byte of **Areg**. **Breg** contains the previously accumulated checksum and **Creg** the polynomial divisor (or ‘generator’). The new CRC checksum, the polynomial remainder, is calculated by repeatedly (8 times) shifting the accumulated checksum left, shifting in successive bits from the **Areg** and if the bit shifted out of the checksum was a 1, then the generator is exclusive-**or**ed into the checksum.

Definition:

$Areg' \leftarrow temp(8)$
$Breg' \leftarrow Creg$
$Creg' \leftarrow$ *undefined*

where
temp(0) = Breg
for i = 1 .. 8
$$temp(i) = (temp(i\ -1) \ll 1) + Areg_{BitsPerWord-i}) \otimes (Creg \times temp(i\ -1)_{BitsPerWord-1})$$

Error signals: none

Comments:
Secondary instruction.

See also: *crcword*

crcword calculate CRC on word

Code: 27 F4

Description: Generate a CRC (cyclic redundancy check) checksum from **Areg**. **Breg** contains the previously accumulated checksum and **Creg** the polynomial divisor (or 'generator'). The new CRC checksum, the polynomial remainder, is calculated by repeatedly (BitsPerWord times) shifting the accumulated checksum left, shifting in successive bits from the **Areg** and if the bit shifted out of the checksum was a 1, then the generator is exclusive-**or**ed into the checksum.

Definition:

$Areg' \leftarrow temp(BitsPerWord)$
$Breg' \leftarrow Creg$
$Creg' \leftarrow$ *undefined*

where

$temp(0) = Breg$
for $i = 1 \,..\, 32$
$temp(i) = (temp(i\ \text{-}1) << 1) + Areg_{BitsPerWord\text{-}i})$
$\otimes\ (Creg \times temp(i - 1)_{BitsPerWord\text{-}1})$

Error signals: none

Comments:

Secondary instruction.

See also: *crcbyte*

cs check sixteen

Code: 2F FA

Description: Check that the value in **Areg** can be represented as a 16-bit signed integer.

Definition:

if (Areg $< -2^{15}$) **or** (Areg $\geq 2^{15}$)
IntegerError

Error signals:

IntegerError signalled if **Areg** is not in range.

Comments:

Secondary instruction.

See also: *cb cbu cir ciru csngl csu cword*

csngl check single

Code: 24 FC

Description: Check that the two word signed value in **Areg** and **Breg** (most significant word in **Breg**) can be represented as a single length signed integer.

Definition:

if ((Areg $\geq$ 0) **and** (Breg $\neq$ 0)) **or** ((Areg < 0) **and** (Breg $\neq$ –1))
IntegerError

Error signals:

IntegerError signalled if **Areg** is not in range.

Comments:

Secondary instruction.

See also: *cb cbu cir ciru cs csu cword*

csu check sixteen unsigned

Code: 2F FB

Description: Check that the value in **Areg** can be represented as a 16-bit unsigned integer.

Definition:

if (Areg $<$ 0) **or** (Areg $\geq 2^{16}$)
IntegerError

Error signals:

IntegerError signalled if **Areg** is not in range.

Comments:

Secondary instruction.

See also: *cb cbu cir ciru cs csngl cword*

csub0 check subscript from 0

Code: 21 F3

Description: Check that **Breg** is in the range 0..(**Areg**-1), interpreting **Areg** and **Breg** as unsigned numbers.

Definition:

if ($Breg_{unsigned} \geq Areg_{unsigned}$)
Integer Error

Areg′ ← Breg
Breg′ ← Creg
Creg′ ← *undefined*

Error signals:
IntegerError signalled if **Breg** is not in range.

Comments:
Secondary instruction.

See also: *ccnt1*

cword check word

Code: 25 F6

Description: Check that the value in **Breg** can be represented as an N-bit signed integer. **Areg** contains $2^{(N-1)}$ to indicate the value of N (i.e. bit N-1 of **Areg** is set to 1 and all other bits are set to zero).

Definition:

if (Breg < –Areg) **or** (Breg ≥ Areg)
IntegerError

Areg′ ← Breg
Breg′ ← Creg
Creg′ ← *undefined*

Error signals:

IntegerError signalled if **Breg** is not in range.

Comments:

The result of the instruction is undefined if **Areg** is not an integral power of 2.
Undefined if **Areg** has more than one bit set.
Secondary instruction.

See also: *cb cs csngl xword*

devlb device load byte

Code: 2F F0

Description: Perform a device read from memory, a memory-mapped device or a peripheral. The byte addressed by **Areg** is read into **Areg** as an unsigned value. The memory access performed by this instruction is guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed after all normal memory load instructions that appear before it in the code sequence, and before all normal memory loads that appear later.

Definition:

if (PeripheralStart $\leq$ Areg $\leq$ PeripheralEnd)
  $Areg'_{0..7}$ $\leftarrow$ PeripheralByte[Areg]
else
  $Areg'_{0..7}$ $\leftarrow$ byte[Areg]

$Areg'_{8..31}$ $\leftarrow$ 0

Error signals: none

Comments:

Secondary instruction.

See also: *devls devlw devsb lb*

devls device load sixteen

Code: 2F F2

Description: Perform a device read from memory, a memory-mapped device or a peripheral. The 16-bit object addressed by **Areg** is read into **Areg** as an unsigned value. The memory access performed by this instruction is guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed after all normal memory load instructions that appear before it in the code sequence, and before all normal memory loads that appear after it.

Definition:

if (PeripheralStart ≤ Areg ≤ PeripheralEnd)

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

else

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

$Areg'_{16..31} \leftarrow 0$

Error signals: none

Comments:

Secondary instruction.

See also: *devlb devlw devsb ls*

devlw device load word

Code: 2F F4

Description: Perform a device read from memory, a memory-mapped device or a peripheral. The word addressed by **Areg** is read into **Areg**. The memory access performed by this instruction is guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed after all normal memory load instructions that appear before it in the code sequence, and before all normal memory loads that appear after it.

Definition:

if (PeripheralStart ≤ Areg ≤ PeripheralEnd)
  Areg′ ← PeripheralWord[Areg]
else
  Areg′ ← word[Areg]

Error signals: none

Comments:
Secondary instruction.

See also: *devlb devls devsw ldnl*

devmove device move

Code: 62 F4

Description: Perform a device copy between memory or memory-mapped devices. Copies **Areg** bytes to address **Breg** from address **Creg**. Only the minimum number of reads and writes required to copy the data will be performed. Each read will be to a strictly higher (more positive) address than the one before and each write will be to a strictly higher byte address than the one before. There is no guarantee of the relative ordering of read and write cycles, except that a write cannot occur until the corresponding read has been performed. The memory accesses performed by this instruction are guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed after all normal memory access instructions that appear before it in the code sequence, and before all normal memory accesses that appear after it.

Definition:

if (*source and destination overlap*)
  Undefined effect
else for i = 0 .. ($Areg_{unsigned}$ − 1)
  byte′[Breg + i] ← byte[Creg + i]

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.
The effect of the instruction is undefined if the source and destination overlap.
Instruction is interruptible.
Devmove will not operate from or to peripheral addresses.

See also: *move*

devsb device store byte

Code: 2F F1

Description: Perform a device write from memory, a memory-mapped device or a peripheral. Store the least significant byte of **Breg** into the byte addressed by **Areg**. The memory access performed by this instruction is guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed after all normal memory store instructions that appear before it in the code sequence, and before all normal memory stores that appear after it.

Definition:

if (PeripheralStart ≤ Areg ≤ PeripheralEnd)
  PeripheralByte′[Areg] ← $\text{Breg}_{0..7}$
else
  byte′[Areg] ← $\text{Breg}_{0..7}$

Areg′ ← Creg
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:
Secondary instruction.

See also: *devlb devss devsw sb*

devss device store sixteen

Code: 2F F3

Description: Perform a device write from memory, a memory-mapped device or a peripheral. Store bits 0..5 of **Breg** into the sixteen bits addressed by **Areg**. A memory access performed by this instruction is guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed after all normal memory store instructions that appear before it in the code sequence, and before all normal memory stores that appear after it.

Definition:

if (PeripheralStart ≤ Areg ≤ PeripheralEnd)
  PeripheralSixteen′[Areg] ← $\text{Breg}_{0..15}$
else
  sixteen′[Areg] ← $\text{Breg}_{0..15}$

Areg′ ← Creg
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:
Secondary instruction.

See also: *devls devsb devsw ss*

devsw device store word

Code: 2F F5

Description: Perform a device write from memory, a memory-mapped device or a peripheral. Store **Breg** into the word of memory addressed by **Areg**. The memory access performed by this instruction is guaranteed to be correctly sequenced with respect to other device-access instructions. Also the instruction is guaranteed to be executed after all normal memory store instructions that appear before it in the code sequence, and before all normal memory stores that appear after it.

Definition:

if (PeripheralStart ≤ Areg ≤ PeripheralEnd)
  PeripheralWord′[Areg] ← Breg
else
  word′[Areg] ← Breg

Areg′ ← Creg
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:
Secondary instruction.

See also: *devlw devsb devss stnl*

diff difference

Code: F4

Description: Subtract **Areg** from **Breg**, without checking for overflow.

Definition:

Areg′ ← Breg – Areg

Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *sub*

disc disable channel

Code: 22 FF

Description: Disable a channel guard in an alternative sequence. **Areg** is the offset from the byte following the *altend* to the start of the guarded process, **Breg** is the boolean guard and **Creg** is a pointer to the channel. If this is the first ready guard then the value in **Areg** is stored in workspace and **Areg** is set to *true*, otherwise **Areg** is set to *false*. Note that this instruction should be used as part of an alternative sequence following an *altwt* or *taltwt* instruction.

Definition:

```
if (Breg = false)                                   – boolean guard is false
    Areg′  ←  false
else if (Creg is internal channel)
{
    if (word[Creg] = NotProcess.p)                  – guard already disabled
        Areg′  ←  false
    else if (word[Creg] = Wdesc)                    – this guard is not ready
    {
        word′[Creg]     ←  NotProcess.p
        Areg′           ←  false
    }
    else if (word[Wptr @ pw.Temp] = NoneSelected.o)
    {                                               – this is the first ready guard
        word′[Wptr @ pw.Temp]    ←  Areg
        Areg′                    ←  true
    }
    else                                            – a previous guard selected
        Areg′  ←  false
}
else if (Creg is external channel)
{
    Disable comms subsystem and receive status
    if (channel not ready)                          – channel not waiting
        Areg′  ←  false
    else if (word[Wptr @ pw.Temp] = NoneSelected.o)
    {                                               – this is the first ready guard
        word′[Wptr @ pw.Temp]    ←  Areg
        Areg′                    ←  true
    }
    else                                            - a previous guard selected
        Areg′  ←  false
}

Breg′  ←  undefined
Creg′  ←  undefined
```

Error signals: none

Comments:

Secondary instruction.

Uses the **pw.Temp** slot in the process workspace.

See also: *alt altend altwt enbc talt taltwt*

diss disable skip

Code: 23 F0

Description: Disable a ‘skip’ guard in an alternative sequence. **Areg** is the offset from the byte following the *altend* to the start of the guarded process and **Breg** is the boolean guard. If this is the first ready guard then the value in **Areg** is stored in workspace and **Areg** is set to *true*, otherwise **Areg** is set to *false*. Note that this instruction should be used as part of an alternative sequence following an *altwt* or *taltwt* instruction.

Definition:

```
if (Breg = false)                          – boolean guard is false
    Areg′  ←  false
else if (word[Wptr @ pw.Temp] = NoneSelected.o)
    Areg′  ←  false                        – this is the first ready guard
else                                       – another guard was selected
{
    word′[Wptr @ pw.Temp]    ←  Areg
    Areg′                    ←  true
}

Breg′  ←  Creg
Creg′  ←  undefined
```

Error signals: none

Comments:

Secondary instruction.
Uses the **pw.Temp** slot in the process workspace.

See also: *alt altend altwt enbs talt taltwt*

dist disable timer

Code: 22 FE

Description: Disable a timer guard in an alternative sequence. **Areg** is the offset from the byte following the *altend* to the start of the guarded process, **Breg** is the boolean guard and **Creg** is the time after which this guard will be ready. If this is the first ready guard then the value in **Areg** is stored in **pw.Temp**, and **Areg** is set to *true*. Note that this instruction should be used as part of an alternative sequence following a *taltwt* instruction.

Definition:

if (Breg = *false*) – *boolean guard is false*
    Areg′ ← *false*
else if (word[Wptr @ pw.TLink] = TimeNotSet.p) – *no timer is ready*
    Areg′ ← *false*
else if (word[Wptr @ pw.TLink]= TimeSet.p) – *a timer is ready*
{
    if not (word[Wptr @ pw.Time] **after** Creg) – *but not this one*
        Areg′ ← *false*
    else if (word[Wptr @ pw.Temp] = NoneSelected.o)
    – *this is the first ready guard*
    {
        word′[Wptr @ pw.Temp] ← Areg
        Areg′ ← *true*
    }
    else – *a previous guard selected*
        Areg′ ← *false*
}
else
    Areg′ ← *false*

Remove this process from the timer list

Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.
Instruction is interruptible.
Uses the **pw.Temp** slot in the process workspace.

See also: *altend enbt talt taltwt*

div divide

Code: 22 FC

Description: Divide **Breg** by **Areg**, with checking for overflow. The result when not exact is rounded towards zero.

Definition:

if (Areg = 0) **or** ((Breg = MostNeg) **and** (Areg = -1))
{
  Areg′ ← *undefined*
  IntegerOverflow
}
else
  Areg′ ← Breg / Areg

Breg′ ← Creg
Creg′ ← *undefined*

Error signals:

IntegerOverflow can be signalled.

Comments:

Secondary instruction.

See also: *rem*

dup duplicate top of stack

Code: 25 FA

Description: Duplicate the top of the integer stack.

Definition:

$Areg' \leftarrow Areg$
$Breg' \leftarrow Areg$
$Creg' \leftarrow Breg$

Error signals: none

Comments:

Secondary instruction.

See also: *pop rev*

enbc enable channel

Code: 24 F8

Description: Enable a channel guard in an alternative sequence. **Areg** is the boolean guard and **Breg** is a pointer to the channel. Note that this instruction should only be used as part of an alternative sequence following an *alt* or *talt* instruction.

Definition:

```
if (Areg ≠ false)
{
    if (Breg is internal channel)
    {
        if (word[Breg] = NotProcess.p)                  – not ready
            word′[Breg]    ←  Wdesc
        else if (word[Breg] ≠ Wdesc)                    – not previously enabled
            word′[Wptr @ pw.State]    ←  Ready.p
    }
    else if (Breg is external channel)
    {
        Request Comms Subsystem to enable external channel
        and receive current status of channel
        if (channel ready)                                          -
            word′[Wptr @ pw.State]    ←  Ready.p
    }
}

Breg′  ←  Creg
Creg′  ←  undefined
```

Error signals: none

Comments:

Secondary instruction.

See also: *alt altend altwt disc talt taltwt*

enbs enable skip

Code: 24 F9

Description: Enable a ‘skip’ guard in an alternative sequence. **Areg** is the boolean guard. Note that this instruction should only be used as part of an alternative sequence following an *alt* or *talt* instruction.

Definition:

if (Areg ≠ *false*)
    word′[Wptr @ pw.State] ← Ready.p

Error signals: none

Comments:

Secondary instruction.

See also: *alt altend altwt diss talt taltwt*

enbt enable timer

Code: 24 F7

Description: Enable a timer guard in an alternative sequence. **Areg** is the boolean guard and **Breg** is the time after which the guard may be selected. Note that this instruction should only be used as part of an alternative sequence following a *talt* instruction; in this case the location **pw.State** will have been initialized to *Enabling.p* and the **pw.Tlink** slot initialized to *TimeNotSet.p*.

Definition:

```
if (Areg ≠ false)
{
    if (word[Wptr @ pw.TLink] = TimeNotSet.p)        – this is the first enbt
    {
        word′[Wptr @ pw.TLink]    ←  TimeSet.p
        word′[Wptr @ pw.Time]     ←  Breg
    }
    else if (word[Wptr @ pw.TLink] = TimeSet.p)      – this is not the first enbt
    {
        if (word[Wptr @ pw.Time] after Breg)          – this enbt has earlier time
            word′[Wptr @ pw.Time] ←  Breg
    }
}

Breg′  ←  Creg
Creg′  ←  undefined
```

Error signals: none

Comments:
Secondary instruction.

See also: *altend dist talt taltwt*

endp end process

Code: F3

Description: Synchronize the termination of a parallel construct. When all branches have executed an *endp* instruction a 'successor' process then executes. **Areg** points to the workspace of this successor process. This workspace contains a data structure which holds the instruction pointer of the successor process and the number of processes still active.

Definition:

```
if (word[Areg @ pp.Count] = 1)
{
    Iptr′    ←  word[Areg @ pp.IptrSucc]
    Wptr′    ←  Areg
}
else
    word′[Areg @ pp.Count]    ←  word[Areg @ pp.Count]–1
    start next process

Areg′  ←  undefined
Breg′  ←  undefined
Creg′  ←  undefined
```

Error signals: none

Comments:

Secondary instruction.
Instruction is a descheduling point.

See also: *startp stopp*

eqc n equals constant

Code: Function C

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

Definition:

if (Areg = n)
Areg′ ← *true*
else
Areg′ ← *false*

Error signals: none

Comments:

Primary instruction.

fmul fractional multiply

Code: 27 F2

Description: Multiply **Areg** by **Breg** treating the values as fractions, rounding the result. The values in **Areg** and **Breg** are interpreted as fractions in the range greater than or equal to -1 and less than 1 - i.e. the integer values divided by $2^{BitsPerWord-1}$. The result is rounded. The rounding mode used is analogous to IEEE round nearest; that is the result produced is the fraction which is nearest to the exact product, and, in the event of the product being equidistant between two factions, the fraction with least significant bit 0 is produced.

Definition:

if (Areg = MostNeg) **and** (Breg = MostNeg) – *MostNeg interpreted as -1*
{
  Areg′ ← *undefined*
  IntegerOverflow
}
else
  Areg′ ← (Breg × Areg) $/_{roundnearest}$ $2^{BitsPerWord-1}$

Breg′ ← Creg
Creg ← *undefined*

Error signals:

IntegerOverflow can occur.

Comments:

Secondary instruction.

fptesterr test for FPU error

Code: 29 FC

Description: Test for an error in the FPU, if present. This instruction always returns *true* on a processor without an FPU.

Definition:

Areg′ ← *true*
Breg′ ← Areg
Creg′ ← Breg

Error signals: none

Comments:

Secondary instruction.

gajw general adjust workspace

Code: 23 FC

Description: Set the workspace pointer to the address in **Areg**, saving the previous value in **Areg**.

Definition:

Wptr′ ← Areg
Areg′ ← Wptr

Error signals: none

Comments:

Secondary instruction.
Areg should be word aligned.

See also: *ajw call gcall*

gcall general call

Code: F6

Description: Jump to the address in **Areg**, saving the previous address in **Areg**.

Definition:

Iptr′ ← Areg
Areg′ ← Iptr

Error signals: none

Comments:

Secondary instruction.

See also: *ajw call gajw ret*

gintdis global interrupt disable

Code: 2C FD

Description: Disable the global interrupt events specified in the bit mask in **Areg**. This allows parts of the built-in scheduler, such as response to external events, timeslicing etc., to be disabled by software. The original value of the global interrupt enable register is returned in **Areg**.

Definition:

$\text{GlobalInterruptEnables}' \leftarrow \text{GlobalInterruptEnables} \wedge \sim\text{Areg}_{7..0}$

$\text{Areg}'_{7..0} \leftarrow \text{GlobalInterruptEnables}$

$\text{Areg}'_{31..8} \leftarrow 0$

Error signals: none

Comments:

Secondary instruction.

See also: *gintenb*

gintenb global interrupt enable

Code: 2C FE

Description: Enable the global interrupt events specified in the bit mask in **Areg.**

Definition:

GlobalInterruptEnables′ $\leftarrow$ GlobalInterruptEnables $\vee$ $\text{Areg}_{7..0}$

$\text{Areg}'_{7..0}$ $\leftarrow$ GlobalInterruptEnables

$\text{Areg}'_{8..31}$ $\leftarrow$ 0

Error signals: none

Comments:

Secondary instruction.

See also: *gintdis*

gt greater than

Code: F9

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′ ← *undefined*

Error signals: none

Comments:
Secondary instruction.

gtu greater than unsigned

Code: 25 FF

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

Definition:

if ($Breg_{unsigned} > Areg_{unsignded}$)
  Areg′ ← *true*
else
  Areg′ ← *false*

Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:
Secondary instruction.

in — input message

Code: F7

Description: Input a message. The corresponding output is performed by an *out*, *outword* or *outbyte* instruction, and must specify a message of the same length. **Areg** is the unsigned length in bytes, **Breg** is a pointer to the channel and **Creg** is a pointer to where the message is to be stored. The process executing *in* will be descheduled if the channel is external or is not ready, and is rescheduled when the communication is complete.

Definition:

Synchronize, and input $\text{Areg}_{\text{unsigned}}$ *bytes from channel* Breg *to address* Creg

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: Can cause *InternalChannel* or *ExternalChannel* trap to be signalled (if enabled) when the process is rescheduled after synchronization.

Comments:

Secondary instruction.
Instruction is a descheduling point.
Instruction is interruptible.

See also: *out*

insertqueue insert at front of scheduler queue

Code: 60 F2

Description: Insert a list of processes at the front of the scheduling list of priority indicated by **Areg**, where 0 indicates high priority and 1 indicates low priority. **Breg** and **Creg** are the front and back, respectively, of the list to be inserted.

Definition:

```
if (Breg ≠ NotProcess.p)
{
    ProcQueueFPtr′[Areg]←Breg

    if (ProcQueueFPtr[Areg] = NotProcess.p)
        ProcQueueBPtr′[Areg]←Creg
    else
        word′[Creg @ pw.Link]  ←  ProcQueueFPtr[Areg]
}

Areg′  ←  undefined
Breg′  ←  undefined
Creg′  ←  undefined
```

Error signals: none

Comments:
Secondary instruction.

See also: *swapqueue*

intdis (localised) interrupt disable

Code: 2C F4

Description: Disable interruption by high priority processes until either an *intenb* instruction is executed or the process deschedules. Timeslicing does not occur while interrupts are disabled. This instruction is only meaningful for low priority processes.

Definition:

Disable high priority interrupts

Error signals: none

Comments:

Secondary instruction.

See also: *intenb settimeslice*

intenb (localised) interrupt enable

Code: 2C F5

Description: Enable interruption by high priority processes. This instruction is only meaningful for low priority processes.

Definition:

Enable high priority interrupts

Error signals: none

Comments:

Secondary instruction.

See also: *intdis settimeslice*

iret interrupt return

Code: 61 FF

Description: Return from external interrupt. Signal *iret* to interrupt handler and return to the context of the interrupted process and resume execution. The interrupted high priority state is recovered from the workspace – if this does not contain a running process the processor switches to the interrupted low priority state held in the shadow registers.

Definition:

```
Status′    ←  word[Wptr]
if (Status has valid bit set)
{
    Wptr′  ←  word[Wptr + 1]
    Iptr′  ←  word[Wptr + 2]
    Areg′  ←  word[Wptr + 3]
    Breg′  ←  word[Wptr + 4]
    Creg′  ←  word[Wptr + 5]
}
else
    Return to interrupted low priority state
```

Error signals: none

Comments:
Secondary instruction.

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. *j* 0 causes a breakpoint.

Definition:

if (n = 0)
  Take a breakpoint *trap*
else
  Iptr′ ← *next instruction* + n

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: *j* 0 can cause a *breakpoint* trap to be signalled.

Comments:

Primary instruction.
Instruction is a descheduling point.
Instruction is a timeslicing point.

See also: *cj lend*

ladd long add

Code: 21 F6

Description: Add with carry in and check for overflow. The result of the operation is the sum of **Areg**, **Breg** and bit 0 of **Creg**.

Definition:

if (sum > MostPos)
{
  Areg′ ← sum – $2^{\text{BitsPerWord}}$
  IntegerOverflow
}
else if (sum < MostNeg)
{
  Areg′ ← sum + $2^{\text{BitsPerWord}}$
  IntegerOverflow
}
else
  Areg′ ← sum

Breg′ ← *undefined*
Creg′ ← *undefined*

where sum = Areg + Breg + Creg_0
– *the value of* sum *is calculated to unlimited precision*

Error signals: *IntegerOverflow* can be signalled.

Comments:
Secondary instruction.

See also: *lsum*

lb load byte

Code: F1

Description: Load the unsigned byte addressed by **Areg** into **Areg**.

Definition:

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

$Areg'_{8..BitsPerWord-1} \leftarrow 0$

Error signals: none

Comments:

Secondary instruction.

See also: *bsub devlb lbx ls ldul*

lbx load byte and sign extend

Code: 2B F9

Description: Load the byte addressed by **Areg** into **Areg** and sign extend to a word.

Definition:

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

$Areg'_{8..BitsPerWord-1} \leftarrow Areg'_{7}$

Error signals: none

Comments:

Secondary instruction.

See also: *bsub devlb lb xbword lsx ldul*

ldc n — load constant

Code: Function 4

Description: Load constant into **Areg**.

Definition:

Areg′ ← n

Breg′ ← Areg
Creg′ ← Breg

Error signals: none

Comments:

Primary instruction.

See also: *adc mint*

ldclock load clock

Code: 64 FD

Description: Load into **Areg** the current value of **ClockReg**, of the priority selected by **Areg**, where 0 indicates high priority and 1 indicates low priority.

Definition:

Areg′ ← ClockReg[Areg]

Error signals: none

Comments:

Secondary instruction.

See also: *stclock*

lddevid load device identity

Code: 21 27 FC

Description: See *ldprodid*. This instruction may be removed in future so *ldprodid* should be used instead.

Definition:

Areg′ ← *ProductId*

Breg′ ← Areg
Creg′ ← Breg

Error signals: none

Comments:

Secondary instruction.

ldiff long diff

Code: 24 FF

Description: Subtract unsigned numbers with borrow in. Subtract **Areg** from **Breg** minus borrow in from **Creg**, producing difference in **Areg** and borrow out in **Breg**, without checking for overflow.

Definition:

if (diff $\geq$ 0)
{
  $Areg'_{unsigned} \leftarrow diff$
  $Breg' \leftarrow 0$
}
else
{
  $Areg'_{unsigned} \leftarrow diff + 2^{BitsPerWord}$
  $Breg' \leftarrow 1$
}
$Creg' \leftarrow$ *undefined*

where $diff = Breg_{unsigned} - Areg_{unsigned} - Creg_0$
– *the value of* diff *is calculated to unlimited precision*

Error signals: none

Comments:
Secondary instruction.

See also: *lsub*

ldinf load infinity

Code: 27 F1

Description: Load the single length floating point number *+infinity* onto the stack.

Definition:

Areg′ ← #7F800000

Breg′ ← Areg
Creg′ ← Breg

Error signals: none

Comments:

Secondary instruction.

See also: *cflerr*

ldiv long divide

Code: 21 FA

Description: Divide the double length unsigned integer in **Breg** and **Creg** (most significant word in **Creg**) by an unsigned integer in **Areg**. The quotient is put into **Areg** and the remainder into **Breg**. Overflow occurs if either the quotient is not representable in a single word, or if a division by zero is attempted; the condition for overflow is equivalent to $\mathbf{Creg}_{\text{unsigned}} \geq \mathbf{Areg}_{\text{unsigned}}$.

Definition:

if $(\text{Creg}_{\text{unsigned}} \geq \text{Areg}_{\text{unsigned}})$
IntegerOverflow
else
{
$\text{Areg}'_{\text{unsigned}} \leftarrow \text{long} / \text{Areg}_{\text{unsigned}}$
$\text{Breg}'_{\text{unsigned}} \leftarrow \text{long}\ \mathbf{rem}\ \text{Areg}_{\text{unsigned}}$
}

$\text{Creg}' \leftarrow$ *undefined*

where $\text{long} = (\text{Creg}_{\text{unsigned}} \times 2^{\text{BitsPerWord}}) + \text{Breg}_{\text{unsigned}}$
– *the value of* long *is calculated to unlimited precision*

Error signals:
IntegerOverflow can occur.

Comments:
Secondary instruction.

See also: *lmul*

ldl n — load local

Code: Function 7

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

Definition:

Areg′ ← word[Wptr @ n]

Breg′ ← Areg
Creg′ ← Breg

Error signals: none

Comments:

Primary instruction.

See also: *ldnl stl*

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

Definition:

Error signals: none

Comments:

Primary instruction.

See also: *ldl ldnlp stlp*

ldmemstartval load value of MemStart address

Code: 27 FE

Description: Load into **Areg** the address of the first free memory location (as defined in the **MemStart** configuration register).

Definition:

Areg′ ← MemStart

Breg′ ← Areg
Creg′ ← Breg

Error signals: none

Comments:
Secondary instruction.

ldnl n load non-local

Code: Function 3

Description: Load into **Areg** the non-local variable at the specified word offset from **Areg**.

Definition:

Areg′ ← word[Areg @ n]

Error signals: none

Comments:

Primary instruction.
Areg should be word aligned.

See also: *ldl ldnlp stnl*

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:

Areg′ ← Areg @ n

Error signals: none

Comments:

Primary instruction.

See also: *ldlp ldnl wsub*

ldpi load pointer to instruction

Code: 21 FB

Description: Load into **Areg** an address 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

Error signals: none

Comments:

Secondary instruction.

ldpri load current priority

Code: 21 FE

Description: Load the current process priority into **Areg**, where 0 indicates high priority and 1 indicates low priority.

Definition:

Areg′ ← Priority

Breg′ ← Areg
Creg′ ← Breg

Error signals: none

Comments:

Secondary instruction.

ldprodid load product identity

Code: 68 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′ ← ProductId

Breg′ ← Areg
Creg′ ← Breg

Error signals: none

Comments:

Secondary instruction.
Different ST20 products may use the same processor type, but return different product identity codes. However a product indentity code uniquely defines the processor type used in that product.

ldshadow load shadow registers

Code: 60 FC

Description: Selectively load (depending on **Areg**) the shadow registers of the priority determined by **Breg** from the block of store addressed by **Creg**. This instruction should only be used with **Breg** not equal to the current priority.

Definition:

if (Breg $\neq$ Priority)
{
  if ($Areg_0$ = 1)
  {
    GlobalInterruptEnables′ $\leftarrow$ $word[Creg]_{16..23}$
    TrapEnables′[Breg] $\leftarrow$ $word[Creg]_{0..13}$
  }
  if ($Areg_1$ = 1)
  {
    Status′[Breg] $\leftarrow$ word[Creg @ 1]
    Wptr′[Breg] $\leftarrow$ word[Creg @ 2]
    Iptr′[Breg $\leftarrow$ word[Creg @ 3]
  }
  if ($Areg_2$ = 1)
  {
    Areg′[Breg] $\leftarrow$ word[Creg @ 4]
    Breg′[Breg] $\leftarrow$ word[Creg @ 5]
    Creg′[Breg] $\leftarrow$ word[Creg @ 6]
  }
  if ($Areg_3$ = 1)
  {
    Load block move registers for priority Breg *from*
      word′[Creg @ 7] .. word′[Creg @ 11]
  }
}
else
  Undefined effect

Areg′ $\leftarrow$ *undefined*
Breg′ $\leftarrow$ *undefined*
Creg′ $\leftarrow$ *undefined*

Error signals: *none*

Comments:

Secondary instruction.

The effect of this instruction is undefined if the items other than the block move registers are loaded into the current priority.

This instruction is abortable.

See also: *stshadow restart*

ldtimer load timer

Code: 22 F2

Description: Load the value of the current priority timer into **Areg**.

Definition:

Areg′ ← ClockReg[Priority]

Breg′ ← Areg
Creg′ ← Breg

Error signals: none

Comments:

Secondary instruction.

See also: *sttimer tin*

ldtraph load trap handler

Code: 26 FE

Description: Install the trap handler structure to be found at the address in **Breg** into the trap handler location for the trap group given by **Areg** and priority given by **Creg**, where 0 indicates high priority and 1 indicates low priority.

Definition:

word′[traphandler @ 0] ← word[Breg @ 0]
word′[traphandler @ 1] ← word[Breg @ 1]
word′[traphandler @ 2] ← word[Breg @ 2]
word′[traphandler @ 3] ← word[Breg @ 3]

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

where traphandler = *the address of the trap handler for the trap group Areg and priority Creg.*

Error signals: Can cause a load trap trap to be signalled.

Comments:

Secondary instruction.

See also: *ldtrapped sttraph sttrapped*

ldtrapped load trapped process status

Code: 2C F6

Description: Install the trapped process structure, to be found at the address in **Breg,** into the trapped process location for the trap group given by **Areg** and priority given by **Creg**, where 0 indicates high priority and 1 indicates low priority.

Definition:

word′[trapped @ 0]	←	word[Breg @ 0]
word′[trapped @ 1]	←	word[Breg @ 1]
word′[trapped @ 2]	←	word[Breg @ 2]
word′[trapped @ 3]	←	word[Breg @ 3]

Areg′	←	*undefined*
Breg′	←	*undefined*
Creg′	←	*undefined*

where trapped = *the address of the stored trapped process for the trap group Areg and priority Creg.*

Error signals: Can cause a load trap trap to be signalled.

Comments:
Secondary instruction.

See also: *ldtraph sttraph sttrapped*

lend loop end

Code: 22 F1

Description: Adjust loop count and index, and do a conditional jump. Initially **Areg** contains the byte offset from the first byte following this instruction to the loop start and **Breg** contains a pointer to a loop end data structure, the first word of which is the loop index and the second is the loop count. The count is decremented and, if the result is greater than zero, the index is incremented and a jump to the start of the loop is taken. The offset to the start of the loop is given as a positive number that is subtracted from the instruction pointer.

Definition:

if (word[Breg @ le.Count] > 1)
{
  word′[Breg @ le.Count] ← word[Breg @ le.Count] – 1
  word′[Breg @ le.Index] ← word[Breg @ le.Index] + 1
  Iptr′ ← *next instruction* – Areg
}

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.
Instruction is a descheduling point.
Instruction is a timeslice point.

See also: *cj j*

lmul long multiply

Code: 23 F1

Description: Form the double length product of **Areg** and **Breg**, with **Creg** as carry in, treating the initial values as unsigned.

Definition:

$Areg'_{unsigned} \leftarrow$ prod **rem** $2^{BitsPerWord}$
$Breg'_{unsigned} \leftarrow$ prod / $2^{BitsPerWord}$

$Creg' \leftarrow$ *undefined*

where prod $= (Breg_{unsigned} \times Areg_{unsigned}) + Creg_{unsigned}$
– *the value of* prod *is calculated to unlimited precision*

Error signals: none

Comments:
Secondary instruction.

See also: *ldiv*

ls load sixteen

Code: 2C FA

Description: Load the unsigned 16-bit object addressed by **Areg** into **Areg**.

Definition:

$Areg'_{0..15} \leftarrow$ sixteen[Areg]
$Areg'_{16..BitsPerWord-1} \leftarrow 0$

Error signals: none

Comments:

Secondary instruction.

See also: *devls lsx ss ssub lt ldul*

lshl long shift left

Code: 23 F6

Description: Logical shift left the double word value in **Creg** and **Breg** (most significant word in **Creg**) by the number of places specified in **Areg**. Only defined if the shift length is less than twice the wordlength.

Definition:

if $(0 \le \text{Areg} < 2 \times \text{BitsPerWord})$
{
  $\text{Areg}' \leftarrow (\text{long} << \text{Areg}_{\text{unsigned}})\ \textbf{rem}\ 2^{\text{BitsPerWord}}$
  $\text{Breg}' \leftarrow ((\text{long} << \text{Areg}_{\text{unsigned}})\ /\ 2^{\text{BitsPerWord}})\ \textbf{rem}\ 2^{\text{BitsPerWord}}$
}
else {
  $\text{Areg}' \leftarrow$ *undefined*
  $\text{Breg}' \leftarrow$ *undefined*
}

$\text{Creg}' \leftarrow$ *undefined*

where $\text{long} = (\text{Creg}_{\text{unsigned}} \times 2^{\text{BitsPerWord}}) + \text{Breg}_{\text{unsigned}}$
– *the value of* long *is calculated to double word precision*

Error signals: none

Comments:

Secondary instruction.
The behavior for shift lengths outside the stated range is implementation dependent.

See also: *lshr norm*

lshr long shift right

Code: 23 F5

Description: Logical shift right the double word value in **Creg** and **Breg** (most significant word in **Creg**) by the number of places specified in **Areg**. This instruction is only defined if the shift length is less than twice the word length.

Definition:

if $(0 \le \text{Areg} < 2 \times \text{BitsPerWord})$
{
Areg′ ← $(\text{long} >> \text{Areg}_{\text{unsigned}})$ **rem** $2^{\text{BitsPerWord}}$
Breg′ ← $((\text{long} >> \text{Areg}_{\text{unsigned}}) / 2^{\text{BitsPerWord}})$ **rem** $2^{\text{BitsPerWord}}$
} **else** {
Areg′ ← *undefined*
Breg′ ← *undefined*
}

Creg′ ← *undefined*

where $\text{long} = (\text{Creg}_{\text{unsigned}} \times 2^{\text{BitsPerWord}}) + \text{Breg}_{\text{unsigned}}$
– *the value of* long *is calculated to double word precision*

Error signals: none

Comments:

Secondary instruction.
The behavior for shift lengths outside the stated range is implementation dependent.

See also: *lshl*

lsub long subtract

Code: 23 F8

Description: Subtract with borrow in and check for overflow. The result of the operation, put into **Areg**, is **Breg** minus **Areg**, minus bit 0 of **Creg**.

Definition:

if (diff > MostPos)
{
Areg′ ← diff – $2^{BitsPerWord}$
IntegerOverflow
}
else if (diff < MostNeg)
{
Areg′ ← diff + $2^{BitsPerWord}$
IntegerOverflow
}
else
Areg′ ← diff

Breg′ ← *undefined*
Creg′ ← *undefined*

where diff = (Breg – Areg) – $Creg_0$
– *the value of* diff *is calculated to unlimited precision*

Error signals: *IntegerOverflow* can be signalled.

Comments:
Secondary instruction.

See also: *ldiff*

lsum long sum

Code: 23 F7

Description: Add unsigned numbers with carry in and carry out. Add **Breg** to **Areg** (treated as unsigned numbers) plus carry in from **Creg**, producing the sum in **Areg** and carry out in **Breg**, without checking for overflow.

Definition:

if (sum $>$ MostPosUnsigned)
{
  $\text{Areg}'_{\text{unsigned}} \leftarrow \text{sum} - 2^{\text{BitsPerWord}}$
  $\text{Breg}' \leftarrow 1$
}
else
{
  $\text{Areg}'_{\text{unsigned}} \leftarrow \text{sum}$
  $\text{Breg}' \leftarrow 0$
}

$\text{Creg}' \leftarrow$ *undefined*

where sum $= \text{Areg}_{\text{unsigned}} + \text{Breg}_{\text{unsigned}} + \text{Creg}_0$
 – *the value of* sum *is calculated to unlimited precision*

Error signals: none

Comments:
Secondary instruction.

See also: *ladd*

lsx load sixteen and sign extend

Code: 2F F9

Description: Load the 16-bit object addressed by **Areg** into **Areg** and sign extend to a word.

Definition:

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

$Areg'_{16..BitsPerWord-1} \leftarrow Areg'_{15}$

Error signals: none

Comments:

Secondary instruction.

See also: *devls ls ss xsword ltx ldnl*

mint minimum integer

Code: 24 F2

Description: Load the most negative integer into **Areg**.

Definition:

Areg′ ← MostNeg

Breg′ ← Areg
Creg′ ← Breg

Error signals: none

Comments:

Secondary instruction.

move move message

Code: 24 FA

Description: Copy **Areg** bytes to address **Breg** from address **Creg**. The copy is performed using the minimum number of word reads and writes.

Definition:

if (*source and destination overlap*)
Undefined effect
else for i = 0..($\text{Areg}_{\text{unsigned}}$ – 1)
byte′[Breg + i] ← byte[Creg + i]

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:
Secondary instruction.

See also: *devmove in move2dall out*

move2dall 2D block copy

Code: 25 FC

Description: Copy a 2D block of memory to another, non-overlapping, area using parameters set up by *move2dinit*. The copy is performed using the minimum number of word reads and writes. **Areg** is the number of bytes in each row, **Breg** is the address of the destination, and **Creg** is the address of the source.

Definition:

if (*source and destination overlap*)
    Undefined effect

else for y = 0 .. (count – 1)
{
    for x = 0 .. ($\text{Areg}_{\text{unsigned}}$ – 1)
        byte′[Breg + (y × dstStride) + x] ← byte[Creg + (y × srcStride) + x]
}

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

where
    count = Move2dBlockLength
    dstStride = Move2dDestStride
    srcStride = Move2dSourceStride

Error signals: none

Comments:
Secondary instruction.
Instruction is interruptible.

See also: *move2dinit move2d nonzero move2dzero1*

move2dinit initialize data for 2D block move

Code: 25 FB

Description: Set up the first three parameters for a 2D block move: **Areg** is the number of rows to copy, **Breg** is the width of the destination array, and **Creg** is the width of the source array. This instruction must be executed before each 2D block move.

Definition:

Move2dBlockLength′	←	Areg
Move2dDestStride′	←	Breg
Move2dSourceStride′	←	Creg

Areg′	←	*undefined*
Breg′	←	*undefined*
Creg′	←	*undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *move2dall move2dnonzero move2dzero stmove2dinit*

move2dnonzero 2D block copy non-zero bytes

Code: 25 FD

Description: Copy non-zero valued bytes from a 2D block of memory to another, non-overlapping, area using parameters set up by *move2dinit*. The copy is performed using the minimum number of word reads and writes. **Areg** is the number of bytes in each row, **Breg** is the address of the destination, and **Creg** is the address of the source.

Definition:

if (*source and destination overlap*)
Undefined effect

else for y = 0 .. (count – 1)
for x = 0 .. ($Areg_{unsigned}$ – 1)
if (byte[Creg + (y × srcStride) + x] ≠ 0)
byte′[Breg + (y × dstStride) + x] ← byte[Creg + (y × srcStride) + x]

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

where

count = Move2dBlockLength
dstStride = Move2dDestStride
srcStride = Move2dSourceStride

Error signals: none

Comments:
Secondary instruction.
Instruction is interruptible.

See also: *move2dinit move2dzero move2dall*

move2dzero 2D block copy zero bytes

Code: 25 FE

Description: Copy zero valued bytes from a 2D block of memory to another, non-overlapping, area using parameters set up by *move2dinit*. The copy is performed using the minimum number of word reads. **Areg** is the number of bytes in each row, **Breg** is the address of the destination, and **Creg** is the address of the source.

Definition:

if (*source and destination overlap*)
Undefined effect

else for y = 0 .. (count – 1)
for x = 0 .. ($\text{Areg}_{\text{unsigned}}$ – 1)
if (byte[Creg + (y × srcStride) + x] = 0)
byte′[Breg + (y × dstStride) + x] ← byte[Creg + (y × srcStride) + x]

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

where

count	=	Move2dBlockLength
dstStride	=	Move2dDestStride
srcStride	=	Move2dSourceStride

Error signals: none

Comments:
Secondary instruction.
Instruction is interruptible.

See also: *move2dinit*

mul multiply

Code: 25 F3

Description: Multiply **Areg** by **Breg**, with checking for overflow.

Definition:

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

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

Error signals:

IntegerOverflow can be signalled by $\times_{\text{checked}}$

Comments:

Secondary instruction.

See also: *prod*

nop no operation

Code: 63 F0

Description: Perform no operation.

Definition:

no effect

Error signals: none

Comments:

Secondary instruction.

norm normalize

Code: 21 F9

Description: Normalize the unsigned double length number stored in **Breg** and **Areg** (most significant word in **Breg**). The value is shifted left until the most significant bit is a one. The number of places shifted is returned in **Creg**. This instruction is used as the first instruction in the single length floating point rounding code sequence, *norm; postnormsn; roundsn*.

Definition:

if $((\text{Breg}_{\text{unsigned}} = 0)$ **and** $(\text{Areg}_{\text{unsigned}} = 0))$

$\text{Creg}' \leftarrow 2 \times \text{BitsPerWord}$

else

{

$\text{Creg}' \leftarrow$ *number of most significant zero bits in* long

$\text{Areg}'_{\text{unsigned}} \leftarrow (\text{long} << \text{Creg}') \text{ rem } 2^{\text{BitsPerWord}}$

$\text{Breg}'_{\text{unsigned}} \leftarrow ((\text{long} << \text{Creg}') / 2^{\text{BitsPerWord}}) \text{ rem } 2^{\text{BitsPerWord}}$

}

where $\text{long} = (\text{Breg}_{\text{unsigned}} \times 2^{\text{BitsPerWord}}) + \text{Areg}_{\text{unsigned}}$

– *the value of* long *is calculated to double word precision*

Error signals: none

Comments:

Secondary instruction.

See also: *lshl lshr postnormsn roundsn shl shr*

not not

Code: 23 F2

Description: Complement bits in **Areg**.

Definition:

Areg′ ← ~Areg

Error signals: none

Comments:

Secondary instruction.

or or

Code: 24 FB

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

Definition:

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

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

Error signals: none

Comments:

Secondary instruction.

out output message

Code: FB

Description: Output a message (where the corresponding input is performed by an *in* instruction, and must specify a message of the same length). **Areg** is the unsigned length, **Breg** is a pointer to the channel, and **Creg** is a pointer to the message. The process executing *out* will be descheduled if the channel is external or is not ready; it is rescheduled when the communication is complete. This instruction is also used to synchronize with an alternative.

Definition:

Synchronize, and output $\text{Areg}_{\text{unsigned}}$ *bytes to channel* Breg *from address* Creg

$\text{Areg}' \leftarrow$ *undefined*
$\text{Breg}' \leftarrow$ *undefined*
$\text{Creg}' \leftarrow$ *undefined*

Error signals: Can cause *InternalChannel* or *ExternalChannel* trap to be signalled (if enabled) when the process is rescheduled on synchronization.

Comments:

Secondary instruction.

See also: *altwt enbc in outbyte outword*

outbyte output byte

Code: FE

Description: Output the least significant byte of **Areg** to the channel pointed to by **Breg** (where the corresponding input is performed by an *in* instruction, and must specify a single byte message). The process executing *outbyte* will be descheduled if the channel is external or is not ready; it is rescheduled when the communication is complete. This instruction is also used to synchronize with an alternative.

Definition:

Synchronize, and output least significant byte of Areg *to channel* Breg

word′[Wptr @ pw.Temp]	←	*undefined*
Areg′	←	*undefined*
Breg′	←	*undefined*
Creg′	←	*undefined*

Error signals: Can cause *InternalChannel* or *ExternalChannel* trap to be signalled (if enabled) when process is rescheduled on synchronization.

Comments:

Secondary instruction.
Instruction is a descheduling point.
Instruction is interruptible.
Uses the **pw.Temp** slot in the process workspace.

See also: *altwt enbc in out outword*

outword output word

Code: FF

Description: Output the word in **Areg** to the channel pointed to by **Breg** (the corresponding input is performed by an *in* instruction, and must specify a four byte message). The process executing *outword* will be descheduled if the channel is external or is not ready; it is rescheduled when the communication is complete. This instruction is also used to synchronize with an alternative.

Definition:

Synchronize, and output Areg *to channel* Breg

word′[Wptr @ pw.Temp]	←	*undefined*
Areg′	←	*undefined*
Breg′	←	*undefined*
Creg′	←	*undefined*

Error signals: Can cause *InternalChannel* or *ExternalChannel* trap to be signalled (if enabled) when the process is rescheduled on synchronization.

Comments:

Secondary instruction.
Instruction is a descheduling point.
Instruction is interruptible.
Uses the **pw.Temp** slot in the process workspace.

See also: *altwt enbc in out outbyte*

pop pop processor stack

Code: 27 F9

Description: Pop top element of integer stack.

Definition:

Areg′ ← Breg
Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *dup rev*

postnormsn post-normalize correction of single length fp number

Code: 26 FC

Description: Perform the post normalised correction on a floating point number, where initially the normalised fraction is in **Areg**, **Breg** and **Creg** as left by the instruction *norm*, and the exponent is in location **pw.Temp** in the workspace. This instruction is only intended to be used in the single length rounding code sequence immediately after *norm* and before *roundsn*.

Definition:

Areg′	←	*post-normalised guardword*
Breg′	←	*post-normalised fractionword*
Creg′	←	*post-normalised exponent*

Error signals: none

Comments:

Secondary instruction.
This instruction uses location **pw.Temp** in the workspace.

See also: *roundsn norm*

prod product

Code: F8

Description: Multiply **Areg** by **Breg** without checking for overflow.

Definition:

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

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

Error signals: none

Comments:
Secondary instruction.

See also: *mul*

reboot reboot

Code: 68 FD

Description: Perform a cold boot. Reset all machine states to the initial state and execute the boot microcode. This examines the state of the **BootFromRom** pin and reboots accordingly.

Error signals:

Reboot the machine and either listen for a boot protocol on link or jump to ROM entry point

Error signals: none

Comments:

Secondary instruction.

rem remainder

Code: 21 FF

Description: Calculate the remainder when **Breg** is divided by **Areg**. The sign of the remainder is the same as the sign of **Breg**.

The remainder, r = x **rem** y, is defined by r = x – (y × (x / y)).

Definition:

if (Areg = 0)
{
    Areg′ ← *undefined*
    IntegerOverflow
}
else
    Areg′ ← Areg **rem** Breg

Breg′ ← Creg
Creg′ ← *undefined*

Error signals:
IntegerOverflow signalled when a remainder by zero is attempted.

Comments:
Secondary instruction.

See also: *div*

resetch reset channel

Code: 21 F2

Description: Reset the channel pointed to by **Areg**. Returns the channel to the empty state. If the channel address points to a hard channel, then the link hardware is reset. **Areg** returns the process descriptor of the process waiting on the channel.

Definition:

if (Areg *points to external channel*)
reset link hardware

word′[Areg]	←	NotProcess.p
Areg′	←	word[Areg]

Error signals: none

Comments:

Secondary instruction.
This instruction is abortable.

restart restart

Code: 62 FE

Description: Restart execution of a saved process in place of the current process. **Areg** is a pointer to a processor state data structure which will have been obtained using *stshadow*.

Definition:

GlobalInterruptEnables′	←	word[Areg @ 0]$_{16..23}$
TrapEnables′	←	word[Areg @ 0]$_{0..13}$
Status′	←	word[Areg @ 1]
Wptr′	←	word[Areg @ 2]
Iptr′	←	word[Areg @ 3]
Areg′	←	word[Areg @ 4]
Breg′	←	word[Areg @ 5]
Creg′	←	word[Areg @ 6]

Load block move registers for current priority from
word′[Areg @ 7] .. word′[Areg @ 11]

Error signals: none

Comments:
Secondary instruction.

See also: *ldshadow stshadow*

ret return

Code: 22 F0

Description: Return from a subroutine and de-allocate workspace.

Definition:

Iptr′ ← word[Wptr @ 0]
Wptr′ ← Wptr @ 4

Error signals: none

Comments:

Secondary instruction.

See also: *ajw call*

rev reverse

Code: F0

Description: Swap the top two elements of the evaluation stack.

Definition:

$Areg' \leftarrow Breg$
$Breg' \leftarrow Areg$

Error signals: none

Comments:

Secondary instruction.

See also: *dup pop*

roundsn round single length floating point number

Code: 26 FD

Description: Round an unpacked result of a floating point operation into **Areg**. Rounding is performed in round-to-nearest mode, as defined by IEEE-754. Initially the post-normalised guardword, fractionword and exponent are in **Areg**, **Breg**, and **Creg**, as left by the instruction *postnormsn*. This instruction is only intended to be used in the single length rounding code sequence immediately after *postnormsn*.

Definition:

Areg′ ← *rounded and packed fp number*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: *none*

Comments:

Secondary instruction.

See also: *postnormsn unpacksn*

runp run process

Code: 23 F9

Description: Schedule a (descheduled) process. The process descriptor of the process is in **Areg**; this identifies the process workspace and priority. The instruction pointer is loaded from the process' workspace data structure.

Definition:

Put process Areg *onto the back of the appropriate scheduling list*

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *endp startp stopp*

satadd saturating add

Code: 26 F8

Description: Perform addition using ‘saturating arithmetic’ i.e. signed arithmetic where overflowing results do not wrap round but return MostPos or MostNeg according to the sign of the result. This instruction is used for clipping algorithms in signal processing.

Definition:

if (sum > MostPos)
  Areg′ ← MostPos
else if (sum < MostNeg)
  Areg′ ← MostNeg
else
  Areg′ ← sum

Breg′ ← Creg
Creg′ ← *undefined*

where sum = Breg + Areg
  – sum *is calculated to unlimited precision*

Error signals: none

Comments:
Secondary instruction.

See also: *satsub satmul*

satmul saturating multiply

Code: 26 FA

Description: Perform multiplication using ‘saturating arithmetic’ i.e. signed arithmetic where overflowing results do not wrap round but return MostPos or MostNeg according to the sign of the result. This instruction is used for clipping algorithms in signal processing.

Definition:

if (prod $>$ MostPos)
  Areg′ $\leftarrow$ MostPos
else if (prod $<$ MostNeg)
  Areg′ $\leftarrow$ MostNeg
else
  Areg′ $\leftarrow$ prod

Breg′ $\leftarrow$ Creg
Creg′ $\leftarrow$ *undefined*

where prod $=$ Breg $\times$ Areg
  – prod *is calculated to unlimited precision*

Error signals: none

Comments:
Secondary instruction.

See also: *satadd satsub*

satsub saturating subtract

Code: 26 F9

Description: Perform subtraction using 'saturating arithmetic' i.e. signed arithmetic where overflowing results do not wrap round but return MostPos or MostNeg according to the sign of the result. This instruction is used for clipping algorithms in signal processing.

Definition:

if (diff > MostPos)
    Areg′ ← MostPos
else if (diff < MostNeg)
    Areg′ ← MostNeg
else
    Areg′ ← diff

Breg′ ← Creg
Creg′ ← *undefined*

where diff = Breg – Areg
    – diff *is calculated to unlimited precision*

Error signals: none

Comments:
Secondary instruction.

See also: *satadd satmul*

saveh save high priority queue registers

Code: 23 FE

Description: Save high priority queue pointers. Stores the contents of the high priority scheduling registers in the block given by the address in **Areg**.

This instruction has been superceded by *insertqueue* and *swapqueue* which should be used instead.

Definition:

word′[Areg @ 0]	←	ProcQueueFPtr[HighPriority]
word′[Areg @ 1]	←	ProcQueueBPtr[HighPriority]

Areg′	←	Breg
Breg′	←	Creg
Creg′	←	*undefined*

Error signals: none

Comments:
Secondary instruction.

See also: *savel insertqueue swapqueue*

savel save low priority queue registers

Code: 23 FD

Description: Save low priority queue pointers. Stores the contents of the low priority scheduling registers in the block given by the address in **Areg**.

This instruction has been superceded by *insertqueue* and *swapqueue* which should be used instead.

Definition:

word′[Areg @ 0] ← ProcQueueFPtr[LowPriority]
word′[Areg @ 1] ← ProcQueueBPtr[LowPriority]

Areg′ ← Breg
Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *saveh insertqueue swapqueue*

sb store byte

Code: 23 FB

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

Definition:

byte′[Areg]	←	$\text{Breg}_{0..7}$
Areg′	←	Creg
Breg′	←	*undefined*
Creg′	←	*undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *bsub devsb lb lbx ss stnl*

seterr set error flags

Code: 21 F0

Description: Unconditionally set the error flag for the current priority.

Definition:

ErrorFlag′[Priority] $\leftarrow$ *set*

Error signals:

The error flag is set by this instruction.

Comments:

Secondary instruction.

See also: *stoperr testerr*

sethalterr set halt-on-error flag

Code: 25 F8

Description: Set the HaltOnError flag to put the processor into halt-on-error mode.

Definition:

HaltOnErrorFlag′ $\leftarrow$ *set*

Error signals: none

Comments:

Secondary instruction.

See also: *clrhalterr tsthalterr*

settimeslice set timeslicing status

Code: 2B F0

Description: Enable or disable timeslicing of the current process, depending on the value of **Areg**, and set **Areg** to indicate whether timeslicing was enabled or disabled prior to execution of the instruction. If **Areg** is initially *false* timeslicing is disabled until either the process deschedules or timeslicing is enabled. If **Areg** is initially *true* timeslicing is enabled. This instruction is only meaningful when run at low priority.

Definition:

if (Areg = *false*)
  Disable timeslicing
else if (Areg = *true*)
  Enable timeslicing
else
  Undefined effect

if (*timeslicing was previously enabled*)
  Areg′ ← *true*
else
  Areg′ ← *false*

Error signals: none

Comments:

Secondary instruction.

See also: *intdis intenb*

shl shift left

Code: 24 F1

Description: Logical shift left **Breg** by **Areg** places, filling with zero bits. If the initial **Areg** is not between 0 and 31 inclusive then the result is zero. The result is only defined for shift lengths less than the word length.

Definition:

if (0 ≤ Areg < BitsPerWord)
  Areg′ ← Breg << Areg
else
  Areg′ ← *undefined*

Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:
Secondary instruction.
The behavior for shift lengths outside the stated range is implementation dependent.

See also: *lshl lshr norm shr*

shr shift right

Code: 24 F0

Description: Logical shift right **Breg** by **Areg** places, filling with zero bits. If the initial **Areg** is not between 0 and 31 inclusive then the result is zero. The result is only defined for shift lengths less than the word length.

Definition:

if (0 ≤ Areg < BitsPerWord)
  Areg′ ← Breg >> Areg
else
  Areg′ ← *undefined*

Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.
The behavior for shift lengths the outside stated range is implementation dependent.

See also: *lshl lshr norm shl*

signal signal

Code: 60 F4

Description: Signal (or V) on the semaphore pointed to by **Areg**. If no process is waiting then the count is incremented, otherwise the first process on the semaphore list is rescheduled.

Definition:

if (word[Areg @ s.Front] = NotProcess.p)
  word′[Areg @ s.Count] ← word[Areg @ s.Count] + 1
else
  Remove the process from the front of the semaphore list and put it on the scheduling list

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg ← *undefined*

Error signals: Can cause the *Signal* trap to be signalled if a process is rescheduled.

Comments:

Secondary instruction.
Count increment is unchecked.

See also: *wait*

slmul signed long multiply

Code: 26 F4

Description: Perform signed long multiplication. This instruction forms the double length product of **Areg** and **Breg**, with **Creg** as carry in, treating the initial values **Areg** and **Breg** as signed.

Definition:

$\text{Areg}'_{\text{unsigned}} \leftarrow \text{prod } \mathbf{rem}\ 2^{\text{BitsPerWord}}$

$\text{Breg}' \leftarrow \text{prod} / 2^{\text{BitsPerWord}}$

$\text{Creg}' \leftarrow$ *undefined*

where $\text{prod} = (\text{Breg} \times \text{Areg}) + \text{Creg}_{\text{unsigned}}$
– *the value of* prod *is calculated to unlimited precision*

Error signals: *none*

Comments:
Secondary instruction.

See also: *lmul sulmul*

ss store sixteen

Code: 2C F8

Description: Store bits 0..15 of **Breg** into the sixteen bits of memory addressed by **Areg**.

Definition:

sixteen′[Areg]	←	$\text{Breg}_{0..15}$
Areg′	←	Creg
Breg′	←	*undefined*
Creg′	←	*undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *devss ldlp ldnlp ls lsx sb stl stnl ssub*

ssub sixteen subscript

Code: 2C F1

Description: Generate the address of the element which is indexed by **Breg**, in an array of 16-bit objects pointed to by **Areg**.

Definition:

Areg′ ← Areg + (2 × Breg)
Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *bcnt bsub wcnt wsub wsubdb*

startp start process

Code: FD

Description: Create and schedule a process at the current priority. Initially **Areg** is a pointer to the workspace of the new process and **Breg** is the offset from the next instruction to the instruction pointer of the new process.

Definition:

word′[Areg @ pw.Iptr] ← *next instruction* + Breg

Put the process Areg *onto the back of the scheduling list for the current priority*

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *endp runp*

stclock store clock register

Code: 64 FC

Description: Store the contents of **Breg** into the clock register of priority **Areg**.

Definition:

ClockReg[Areg] ← Breg

Areg′ ← Creg
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *ldclock*

sthb store high priority back pointer

Code: 25 F0

Description: Store the contents of **Areg** into the back pointer of the high priority queue.

This instruction has been superceded by *insertqueue* and *swapqueue* which should be used instead.

Definition:

ProcQueueBPtr[HighPriority] $\leftarrow$ Areg

Error signals: none

Comments:

Secondary instruction.

See also: *insertqueue sthf stlb stlf swapqueue*

sthf store high priority front pointer

Code: 21 F8

Description: Store the contents of **Areg** into the front pointer of the high priority queue.

This instruction has been superceded by *insertqueue* and *swapqueue* which should be used instead.

Definition:

ProcQueueFPtr[HighPriority] $\leftarrow$ Areg

Error signals: none

Comments:

Secondary instruction.

See also: *insertqueue sthb stlb stlf swapqueue*

stl n — store local

Code: Function D

Description: Store the contents of **Areg** into the local variable at the specified word offset in workspace.

Definition:

word′[Wptr @ n] ← Areg

Areg′ ← Breg
Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:

Primary instruction.

See also: *devsw ldl ldlp sb ss stnl*

stlb store low priority back pointer

Code: 21 F7

Description: Store the contents of **Areg** into the back pointer of the low priority queue.

This instruction has been superceded by *insertqueue* and *swapqueue* which should be used instead.

Definition:

ProcQueueBPtr[LowPriority] $\leftarrow$ Areg

Error signals: none

Comments:

Secondary instruction.

See also: *insertqueue sthb sthf stlf swapqueue*

stlf store low priority front pointer

Code: 21 FC

Description: Store the contents of **Areg** into the front pointer of the low priority queue.

This instruction has been superceded by *insertqueue* and *swapqueue* which should be used instead.

Definition:
ProcQueueFPtr[LowPriority] ← Areg

Error signals: none

Comments:
Secondary instruction.

See also: *insertqueue sthb sthf stlb swapqueue*

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:

word′[Areg @ n] ← Breg

Areg′ ← Creg
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:

Primary instruction.

See also: *devsw ldlp ldnlp sb ss stl*

stoperr stop on error

Code: 25 F5

Description: Deschedule the current process if the ErrorFlag is set.

Definition:

if (ErrorFlag[Priority] = *set*)
  word′[Wptr @ pw.Iptr] ← *next instruction*
  Stop process

Error signals: none

Comments:

Secondary instruction.

See also: *seterr testerr*

stopp stop process

Code: 21 F5

Description: Terminate the current process, saving the current **Iptr** for later use.

Definition:

word′[Wptr @ pw.Iptr] ← *next instruction*

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *endp runp startp*

stshadow store shadow registers

Code: 60 FD

Description: Selectively store shadow registers of priority **Breg** into the block of store addressed by **Creg**. This instruction is normally used in high priority with **Breg** set to 1 for low priority. Storing high priority registers from low priority will give undefined values.

Definition:

if ($Areg_0 = 1$)
{
  word′[Creg]$_{16..23}$ ← GlobalInterruptEnables
  word′[Creg]$_{0..13}$ ← TrapEnables[Breg]
}
if ($Areg_1 = 1$)
{
  word′[Creg @ 1] ← Status[Breg]
  word′[Creg @ 2] ← Wptr[Breg]
  word′[Creg @ 3] ← Iptr[Breg]
}
if ($Areg_2 = 1$)
{
  word′[Creg @ 4] ← Areg[Breg]
  word′[Creg @ 5] ← Breg[Breg]
  word′[Creg @ 6] ← Creg[Breg]
}
if ($Areg_3 = 1$)
  Store block move registers for priority Breg *in*
    word′[Creg @ 7] .. word′[Creg @ 11]

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:
Secondary instruction.

See also: *ldshadow restart*

sttimer store timer

Code: 25 F4

Description: Initialize the timers. Set the low and high priority clock registers to the value in **Areg** and start them ticking and scheduling ready processes.

Definition:

Clockreg′[0] ← Areg
Clockreg′[1] ← Areg
Start timers

Areg′ ← Breg
Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *clockenb clockdis ldclock*

sttraph store trap handler

Code: 26 FF

Description: Store the trap handler structure from the trap handler location for the trap group given by **Areg** and priority given by **Creg**, where 0 indicates high priority and 1 indicates low priority, to the block of memory pointed to by **Breg**.

Definition:

word′[Breg @ 0] ← word[traphandler @ 0]
word′[Breg @ 1] ← word[traphandler @ 1]
word′[Breg @ 2] ← word[traphandler @ 2]
word′[Breg @ 3] ← word[traphandler @ 3]

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

where traphandler *is the address of the traphandler for the trap group Areg and priority Creg.*

Error signals:

Will cause a *StoreTrap* trap if enabled.

Comments:

Secondary instruction.

See also: *ldtraph ldtrapped sttrapped*

sttrapped store trapped process

Code: 2C FB

Description: Store the trapped process structure from the trapped process location for the trap group given by **Areg** and priority given by **Creg**, where 0 indicates high priority and 1 indicates low priority, to the block of memory pointed to by **Breg**.

Definition:

word′[Breg @ 0] ← word[trapped @ 0]
word′[Breg @ 1] ← word[trapped @ 1]
word′[Breg @ 2] ← word[trapped @ 2]
word′[Breg @ 3] ← word[trapped @ 3]

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

where trapped *is the address of the stored trapped process for the trap group Areg and priority Creg.*

Error signals:

Will cause a *StoreTrap* trap if enabled.

Comments:

Secondary instruction

See also: *ldtrph ldtrapped sttraph*

sub subtract

Code: FC

Description: Subtract **Areg** from **Breg**, with checking for overflow.

Definition:

$\text{Areg}' \leftarrow \text{Breg} -_{\text{checked}} \text{Areg}$

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

Error signals:

IntegerOverflow can be signalled by $-_{\text{checked}}$

Comments:

Secondary instruction.

See also: *diff add*

sulmul signed times unsigned long multiply

Code: 26 F5

Description: Perform signed long multiplication. This instruction forms the double length product of **Areg** and **Breg**, with **Creg** as carry in, treating the initial value **Areg** as signed and **Breg** as unsigned.

Definition:

$\text{Areg}'_{\text{unsigned}} \leftarrow \text{prod } \mathbf{rem}\ 2^{\text{BitsPerWord}}$

$\text{Breg}' \leftarrow \text{prod} / 2^{\text{BitsPerWord}}$

$\text{Creg}' \leftarrow$ *undefined*

where $\text{prod} = (\text{Breg}_{\text{unsigned}} \times \text{Areg}) + \text{Creg}_{\text{unsigned}}$
– *the value of* prod *is calculated to unlimited precision*

Error signals: none

Comments:
Secondary instruction.

See also: *lmul slmul*

sum sum

Code: 25 F2

Description: Add **Areg** and **Breg**, without checking for overflow.

Definition:

Areg′ ← Breg + Areg

Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *add bsub diff*

swapqueue swap scheduler queue

Code: 60 F0

Description: Swap the scheduling list of priority indicated by **Areg**, where 0 indicates high priority and 1 indicates low priority. **Breg** and **Creg** are the front and back pointers, respectively, of the list to be inserted. The old front and back pointers are returned in **Areg** and **Breg**, respectively.

Definition:

Areg′	←	ProcQueueFPtr[Areg]
Breg′	←	ProcQueueBPtr[Areg]
ProcQueueFPtr′[Areg]	←	Breg
ProcQueueBPtr′[Areg]	←	Creg

Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *insertqueue swaptimer*

swaptimer swap timer queue

Code: 60 F1

Description: Swap the timer list of priority indicated by **Areg** and update the alarm register for the new list. An initial **Areg** of value 0 indicates high priority and 1 indicates low priority. **Breg** is the front pointer of the list to be inserted. The old front pointer is returned in **Areg**.

Definition:

if (Breg ≠ NotProcess.p)
    Tnextreg′[Areg] ← word[Breg @ pw.Time]

Areg′ ← TptrReg[Areg]
Tptrreg′[Areg] ← Breg

Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:
Secondary instruction.

See also: *swapqueue*

talt timer alt start

Code: 24 FE

Description: Start a timer alternative sequence. The **pw.State** location of the workspace is set to *Enabling.p*, and the **pw.TLink** location is set to *TimeNotSet.p*.

Definition:

Enter alternative sequence

word′[Wptr @ pw.State]	←	Enabling.p
word′[Wptr @ pw.TLink]	←	TimeNotSet.p

Error signals: none

Comments:

Secondary instruction.

See also: *alt altend altwt disc disg diss dist enbc enbg enbs enbt taltwt*

taltwt timer alt wait

Code: 25 FI

Description: Wait until one of the enabled guards of a timer alternative is ready and initialize **pw.Temp** for use during the disabling sequence. If the alternative has no ready guard but may become ready due to a timer, place the process onto the timer list.

Definition:

if (word[Wptr @ pw.State] = Ready.p)
    word′[Wptr @ pw.Time] ← ClockReg[Priority]
else if (word[Wptr @ pw.Tlink] = TimeNotSet.p)
{
    word′[Wptr @ pw.State] ← Waiting.p
    deschedule process and wait for one of the guards to become ready
}
else if (word[Wptr @ pw.Tlink] = TimeSet.p)
{
    if (ClockReg[Priority] **after** word[Wptr @ pw.Time]
    {
        word′[Wptr @ pw.State] ← Ready.p
        word′[Wptr @ pw.Time] ← ClockReg[Priority]
    }
    else
    {
        word′[Wptr @ pw.Time] ← (word[Wptr @ pw.Time] + 1)
        insert this process into timer list with alarm time (word[Wptr @ pw.Time] + 1)
        if (*no guards ready*)
        {
            word′[Wptr @ pw.State] ← Waiting.p
            deschedule process and wait for one of the guards to become ready
        }
    }
}
else
    Undefined effect

word′[Wptr @ pw.Temp] ← NoneSelected.o

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.
Instruction is a descheduling point.
Instruction is interruptible.
Uses the **pw.Temp** and **pw.State** slots in the process workspace.

See also: *alt altend altwt disc diss dist enbc enbs enbt talt*

testerr test error flag

Code: 22 F9

Description: Test the error flag at the current priority, returning *false* in **Areg** if error is set, *true* otherwise. It also clears the error flag.

Definition:

if (ErrorFlag[Priority] = *set*)
  Areg′ ← *false*
else
  Areg′ ← *true*
ErrorFlag′[Priority] ← *clear*

Error signals: none

Comments:
Secondary instruction.

See also: *seterr stoperr*

testhalterr test halt-on-error flag

Code: 25 F9

Description: Test HaltOnError mode. If HaltOnError is set then **Areg** is set to *true* otherwise **Areg** is set to *false*.

Definition:

if (HaltOnError = *set*)
    Areg′ ← *true*
else
    Areg′ ← *false*

Breg′ ← Areg
Creg′ ← Breg

Error signals: none

Comments:
Secondary instruction.

See also: *clrhalterr sethalterr*

testpranal test processor analysing

Code: 22 FA

Description: Push *true* onto the stack if the processor was analyzed when the processor was last reset, or *false* otherwise.

Definition:

if (*analyse asserted on last reset*)
  Areg′ ← *true*
else
  Areg′ ← *false*

Breg′ ← Areg
Creg′ ← Breg

Error signals: none

Comments:
Secondary instruction.

timeslice timeslice

Code: 60 F3

Description: Cause a timeslice, putting the current process on the back of the scheduling list and executing the next process. If the scheduling list is empty then this instruction acts as a no-operation.

Definition:

if (*scheduling list empty*)
    no effect
else
{
    Put current process on back of list
    Start next process
}

Areg′ $\leftarrow$ *undefined*
Breg′ $\leftarrow$ *undefined*
Creg′ $\leftarrow$ *undefined*

Error signals: Can cause a timeslice trap to be signalled.

Comments:

Secondary instruction.
This instruction works at high and low priorities.
This instruction is unaffected by disabling timeslice.
Instruction is a descheduling point.

tin timer input

Code: 22 FB

Description: If **Areg** is after the value of the current priority clock, deschedule until the current priority clock is after the time in **Areg**.

Definition:

if not (ClockReg[Priority] **after** Areg)
{
  word′[Wptr @ pw.State] ← Enabling.p
  word′[Wptr @ pw.Time] ← (Areg + 1)
  Insert process into timer list with time of (Areg + 1) *and start next process*
}

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.
Instruction is a descheduling point.
Instruction is interruptible.
Uses **pw.State** slot in the process workspace.

See also: *enbt dist ldtimer talt taltwt*

trapdis trap disable

Code: 60 F6

Description: Disable those traps selected by the mask in **Areg** at the priority selected by **Breg**, where 0 indicates high priority and 1 indicates low priority. The original value of TrapEnables is returned in **Areg**.

Definition:

$TrapEnables'[Breg] \leftarrow TrapEnables[Breg] \wedge \sim Areg$
$Areg'_{13..0} \leftarrow TrapEnables[Breg]$
$Areg'_{31..14} \leftarrow 0$

$Breg' \leftarrow$ *undefined*
$Creg' \leftarrow$ *undefined*

Error signals: none

Comments:
Secondary instruction.

See also: *trapenb*

trapenb trap enable

Code: 60 F7

Description: Enable those traps selected by the mask in **Areg** at the priority selected by **Breg**, where 0 indicates high priority and 1 indicates low priority. The original value of TrapEnables is returned in **Areg**.

Definition:

TrapEnables′[Breg] $\leftarrow$ TrapEnables[Breg] $\vee$ Areg
$Areg'_{13..0}$ $\leftarrow$ TrapEnables[Breg]
$Areg'_{31..14}$ $\leftarrow$ 0

Breg′ $\leftarrow$ *undefined*
Creg′ $\leftarrow$ *undefined*

Error signals: none

Comments:
Secondary instruction

See also: *trapdis*

tret trap return

Code: 60 FB

Description: Return from a trap handler.

Definition:

GlobalInterruptEnables′	←	word[traphandler @ 0]$_{16..23}$
TrapEnables[Priority]	←	word[traphandler @ 0]$_{0..13}$
Status′	←	word[traphandler @ 1]
Wptr′	←	word[traphandler @ 2]
Iptr′	←	word[traphandler @ 3]

where traphandler *is the address of the traphandler for the trap group of the current handler and the current priority.*

Error signals: none

Comments:
Secondary instruction.

See also: *ldtraph sttraph*

unpacksn unpack single length fp number

Code: 26 F3

Description: Unpack a packed IEEE single length floating point number. **Areg** initially holds the packed number, and the instruction returns the exponent in **Breg** and the fractional field in **Areg**, not including the implied most significant bit for normalised numbers. In addition a code indicating the type of number is added to 4 times the initial value of **Breg** and left in **Creg.** The codes are**:**

0	if **Areg** is zero
1	if **Areg** is a denormalised or normalised number
2	if **Areg** is an infinity
3	if **Areg** is not-a-number

Definition:

Areg′ $\leftarrow$ fractional field contents of Areg
Breg′ $\leftarrow$ exponent field contents of Areg
Creg′ $\leftarrow$ 4 $\times$ Breg + ‘code’ of type of Areg (see above)

Error signals: none

Comments:

Secondary instruction.

See also: *roundsn postnormsn*

wait wait

Code: 60 F5

Description: Wait (or P) on the semaphore pointed to by **Areg**. If the semaphore count is greater than zero then the count is decremented and the process continues; otherwise the current process is descheduled and added to the back of the semaphore list.

Definition:

if (word[Areg @ s.Count] = 0)
{
    Put process on back of semaphore list
    Start next process
}
else
    word′[Areg @ s.Count] ← word[Areg @ s.Count] – 1

Areg′ ← *undefined*
Breg′ ← *undefined*
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.
Instruction is a descheduling point.

See also: *signal*

wcnt word count

Code: 23 FF

Description: Convert the byte offset in **Areg** to a word offset and a byte selector.

Definition:

Areg′ ← (Areg ∧ WordSelectMask) / BytesPerWord
Breg′ ← Areg ∧ ByteSelectMask

Creg′ ← Breg

Error signals: none

Comments:

Secondary instruction.

wsub word subscript

Code: FA

Description: Generate the address of the element which is indexed by **Breg**, in the word array pointed to by **Areg**.

Definition:

Areg′ ← Areg @ Breg

Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:
Secondary instruction.

See also: *bsub ldlp ldnlp ssub wcnt wsubdb*

wsubdb form double word subscript

Code: 28 F1

Description: Generate the address of the element which is indexed by **Breg**, in the double word array pointed to by **Areg**.

Definition:

Areg′ ← Areg @ (2 × Breg)

Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

See also: *bsub ssub wcnt wsub*

xbword sign extend byte to word

Code: 2B F8

Description: Sign-extend the value in the least significant byte of **Areg** into a signed integer.

Definition:

$Areg'_{0..7} \leftarrow Areg_{0..7}$

$Areg'_{8..BitsPerWord-1} \leftarrow Areg_{7}$

Error signals: none

Comments:

Secondary instruction.

xdble extend to double

Code: 21 FD

Description: Sign extend the integer in **Areg** into a double length signed integer.

Definition:

if (Areg $\geq$ 0)
  Breg′ $\leftarrow$ 0
else
  Breg′ $\leftarrow$ –1

Creg′ $\leftarrow$ Breg

Error signals: none

Comments:
Secondary instruction.

xor exclusive or

Code: 23 F3

Description: Bitwise exclusive **or** of **Areg** and **Breg**.

Definition:

Areg′ ← Breg ⊗ Areg

Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:

Secondary instruction.

xsword sign extend sixteen to word

Code: 2F F8

Description: Sign extend the value in the least significant 16 bits of **Areg** to a signed integer.

Definition:

$Areg'_{0..15} \leftarrow Areg_{0..15}$
$Areg'_{16..BitsPerWord-1} \leftarrow Areg_{15}$

Error signals: none

Comments:
Secondary instruction.

See also: *xbword*

xword extend to word

Code: 23 FA

Description: Sign extend an N-bit signed number in **Breg** into a full word. To indicate the value of N, bit N-1 of **Areg** is set to 1; all other bits must be 0.

Definition:

if (Areg *is a not power of 2*)
  Areg′ ← *undefined*
else if (Areg = MostNeg) – *N is BitsPerWord*
  Areg′ ← Breg
else if (Breg ≥ 0) **and** (Breg < Areg) – *Breg N bits and positive*
  Areg′ ← Breg
else if (Breg ≥ Areg) **and** ((Breg >> 1) < Areg) – *Breg N bits and negative*
  Areg′ ← Breg – (Areg << 1)
else – *Breg more than N bits*
  Areg′ ← *undefined*

Breg′ ← Creg
Creg′ ← *undefined*

Error signals: none

Comments:
Secondary instruction.

See also: *xbword xsword*

Index

SGS-THOMSON MICROELECTRONICS

SGS-THOMSON Microelectronics GROUP OF COMPANIES
Australia - Brazil - Canada - China - France - Germany - Hong Kong - Italy - Japan - Korea - Malaysia - Malta - Morocco - The Netherlands - Singapore - Spain - Sweden - Switzerland - Taiwan - Thailand - United Kingdom - U.S.A.

Document number: 72-TRN-273-01