User’s Manual

IMAPCAR Series Processor

1DC LANGUAGE SPECIFICATIONS

Software

Document No. U20036EE1V0UM00

Date Published: Sept 2009

© NEC Electronics (Europe) GmbH

Legal Notes

The information in this document is current as of September 2009. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.

- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.
- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".
  The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.
  "Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.
  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).
  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

The quality grade of NEC Electronics products is "Standard" unless otherwise expressly specified in NEC Electronics data sheets or data books, etc. If customers wish to use NEC Electronics products in applications not intended by NEC Electronics, they must contact an NEC Electronics sales representative in advance to determine NEC Electronics' willingness to support a given application.

(Note)

(1) "NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.

(2) "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

Table of Contents

1 PREFACE 4
1.1 SIMD Mode 4
1.2 MP Mode..5
1.3 Mixed mode..6
1.4 Hardware Configuration Example (for XC Core) 6
2 1DC SPECIFICATION EXPANSION FROM C 7
2.1 Additional Data Attributes 8
2.2 Additional Operators 11
2.3 Additional Syntax 14
3 Revision history 15

1 PREFACE

1DC (One Dimensional C) is an expanded implementation of C designed for programming parallel processor systems in which many PEs (Processing Elements) and memory blocks are linked together one dimensionally (Integrated Memory Array Processor, or IMAP, systems).

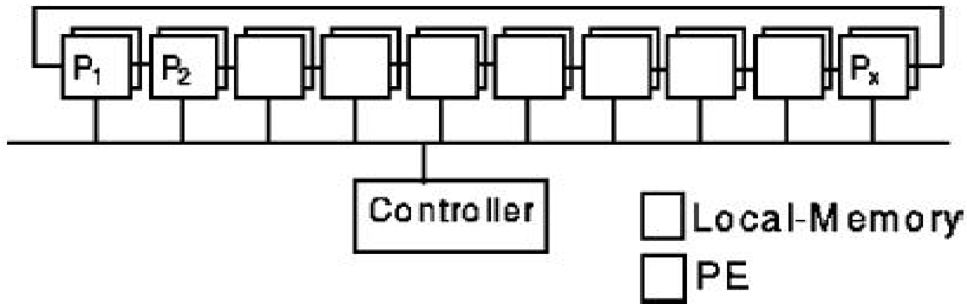

Figure 1 Example IMAP System Configuration

IMAP systems include two main operating modes: the SIMD (Single Instruction Multiple Data) mode, and the MP (Multi Processor) mode. A simple overview of how to use 1DC in each of these modes is provided below, and then the 1DC language specifications are described.

1.1 SIMD Mode

The figure below shows a configuration in which an IMAP system is used as an SIMD parallel processor. Here, *IMEM* refers to memory blocks allocated to each PE

Figure 2 IMAP System Configuration in SIMD Mode

When using an IMAP system in the SIMD mode, the processing for a certain collection of data is normally assigned to each PE. In this mode, each PE has to execute the same instructions broadcast from the controller (the control processor, or CP), but each PE can access the memory block (local memory) where the data it is in charge of is stored using high speed, unique addresses, and data can be directly exchanged between PEs by using a network (without having to wait). To the user, the SIMD mode makes it seem as though there is a working 2D memory area that has extremely little access overhead.

1DC is an IMAP system programming language designed to enable the design of parallel algorithms for operating in this memory area and to minimize the expansion of standard C specifications as much as possible. The following figure shows an operational overview of a parallel algorithm that uses the 2D memory area.

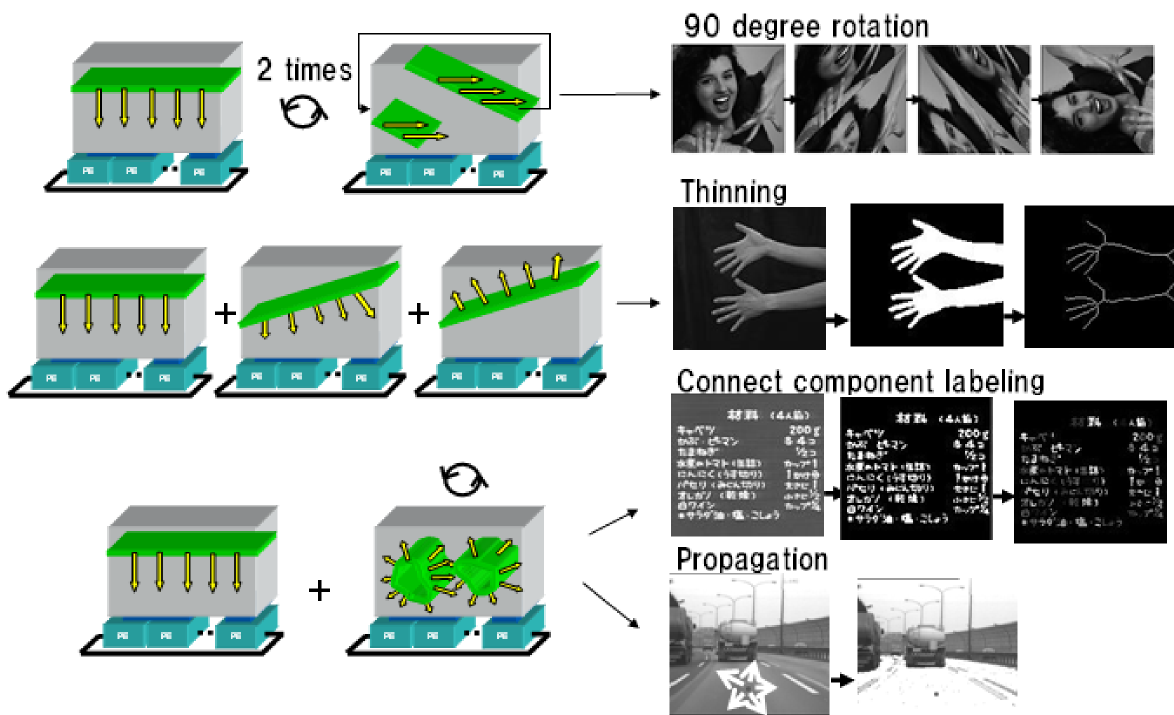

Figure 3 Operational Overview of Parallel Algorithm Using 2D Memory Area

1.2 MP Mode

When using an IMAP system in the MP mode, the system runs in a multi-processor configuration in which multiple PEs are grouped into one PU (Processing Unit). Because each PU runs on its own program counter, waiting is required when exchanging data between PUs, and message communication and non-cached external variable access methods are used, but different processing can be performed for each PU in parallel. This multi-processor parallel processing is programmed so that it is available without using the portion of the 1DC specifications outside the range of standard C.

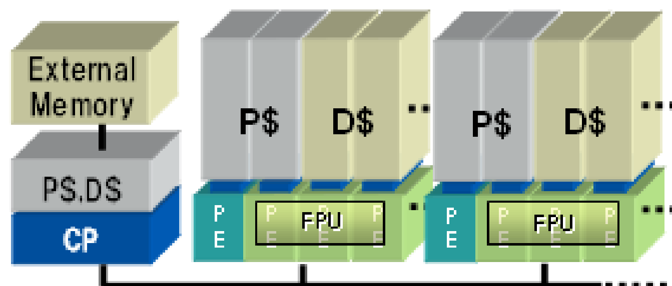

Figure 4 IMAP System Configuration in MP Mode (When Each PU Consists of 4 PEs)

1.3 Mixed mode

An IMAP system can also be used in the MIXED mode, in which some PEs run in the SIMD mode, and the rest run as multiple PUs in the MP mode. When using the MIXED mode, tasks (functions) that run in the SIMD mode are programmed using the portion of 1DC that is expanded from standard C, while those that run in the MP mode are programmed using standard C. Data exchanges between PUs running in the same operating mode are performed through the CP or through external memory.

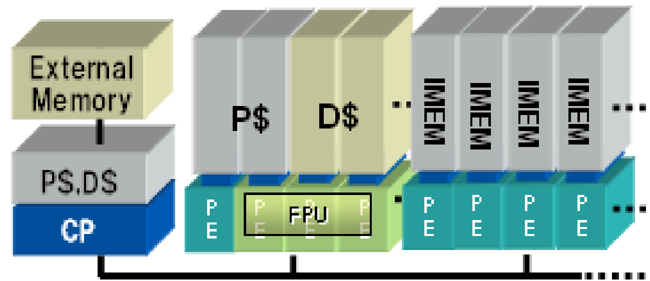

Figure 5 . IMAP System Configuration in MIXED Mode (When Each PU Consists of 4 PEs)

1.4 Hardware Configuration Example (for XC Core)

The figure below shows the configuration of an XC core, an example of an actual IMAP system. Each tile is made up of 8 PEs, which are grouped into a higher 4 PEs and a lower 4 PEs, and the user can select whether each of these groups runs as a PU or as 4 separate PEs. No 4-PE group runs as a PU in the SIMD mode, all 4-PE groups run as PUs in the MP mode, and only lower 4-PE groups run as PUs in the MIXED mode.

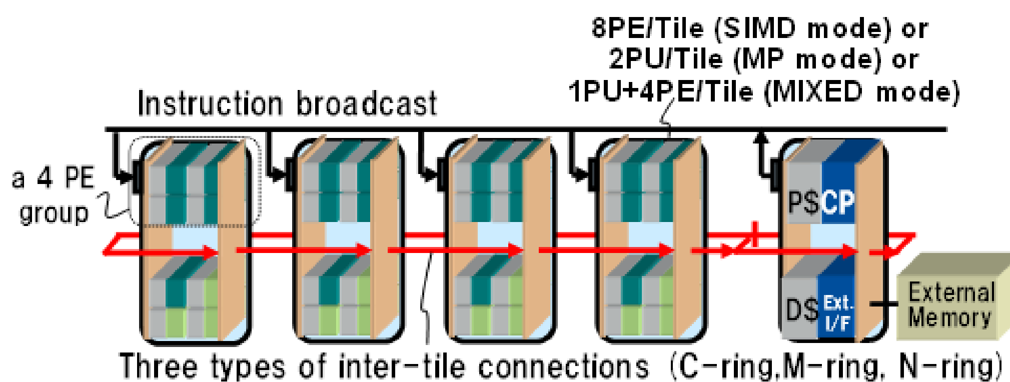

Figure 6 Example Physical Configuration of IMAP System (for XC Core)

2 1DC SPECIFICATION EXPANSION FROM C

Compared to C, the 1DC specifications have been expanded as follows.

- Additional data attributes
    - Data allocation attributes
        - uni (default)
        - multi
        - outside qualifier
        - align qualifier
    - Data value attributes
        - common (default)
        - sep (or separate)
    - Pointers to multi sep or sep data

- Additional operators
    - Operators for transferring data between PEs: :> and :<
    - Operator for selecting PEs: :[and :] (used as a pair)
    - Logical operators for PE data: :|| and :&&
    - Operator for initializing PE data: :(and :) (used as a pair)
    - PE data assignment operator: :=

- Additional syntax
    - PE selecting conditional statements 1: mifa and melsea
    - PE selecting conditional statements 2: mif and melse
    - PE selecting loop statement: mwhile
    - PE selecting loop statement: mfor
    - PE selecting loop statement: mdo

These expanded specifications are described in the above order below.

2.1 Additional Data Attributes

Data allocation attributes	

uni default)	The *uni* attribute is the default in 1DC for data allocation. Data that has this attribute actually consists of only one element. *uni* data is assigned to the memory for the controller that controls all PEs, which enables the compiler to broadcast this data to all PEs at compile time as necessary. Therefore, the *uni* data has the same value when viewed from each PE. To give a variable the *uni* attribute, either specify the *uni* type qualifier before the variable data type when declaring the variable, or declare the variable without the type qualifier.
	
	int x; // Declares the 2-byte signed integer x for the controller.
	uni int x; // Declares the 2-byte signed integer x for the controller (as above).

multi	Data that has the *multi* attribute has as many elements as there are LPA PEs and is assigned to the local memory for the PE array of the LPA. To give a variable the *multi* attribute, specify the *multi* type qualifier before the variable data type when declaring the variable. However, if the value attribute of a variable is specified using the *sep* type qualifier described below, the variable is automatically given the *multi* attribute, so the *sep* type qualifier is normally sufficient for declaring *multi*-type data.

outside qualifier	The *outside* type qualifier is used to directly assign data to external memory. To give a variable the *outside* attribute, specify the outside type qualifier before the variable data type and other qualifiers as follows when declaring the variable:
	
	outside sep int a[20],b; // Declares the sep array a and sep
	// variable b, which have the outside attribute.
	
	outside int c; // Declares a uni-type variable that has
	// the outside attribute.
	outside int d[10]; // Declares a uni-type array that has
	// the outside attribute.
	
	All arithmetic operations can be used for *uni*-type variables and arrays that have the outside attribute, but declaring a *sep*-type variable or array that has the outside attribute only allocates memory for the data, and the data must be explicitly transferred from the external memory to the PE-local memory in order to actually reference the data. The data can be transferred using the standard *IxEmemrd* and *IxEmemwr* functions as shown in the following example:
	
	outside sep int owrk[VLINES]; // sep array owrk that has the
	// outside attribute
	sep int src[VLINES]; // sep array src in the PE array memory
	int i;
	IxEmemrd(src,work*PENO,VLINES*2); // Copies the data in the
	// work array to the src array.
	for(i=0; i<VLINES; i++)
	src[i] = ...; // Processes the data in the src array.
	IxEmemwr(src,work*PENO,VLINES*2); // Writes the data in the
	// src array back to the work array.

align qualifier	The *align* type qualifier is used to allocate memory for an array, starting at the beginning of a memory page (using a hardware dependent number of bytes) and is only meaningful for a global array. To give a global array the *align* attribute, specify the *align* type qualifier before the variable data type and other qualifiers when declaring the array.
	
	align sep int tbl[256]; // Declares a sep array that has
	// the align attribute.
	sep int a,idx; // Declares a sep variable.

Data value attributes	

common (default)	The *common* attribute is the default in 1DC for data values. If data that has this attribute has multiple elements, the values of each of those elements are assumed to be the same. To give a variable the *common* attribute, either specify the *common* type qualifier before the variable data type when declaring the variable, or declare the variable without the type qualifier.
	
	int x; // Declares the 2-byte signed integer x for the controller.
	common int x; // Declares the 2-byte signed integer x for the PE array.
	
	However, because IMAP Series processors can efficiently broadcast data from the PE array controller to all PEs, declaring a *uni* type *common* variable and assigning it to the controller memory, instead of declaring a *multi* type variable and assigning it to the PE array internal memory, can reduce the amount of consumed PE memory without reducing execution efficiency. Therefore, *multi* type *common* variables are not normally required.

sep (separate)	*sep* data (data that has the *sep* attribute) refers to data for which the value of each element might differ, assuming the data has multiple elements. To declare a *sep* variable, specify the *sep* or *separate* type qualifier before the variable data type when declaring the variable. *sep* data has the *multi* attribute by default, and variables to be processed by the PE array are normally declared simply as sep variables.
	
	int x; // Declares the 2-byte signed integer x (that
	// has the uni attribute) for the controller.
	sep char y; // Declares the 1-byte sep variable y.

<table>
<tr><th colspan="2">Pointers to data with added attributes and structure specification method</th></tr>
<tr><td>Pointers</td><td>Pointers to data that has one of the added attributes described above can be used in 1DC. The following examples show how the meaning changes depending on where the sep type qualifier is specified when declaring a variable:

sep char *p; // Declares p, a pointer to sep char data.
char * sep p; // Declares p, a sep variable that stores a
// pointer to char data.
sep char * sep p; // Declares p, a sep variable that stores
// a pointer to sep char data.
sep int (*f)(); // Declares f, a pointer to a function that
// returns a sep int value.</td></tr>
<tr><td>Structures</td><td>Normally, the declarations for structure members and the actual data allocation attributes and value attributes that are used are specified separately. The data allocation attributes and value attributes cannot be specified when declaring the members.

struct ab {
int a;
unsigned char b[20];
};
sep struct ab sep_ab; // Declares a structure for the PE array.
struct ab uni_ab; // Declares a structure for the controller.
outside struct ab ouni_ab; // Declares a uni-type structure for
// the external memory.
outside sep struct ab osep_ab; // Declares a sep-type structure
// for the external memory

An added attribute can also be directly specified when declaring a structure as follows:
sep struct ab { // Declares a structure for the PE array.
int a;
unsigned char b[20];
} sa</td></tr>
</table>

2.2 Additional Operators

Operators for transferring data between PEs: :> and :<	In the PE array of the LPA, :> is used as a unary operator to reference the sep data in the adjacent PE on the left, and :< is used similarly for the adjacent PE on the right. In addition, :> is used as a binary operator to reference the sep data in the nth PE to the left of the current PE, and :< is used similarly for the nth PE to the right. This operator actually transfers data one adjacent PE at a time, and performs n transfers to transfer data a distance of n PEs.
	
	int n;
	sep unsigned char x, y;
	
	y = x :< n; // Assigns the x value of the PE n PEs to the
	// right of the current PE to y of the current PE.
	y = x :< (n+1); // Assigns the x value of the PE n + 1 PEs
	// to the right of the current PE to y of the current PE.
	x = x :> (n+1); // Assigns the x value of the PE n + 1 PEs
	// to the left of the current PE to x of the current PE.
	x = :< x; // Assigns the x value of the adjacent PE to
	// the right of the current PE to x of the current PE.

Operator for selecting PEs: :[and :]	This operator is used to select a PE in the PE array, send sep data in the PE to the controller, and then assign that data to a specified PE or broadcast the data to all PEs.
	
	sep int x;
	int n,y;
	*x:[n:] = x:[2*n+1:]; // Assigns the x value of PE (2 * n + 1) to x of the nth PE.*
	*y = 2 * x:[n:]; // Multiplies the x value of the nth PE*
	// by 2 and assigns the result to y of the controller.
	x = x:[1:] // Assigns the x value of the first PE to
	// the x of all PEs (by broadcasting to all PEs).

<table>
<tr>
<td>Logical operators for PE data: :|| and :&&</td>
<td>These operators are used to calculate the logical disjunction (:||) or the logical conjunction (:&&) of the sep data of unmasked (active) PEs. In general, these operations are efficiently implemented by LPA hardware that reads out the result of calculating the logical disjunction of the values of the status registers (1 bit) for each PE.
sep int a,b;
int c;
c = :||(a || b); // Calculates the logical disjunction of
// a and b for each PE, calculates the logical
// disjunction of this result for all PEs,
// and then assigns the final result to c of the controller.
c = :&&a; // Calculates the logical conjunction of a for
// all PEs, and then assigns this result to c of the controller.</td>
</tr>
<tr>
<td>Operator for initializing PE data: :(and :)</td>
<td>This operator is used to assign different sep data constants to PEs. This operator can be used to easily specify sep constants. Only constants can be specified between :(and :), and multiple constants must be separated with commas. If the constant c0 is followed by : and another constant c1, c0 is assigned to c1 PEs. If less constants are specified between :(and :) than the number of PEs, 0 is automatically assigned to the remaining PEs. However, if a comma (,) is specified after the last constant cn, cn is automatically assigned to the remaining PEs instead. Constant propagation is performed for this operator at compile time.
For the following examples, it is assumed that there are 8 PEs and that the system sep constant PENUM, which stores the PE number of each PE (0 for the leftmost PE), has been defined:

sep int x = :(0,1,2,3,4 :); // The operator can also be used when initializing a
//sep variable during declaration.

x = :(0,1,2,3,4 :); // Equivalent to x = :(0,1,2,3,4,0,0,0 :)
x = :(0,1,2,3,4, :) // Equivalent to x = :(0,1,2,3,4,4,4,4 :)
x = :(0:2, 1:3, 4 :); // Equivalent to x = :(0,0,1,1,1,4,0,0 :)
x = PENUM;</td>
</tr>
</table>

<table>
<tr>
<td>PE data assignment operator: :=</td>
<td>This assignment operator specifies that a value be assigned to sep data for all PEs regardless of the current context. This operator is used to explicitly assign sep data to all PEs. For details about how the context is defined, see the next section.

sep int a,b;
a := b; // Assigns the b value of all PEs in the PE array
// to the a of all PEs.</td>
</tr>
<tr>
<td>Order of operations</td>
<td>The order of operations for 1DC operators is based on the order of operations for C. The following shows the order of operations for 1DC operators and their associativity.
<table>
<tr><th>Operators</th><th>Associativity</th></tr>
<tr><td>:(:) :[:] () [] -> .</td><td>Left to right</td></tr>
<tr><td>! ~ ++ -- - (type) * & sizeof :< :> :|| :&&</td><td>Right to left</td></tr>
<tr><td>* / %</td><td>Left to right</td></tr>
<tr><td>+ -</td><td>Left to right</td></tr>
<tr><td><< >> :< :></td><td>Left to right</td></tr>
<tr><td>< <= > >=</td><td>Left to right</td></tr>
<tr><td>== !=</td><td>Left to right</td></tr>
<tr><td>&</td><td>Left to right</td></tr>
<tr><td>^</td><td>Left to right</td></tr>
<tr><td>|</td><td>Left to right</td></tr>
<tr><td>&&</td><td>Left to right</td></tr>
<tr><td>||</td><td>Left to right</td></tr>
<tr><td>?:</td><td>Right to left</td></tr>
<tr><td>= += -= := etc.</td><td>Right to left</td></tr>
<tr><td>,</td><td>Left to right</td></tr>
</table>
</td>
</tr>
</table>

2.3 Additional Syntax

PE selecting conditional statements: *mif* and *melse*	The syntax of *mif* and *melse* statements is the same as that of C *if* and *else* statements: *mif* (expression-1) *statement-block-1* *melse* *statement-block-2* Remark The *melse* statement is optional However, the value of expression1 must have the *sep* attribute, *statement-block-1* is executed only for the group of PEs (or PE context) for which the value of *expression-1* is not zero, and *statement-bloc-2* is executed only for the group of PEs (or PE context) for which the value of *expression-1* is zero. Note that, when these statements are used in a function, the PE context that is used is calculated by finding the logical conjunction of the PE context for the caller of the function and the PE context for *expression-1.*
PE selecting conditional statements: *mifa* and *melsea*	The syntax of *mifa* and *melsea* statements is the same as that of the previously described *mif* and *melse* statements: *mifa* (expression-1)*statement-block1* *melsea* *statement-block2* Remark The *melsea* statement is optional However, the value of *expression-1* must have the *sep* attribute. *mifa* and *melsea* differ from *mif* and *melse* in that, while *mif* and *melse* use the context calculated by finding the logical conjunction of the current context and the context for *expression-1*, *mifa* and *melsea* use the context for *expression-1* regardless of the current context. The *mifa* statement generates code that is more efficient than that of the *mif* statement because *mifa* does not consider nesting. Therefore, using *mifa* is faster if there is no need to nest masking conditions
PE selecting loop statement: *mwhile*	The syntax of the *mwhile* statements is the same as that of C while statements: *mwhile* (expression-1) *statement-block-1* However, the value of *expression-1* must have the sep attribute, and *statement-block-1* is executed only for the group of PEs (or PE context) for which the value of *expression-1* is not zero. For the *mwhile* statement, *statement-block-1* is executed in the context based on the (truth) value of *expression-1* during the first iteration, and, during the nth iteration (n = 2…), all operations for *statement-block-1* and *expression-1* (except assignment operations using :=) are performed only in the context determined according to the value of *expression-1* during the previous iteration.

PE selecting loop statement: *mfor*	The syntax of *mfor* statements is the same as that of C for statements:
	mfor (*expression-1*; *expression-2*; *expression-3*) *statement-block-1*
	
	This is equivalent to the following *mwhile* statement:
	expression-1;
	mwhile (*expression-2*) {
	statement-block-1
	expression-3;
	}

PE selecting loop statement: *mdo* and *mwhile*	The syntax of *mdo* and *mwhile* statements is the same as that of C *do* and *while* statements: *mdo statement-block-1 mwhile* (*expression-1*); However, the value of *expression-1* must have the *sep* attribute, and *statement-block-1* is executed during the first iteration regardless of the context determined using *expression-1*.

3 Revision history

Version	Date	Document Number	Description
1.0	Sept 2009	U20036EE1V0UM00	First version

The following revision list shows all functional changes compared to the previous version.

Chapter	Page	Description