ODM CHILI

Programmer's Guide

V020

ODM 301000106

Contents

Figures

Examples

Tables

Abbreviations

DMA	Direct Memory Access
DMS	Data Memory Subsystem
DRAM	Dynamic Random Access Memory
DE	Decode stage of instruction pipeline.
EP	Expand stage of instruction pipeline.
EX	Execute stage of instruction pipeline.
FE	Fetch stage of instruction pipeline.
IVS	Immediate Value Section
IW	Instruction Word
MEM	Memory read stage of instruction pipeline.
OCS	Operation Code Section
PC	Program Counter
PO	Post stage of instruction pipeline.
R/W	Read-Write
SIMD	Single Instruction, Multiple Data
VLC	Variable Length Coding
VLIW	Very Long Instruction Word
WB	Writeback stage of instruction pipeline.

Preface

About this Document

This document provides reference information for ON DEMAND Microelectronics' (ODM) CHILI. It describes the processor's instruction set architecture and its programming model.

How this Document is Organized

This document contains the following chapters and sections:

- ***Introduction*** on page 3 introduces the ODM **CHILI** and describes its key features.
- **CHILI *Architecture*** on page 4 describes the main functional blocks.
- ***Program Control*** on page 7 gives details about the instruction pipeline, the assignment of parallel computational units (slots), conditional executions, and DMA programming.
- ***Instruction Set*** on page 21 describes all commands of the instruction set and its SIMD extensions.
- ***Preprocessor*** on page 90 describes the preprocessor.
- ***Appendix A: Instruction Set Quick Reference*** on page 92 lists all available instructions of the **CHILI**.
- ***Appendix B: Preprocessor Commands*** on page 95 lists all commands of the preprocessor.

Intended Audience

The information herein is targeted at system software developers, hardware designers and application engineers with experience in assembler programming.

Typographical Conventions

This document uses the following typographical conventions:

`code`	Single lines of code or instructions are written in a `monospaced` typewriter font.
italics	*Italics* are used to indicate placeholders or for emphasis.
UPPERCASE	Keywords are written in UPPERCASE.
{}	Braces indicate the beginning and ending of instructions executed within one cycle.
`Rn`	Instruction syntax indicating a destination general-purpose register.
`Rx, Ry`	Instruction syntax indicating source general-purpose registers.

Contact Information

To learn more about ON DEMAND Microelectronics and our products and services or to find one of our offices near you contact our headquarters or visit our web site:

Corporate Headquarters

ON DEMAND Microelectronics
Wagramer Str. 17-19, IZD Tower
AT-1220 Vienna/Austria
Phone: +43 (1) 269 79 85-0
Fax: +43 (1) 269 79 85-200

Web Site

http://www.odm.at

CHAPTER 1

Introduction

In This Chapter

1.1 What is the CHILI?

The **CHILI** is a multimedia processor specifically designed for the following tasks:

- video decoding and encoding
- audio decoding and encoding
- image processing

The **CHILI** combines low power consumption with high performance and high data throughput.

1.1.1 CHILI Configuration

The **CHILI** configuration parameters and their values are described in the following table.

Table 1: Configuration Parameters

Configuration Parameter	Value
Number of slots	4
Register File Size	64 registers
Data Path Width	32 bits
Data Memory Size	Memory space 32 bits
Code Memory Size	Memory space 32 bits

CHAPTER 2

CHILI Architecture

In This Chapter

2.1 CHILI System Architecture

The **CHILI** communicates with external memory (DRAM) and I/O devices through the system bus interface. Internally, the **CHILI** consists of a core and a data memory subsystem (DMS) plus other system components. The DMS can be perceived as an additional layer between the core (CPU) and the core memory (RAM). The DMS handles, among other things, the DMA transfers between the **CHILI** core memory and external memory and performs non-blocking memory load operations to reduce CPU overhead.

The following figure shows the system architecture of the **CHILI**.

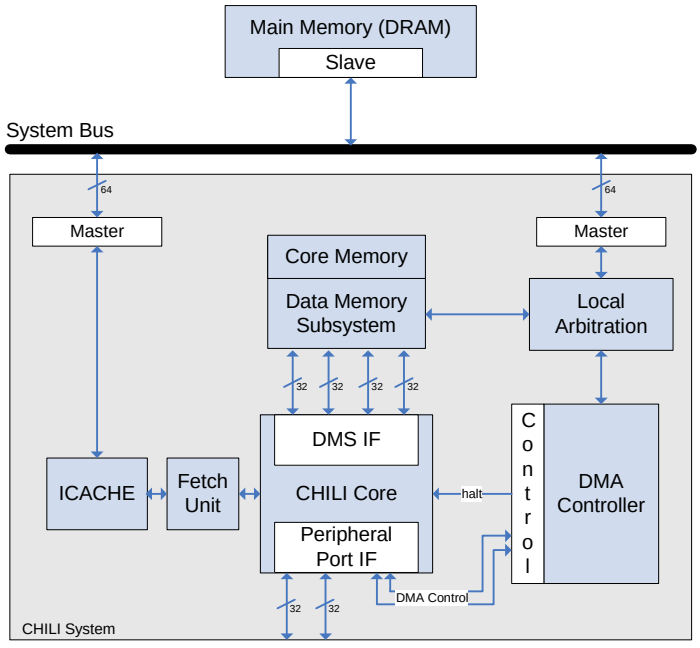

Figure 1: CHILI System Block Diagram

The following table describes the main functional units of the **CHILI** system.

Table 2: CHILI System Architecture Legend

Functional Unit	Description
CHILI Core	The central processing unit.
DMS IF	A four-ported interface to the **CHILI** data memory subsystem.
Peripheral Port IF	Provides a four-ported interface to the DMA Control interface and/or alternative peripherals.
DMA Control	A memory-mapped port interface to the DMA Controller.
DMA Controller	A functional unit that handles the internal and external memory access of the **CHILI**.
Data Memory Subsystem	Includes the **CHILI** core memory and does the following: ▪ Performs out-of-order command execution ▪ Performs non-blocking memory load operations ▪ Supports memory-mapped I/O ▪ Performs block data transfer between internal (core RAM) and external memory (DRAM) ▪ Provides direct access of the **CHILI** core to external memory ▪ Supports single DMA transfers (one request at a time) ▪ Allows predefined DMA transfers
Core Memory	The internal data memory of the **CHILI**.
Main Memory (DRAM)	External memory accessible through a 64-bit read/write master port.
ICACHE	Instruction cache unit.
Fetch Unit	Fetches instructions from ICACHE or external memory.

2.2 CHILI Slot Architecture

The **CHILI** architecture is based on a VLIW RISC engine and can be described as a load-store architecture. It supports the execution of up to four instructions in parallel. In contrast to other VLIW architectures there are no restrictions concerning the choice of instructions within an execution unit. An execution unit of the **CHILI** is called slot. Each slot provides identical functionality (arithmetic instructions, memory access, control flow instructions), and every second slot is dedicated to execute test instructions for the evaluation of conditions. The following figure shows a diagram of the **CHILI** slots.

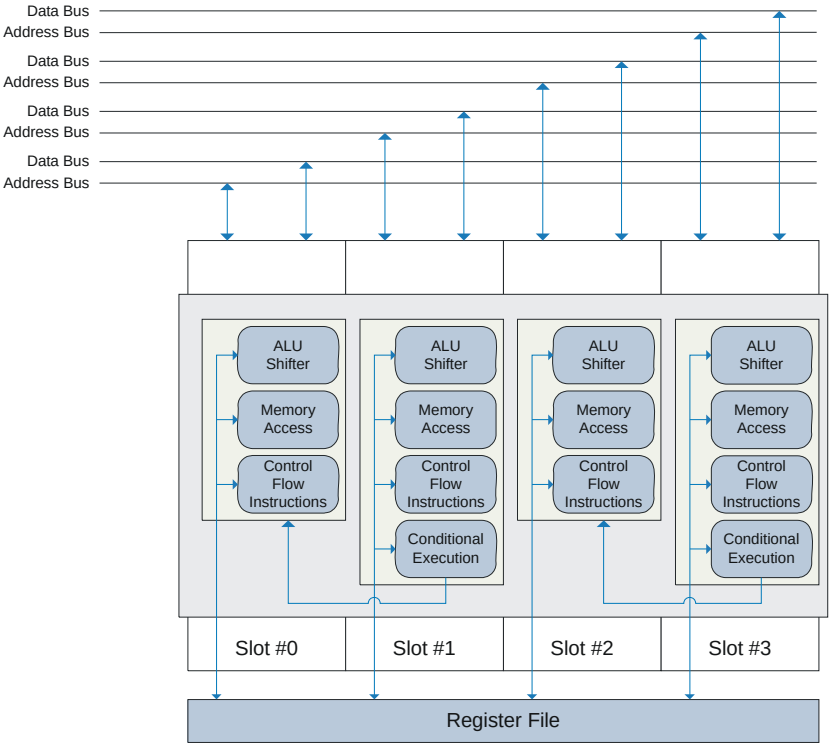

Figure 2: CHILI Slot Diagram

The following table describes the main functional units of the **CHILI** slots.

Table 3: CHILI Slot Architecture Legend

Functional Unit	Description
Slot #0-3	The parallel computational units.
ALU/Shifter, Memory Access, Control Flow Instructions	Identical logical units for each slot.
Conditional Execution	Units for executing test instructions on every second slot.
Data Bus	Port-interfaced buses provided for each slot.
Address Bus	Port-interfaced buses provided for each slot.
Register File	Holds 64 32-bit general purpose registers (r0-r63).

CHAPTER 3

Program Control

In This Chapter

3.1 Instruction Pipeline

An instruction pipeline consists of a sequence of operations that occur during the execution of an instruction. The **CHILI** pipeline has the following stages:

- *Fetch (FE0 & FE1):* The two stages fetch instruction data from the instruction cache or the program memory (located in the main memory, or DRAM).
- *Expand (EP)*: Expands the VLC-encoded instruction words.
- *Decode (DE)*: Decodes the current instruction and provides information for the selection of operands from the register file and for the calculation of forwarding data.
- *Forward (FW)*: Based on the forwarding information of the decode stage the forward stage replaces the operands selected in the decode stage with the registered results of the subsequent pipeline stages.
- *Execute (EX)*: Performs the arithmetic operations, assigns addresses and data for memory read/write, and performs test operations (conditions).
- *Memory (MEM)*: Retrieves data from the data cache for memory read operations.
- *Writeback (WB)*: Stores the results of previous operations in the register file.
- *Post (PO)*: Forwards data that is written in the register file.

Typically, the pipeline is full with a sequential set of instructions, each at a certain stage. When a PC discontinuity occurs, such as during a branch, call, or return, one or more stages of the pipeline may be temporarily unused.

The pipeline architecture is shown in the following figure.

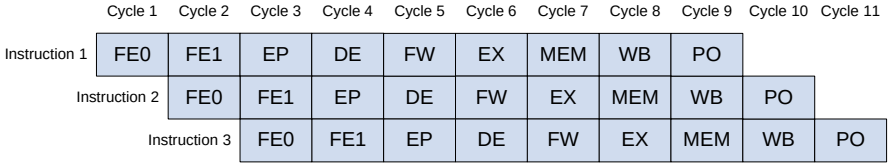

Figure 3: Instruction Pipeline

Note After a jump the instruction pipeline will be emptied, resulting in a jump cycle penalty of five cycles.

3.2 Conditional Execution

The main characteristic of conditional execution in the **CHILI** is the omission of flags. Other than commonly implemented conditions are evaluated “on the fly”. This results in efficient execution of control code with many comparisons. Conditions are used both in conditional jumps and conditional execution.

Syntax

```
if (condition) command;
```

3.2.1 Assignment of Processing Slots

Each conditional command requires two adjacent slots for processing. Therefore, in a four-slot **CHILI**, conditional execution is assigned either to slots 0 and 1 or slots 2 and 3. Commands within braces (“{“, “}”) are processed within one clock cycle. For a four-slot **CHILI** up to four unconditional commands, two conditional commands, or a combination of both can be processed.

3.2.2 Examples of Slot Computation and Assignment

The following example shows the computation of four unconditional commands within one clock cycle.

Example 1: Slot Computation with 4 Unconditional Commands

```
{
  R12 = R14;                          // slot 0
  R4  = 45;                           // slot 1
  R13 = R34 + R25;                    // slot 2
  R4  =- R5;                          // slot 3
}
```

The next example shows the computation of three unconditional commands within one clock cycle. There is no command assigned to Slot # 3 which therefore performs a NOP operation.

Example 2: Slot Computation with 3 Unconditional Commands

```
{
  R12 = R14;                          // slot 0
  R4  = 45;                           // slot 1
  R13 = R34 + R25;                    // slot 2
}
```

The following example shows a conditional execution assigned to slots 0 and 1.

Example 3: Slot Assignment with Conditional Execution (1)

```
{
  if (R0 == 1) R12 = R14;              // slots 0 and 1
  R13 = R34 + R25;                     // slot 2
  R4  =- R5;                           // slot 3
}
```

The next example shows a conditional execution assigned to slots 2 and 3.

Example 4: Slot Assignment with Conditional Execution (2)

```
{
  R13 = R34 + R25;                     // slot 2
  R4  =- R5;                           // slot 3
  if (R0 == 1) R12 = R14;              // slots 0 and 1
}
```

The next example shows two conditional executions assigned to slots 0 and 1, and 2 and 3, respectively.

Example 5: Slot Assignment with Conditional Execution (3)

```
{
  if (R0 == 1) R12 = R14;              // slots 0 and 1
  if (R0 == 0) R13 = R34 + R25;        // slots 2 and 3
}
```

The following example shows two conditional executions assigned to slots 0 and 1, and 2 and 3, respectively. If both conditions are true the execution paths are controversial. The commands of the higher slot are processed (condition 2).

Example 6: Slot Assignment with Conditional Execution (4)

```
{
  if (R0 == 1) R12 = R14;                        // slots 0 and 1
  if (R1 == 1) R12 = R34 + R25;                  // slots 2 and 3
}
```

3.3 Structure of Assembly Language

This section describes how to write a **CHILI** assembly language program.

3.3.1 Case Rules

Instructions, directives and registers have to be lower case. Labels are case-sensitive.

3.3.2 Labels

Labels are symbols that represent addresses in both program memory and data

memory. The address given by the label is calculated by the assembler.

3.3.3 Comments

Two slashes "//" mark the beginning of a single-line comment. The end of the line is the end of the comment. A multi-line comment starts with "`/*`" and ends with "`*/`". All comments are ignored by the assembler.

3.3.4 Data Section

The data section declares and allocates data memory for later use by a **CHILI** program. Together with one or several ***code sections*** (see page 10) it is part of a **CHILI** program. A program may contain an unlimited number of data sections.

The data section begins with the `.data` directive and ends at the `.code` directive or at the end of file. Initially runtime contents can be also defined in this section.

3.3.4.1 Initialization of the Data Memory

The data section contains the declaration of address labels.

```
label;
```

Using `label` as an address will always access the same memory word and initializes it with the value `int32`.

```
label = int32;
label[uint12] = {int32, int32,…, int32}
```

The following example initializes a data memory range with the values given in braces. `identifier[uint12]` represents an address alias, where `uint12` is the offset to the base address.

Example 7: Memory Initialization

```
.data
vertical_size = 144;                    // mem[0]=144
speed = 90;                             // mem[1]=90
array_size[8] = {0,1,2,3,4,5,6,7}       // mem[2]=0
                                        // mem[3]=1
                                        // mem[4]=2
                                        // mem[5]=3
                                        // mem[6]=4
                                        // mem[7]=5
                                        // mem[8]=6
                                        // mem[9]=7
```

3.3.5 Code Section

Code sections contain assembler instructions which are described in the following. Together with a ***data section*** (see page 10) they are part of a **CHILI** program. These program sections are delimited by the `.data` and `.code` designators,

respectively. The beginning of a code section is marked with the `.code` designator and a code section ends alternatively with the `.code` designator or the end of file. A program may contain an unlimited number of code sections.

Note An address label declared in a data section can be used in any code section.

3.3.5.1 Example of CHILI Assembly Language Module

The following example shows a simple program where the constant LOOP_MAX is defined by a preprocessor directive (see ***Preprocessor*** on page 90 for more information).

Example 8: Program Module

```
#define LOOP_MAX 63

.data
loop_init = 0;

.code
  {
  r0 = port[loop_init];               // initialize r0 from memory
}
loop:
{
  if (r0 < LOOP_MAX) jump(loop);      // loop as long as r0 < 63
  r0 = r0 + 1;
}
{
  halt;                               // halt processor
}
```

The following table shows values to express an effective memory address.

Table 4: Effective Address Values

Value	Description
`int32`	The immediate value `int32` is used as the address of this transfer.
`Rx`	The value in `Rx` is used as the address of this transfer.
`Rx + int32`	The immediate value `int32` is added to the value in `Rx` before the data transfer takes place. The result is used as the memory address.
`Rx + Ry`	The value in `Rx` is added to the value in `Ry` before the data transfer takes place. The result is used as the memory address.

3.4 DMA Programming

The main task of the **CHILI** data memory subsystem is to provide non-blocking memory access through a DMA controller. Typically, a predefined block of memory is transferred from internal to external memory or vice versa. The necessary information for such a transfer is stored in a descriptor register. When the DMS initiates a transfer request for a descriptor, the request is assigned an identifier for

further tracking and is routed to a command queue until the transfer is finished.

The DMA controller provides the following registers:

- DMA Descriptor Registers

  Contain the necessary information for transferring predefined blocks of memory between external and internal memory or vice versa. The blocks describe two-dimensional areas in memory. 32 descriptors are supported.

- DMA Command Queue Registers

  Hold the parameters of a DMA transfer request. 64 command queue entries are supported.

- DMA Status Registers

  A set of 2-bit registers each of which corresponds to a command queue entry.

The following figure illustrates the DMA controller in detail.

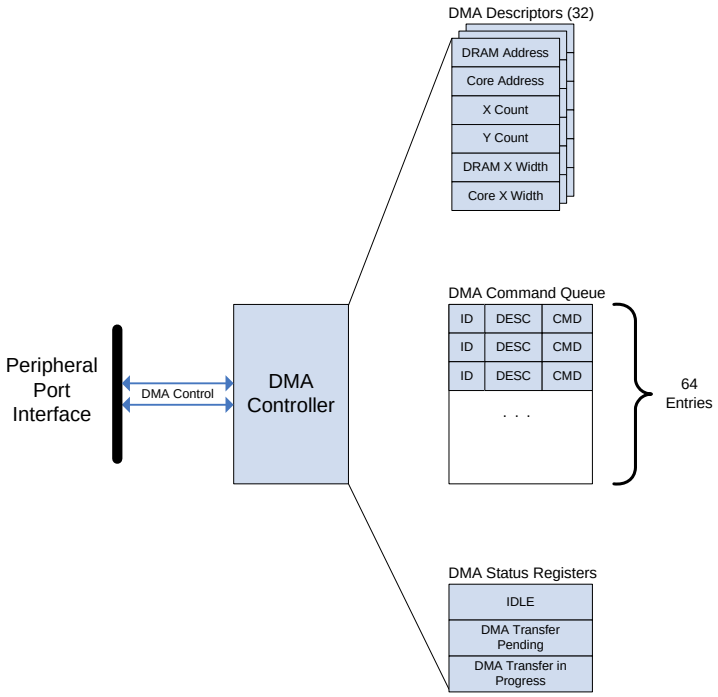

Figure 4: DMA Controller Details

DMA is performed through the **CHILI** peripheral port interface. See ***Peripheral Port Read Instructions*** on page 39 and ***Peripheral Port Write Instructions*** on page 41 for more details on how to access the **CHILI** peripheral port. The peripheral port interface uses three address port bits to indicate a specific command. The values and their description are listed in the following table.

Table 5: DMA Commands

Port Bits A[10..8]	Description
000	IDLE
001	Initiates a new DMA transfer request.
010	Non-blocking status query; CPU continues to run.
011	Blocking status query; halts the CPU.

Port Bits A[10..8]	Description
100	READ access to DMA transfer control registers (descriptors).
101	WRITE access to define DMA transfer control registers (descriptors).
110	IDLE
111	IDLE

Note DMA commands are automatically non-blocking unless the DMA command queue is full, which means exceeding the maximum of 64 entries.

Apart from defining the DMA command parameters the **CHILI** peripheral port's address and data bits hold different values, depending on the specified DMA command. See ***DMA Commands*** on page 15 for more details.

3.4.1 DMA Descriptor Registers

Descriptor registers hold the information necessary for the transfer of 2D blocks of memory from external to internal memory or vice versa. The **CHILI** DMS supports 32 of these descriptors.

A DMA descriptor and its parameters are illustrated in the following figure.

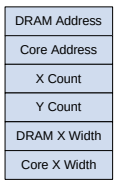

Figure 5: DMA Descriptor

The structure of a DMA descriptor is described in the following table.

Table 6: Descriptor Structure

Offset[1]	Parameter	Description
000_B	DRAM Address	Transfer start address in external address space.
001_B	Core Address	Transfer start address in core address space.
010_B	X Count	Number of bytes per line to be transferred.
011_B	Y Count	Number of lines to be transferred.
100_B	DRAM X Width	Address span to next line after completing a line transfer.
101_B	Core X Width	Address span to next line after completing a line transfer.

3.4.1.1 Descriptor Details

This section describes the transfer of a 2D memory block from external memory (DRAM) to internal (core) memory and how this translates into the proper values of a descriptor.

The following figure shows the external memory block that is to be transferred to

[1] Address port bits A[2..0]

the core memory.

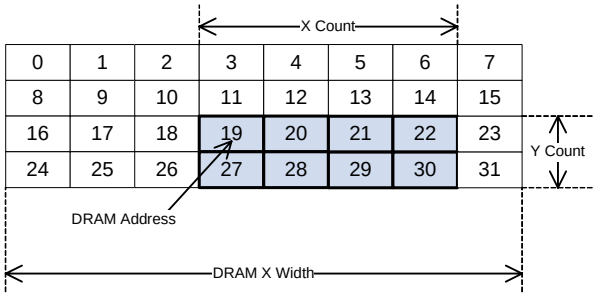

Figure 6: External Memory Block

The numbered rectangles in the previous figure represent byte addresses; the intention is to transfer the bytes from addresses 19-22 and 27-30 to an address space in internal memory, namely addresses 7-10 and 13-16, as shown in the following figure.

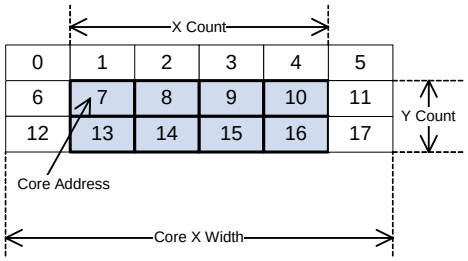

Figure 7: Internal Memory Block

The appropriate descriptor values are given in the following table.

Table 7: Descriptor Values

Offset[2]	**Parameter**	**Value**
000	DRAM Address	19
001	Core Address	7
010	X Count	4
011	Y Count	2
100	DRAM X Width	8
101	Core X Width	6

3.4.2 DMA Command Queue Registers

Command queue registers hold the parameters for initiated DMA transfer requests. The **CHILI** DMS supports 64 command queue entries.

[2] Address port bits A[2..0]

A DMA command queue and its parameters are illustrated in the following figure.

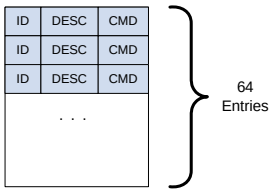

Figure 8: DMA Command Queue

The structure of a DMA command queue is described in the following table.

Table 8: Command Queue Parameters

Parameter	Description
ID	The identifier a newly initiated DMA transfer request receives; it can be used for status queries until the transfer is finished. The ID is assigned automatically by port data bits DOUT[5..0] in transfer requests and DIN[5..0] in status queries.
DESC	The address of a descriptor to be transferred given by port bits A[7..3] for descriptor READ or WRITE accesses or DIN[7..3] for transfer requests.
CMD	Indicates an incoming or outgoing transfer request given by port data bit DIN[0] as follows: ▪ LOW (set to 0): indicates incoming transfer ▪ HIGH (set to 1): indicates outgoing transfer

3.4.3 DMA Status Registers

DMA Status registers provide information about DMA transfers and return information related to each DMA command queue entry. The status values and their description are listed in the following table.

Table 9: Status Register Values

Port Bits DOUT[1..0]	Description
00	IDLE
01	DMA transfer pending.
10	DMA transfer in progress.
11	Invalid, not driven.

3.4.4 DMA Commands

This section describes all available DMA commands and the data and address bits of the **CHILI** peripheral port that hold the appropriate values. Each command description is followed by a brief example.

Note Only the **CHILI** peripheral port slots 0 (WRITE) and 1 (READ) can be used for DMA commands.

3.4.4.1 DMA Transfer Request

Port Bits	Description
A[10..8] = "001"	Initiates the command ("001") for a new transfer request.
A[31..11,7..0] = "x..x"	Don't cares.
DIN[7..3]	Input data port used to distinguish the addresses of available descriptors in an indexed list from 0-31. Up to 32 descriptors are supported.
DIN[0]	Indicates an incoming or outgoing transfer request as follows: ▪ LOW (set to 0): indicates incoming transfer ▪ HIGH (set to 1): indicates outgoing transfer
DIN[31..8, 2..1] = "x..x"	Don't cares.
DOUT[5..0]	The identifier internal logic assigns to a descriptor when it is entered in the command queue.
DOUT[31..6] = "x..x"	Don't cares.

Function

A DMA transfer request is initiated for a descriptor. This WRITE command triggers the assignment of an identifier to the descriptor in the same cycle through a parallel slot. Identifiers are used to track the status of the transfer request during further program execution.

The following example shows how to initiate a DMA transfer through the **CHILI *peripheral port interface*** (see page 41).

Example 9: DMA Transfer

```
{
pport32[001] = 5;
/* this command initiates a new DMA transfer request for a target descriptor
 specified by the target location */


pport32[ID] = rx;
/* internal logic assigns ID and the transfer is entered in the command
 queue in the same cycle */
}
```

3.4.4.2 DMA Non-Blocking Status Query

Port Bits	Description
A[10..8] = "010"	Initiates the command ("010") for a non-blocking status query once a transfer request has been initiated.
A[31..11, 7..0] = "x..x"	Don't cares.
DIN[5..0]	Input data port used to distinguish the transfers entered in the command queue. Up to 64 command queue entries are supported.
DIN[31..6] = "x..x"	Don't cares.
DOUT[1..0]	Returns the transfer status as follows: ▪ "00": IDLE ▪ "01": Transfer pending ▪ "10": Transfer in progress ▪ "11": Non-valid, not driven
DOUT[31..2] = "x..x"	Don't cares.

Function

Non-blocking status queries do not interfere with the CPU program execution.

The following example shows a non-blocking status query through the **CHILI** peripheral port interface.

Example 10: Non-Blocking DMA Status Query

```
{
pport32[010] = ID;
/* program sends non-blocking status query for specified transfer ID
 distinguished by data input port bits DIN[5..0] */


pport32[status] = ry;
/* returns status of asserted command queue ID (idle/pending/in progress)
 distinguished by data output port bits DOUT[1..0] */
}
```

3.4.4.3 DMA Blocking Status Query

Port Bits	Description
A[10..8] = "011"	Initiates the command ("011") for a blocking status query once a transfer request has been initiated.
A[31..11, 7..0] = "x..x"	Don't cares.
DIN[5..0]	Input data port used to distinguish the transfers entered in the command queue. Up to 64 command queue entries are supported.
DIN[31..6]	Don't cares.
DOUT[1..0]	Returns the transfer status as follows: ▪ "00": IDLE ▪ "01": Transfer pending ▪ "10": Transfer in progress ▪ "11": Non-valid, not driven
DOUT[31..2] = "x..x"	Don't cares.

Function

Blocking status queries halt the CPU program execution. This may be necessary when the data required for program execution is not available and the CPU would otherwise go in a loop.

The following example shows a blocking status query through the **CHILI** peripheral port interface.

Example 11: Non-Blocking DMA Status Query

```
{
pport32[011] = ID;
/* program sends blocking status query for specified transfer ID
 distinguished by data input port bits DIN[5..0] */
}
{
pport32[status] = 10;
/* returns status "in progress" of asserted command queue ID distinguished
 by data output port bits DOUT[1..0] */
}
```

3.4.4.3.1 Descriptor READ Access

Port Bits	Description
A[10..8] = "100"	Initiates the command ("100") for a descriptor READ access.
A[7..3]	Address port bits used to address a target descriptor from an indexed list between 0-31. Up to 32 descriptors are available.
A[2..0]	Address port bits asserted to distinguish different register entries within a target descriptor. See ***DMA Descriptors*** on page 13 for more details.
A[31..11] = "x..x"	Don't cares.
DOUT[31..0]	The address values of the target descriptor.

Function

Returns the parameters of DMA transfers that have not been initiated.

The following example illustrates a descriptor READ command.

Example 12: Descriptor READ Command

```
{
pport32[100] = desc;
// READ command to a specified port address A[7..3]
}

{
// starting sequence of reading descriptor parameters
pport32[desc + dram_addr] = r0;
}
{
pport32[desc + core_addr] = r1;
}
{
pport32[desc + x_count] = r2;
}
{
pport32[desc + y_count] = r3;
}
{
pport32[desc + dram_x_width] = r4;
}
{
pport32[desc + core_x_width] = r5;
}
```

3.4.4.4 Descriptor WRITE Access

Port Bits	Description
A[10..8] = "101"	Initiates the command ("101") for a descriptor WRITE access.
A[7..3]	Address port bits used to address a target descriptor from an indexed list between 0-31. Up to 32 descriptors are available.
A[2..0]	Address port bits asserted to distinguish different register entries within a target descriptor. See ***DMA Descriptors*** on page 13 for more details.
A[31..11] = "x..x"	Don't cares.
DIN[31..0]	The address values of the target descriptor.

Function

Specifies the parameters for DMA transfers.

The following example illustrates a descriptor WRITE command.

Example 13: Descriptor WRITE Command

```
{
pport32[101] = desc;
// WRITE command to a specified port address using bits A[7..3]
}
{
/* starting sequence of writing descriptor parameters for future DMA
 transfers */
pport32[desc + dram_addr] = 0xAAAA;
}
{
pport32[desc + core_addr] = 0x1111;
}
{
pport32[desc + x_count] = 4;
}
{
pport32[desc + y_count] = 2;
}
{
pport32[desc + dram_x_width] = 0x100;
}
{
pport32[desc + core_x_width] = 0x10;
}
```

CHAPTER 4

Instruction Set

In This Chapter

4.1 Instruction Overview

This section describes the **CHILI** instructions that are supported by the **CHILI** assembler. Each instruction is listed with its syntax and latency, functional description and a practical example of all available commands. The instructions are grouped as follows:

4.1.1 Transfer Instructions

Syntax	Latency
`Rn = int32;`	1
`Rn = Rx;`	1

Function

These instructions copy a 32-bit immediate value or the value of the source register Rx to the destination register Rn.

4.1.1.1 MOV_IMM

Syntax

```
Rn = int32;
```

Example 14: MOV_IMM Command

Current state
```
r1 = 0xAAAAAAAA
```

Command
```
r1 = 0x45678912;
```

New state
```
r1 = 0x45678912
```

4.1.1.2 MOV_REG

Syntax
```
Rn = Rx;
```

Example 15: MOV_REG Command

Current state
```
r1 = 0x12345678
r2 = 0xAAAAAAAA
```

Command
```
r2 = r1;
```

New state
```
r2 = 0x12345678
```

4.1.2 Core Memory Read Instructions

Syntax	Latency
`Rn = port8/16/32[int32];`	8
`Rn = port8/16/32[Rx];`	8
`Rn = port8/16/32[Rx + int32];`	8
`Rn = port8/16/32[Rx + Ry];`	8
`Rn = sport8/16[int32];`	8
`Rn = sport8/16[Rx];`	8
`Rn = sport8/16[Rx + int32];`	8
`Rn = sport8/16[Rx + Ry];`	8

Function

These instructions transfer the value from the core data memory to the destination register `Rn` and read from address-mapped port locations.
The instructions `port8` and `port16` transfer the lower 8 and 16 bits respectively to `Rn` and set the upper bits to 0.
Port-signed instructions (`sport`) expand bit 7 and bit 16 of the loaded values as sign for bits 31 to 8 and 31 to 16 respectively.

4.1.2.1 LOAD_IMM

Syntax

```
Rn = port32[int32];
```

Example 16: LOAD_IMM Command

Current state

```
(0x80000000) = 0xAB123456
r1 = 0xAAAAAAAA
```

Command

```
r1 = port32[0x80000000];
```

New state

```
r1 = 0xAB123456
```

4.1.2.2 LOAD_REG

Syntax

```
Rn = port32[Rx];
```

Example 17: LOAD_REG Command

Current state

```
(0x80000000) = 0xAB123456
r1 = 0xAAAAAAAA
r2 = 0x80000000
```

Command

```
r1 = port32[r2];
```

New state

```
r1 = 0xAB123456
```

4.1.2.3 LOAD_REG_IMM

Syntax

```
Rn = port32[Rx + int32];
```

Example 18: LOAD_REG_IMM Command

Current state

```
(0x80000000) = 0xBBBBBBBB
(0x80000004) = 0xAB123456
r1 = 0xAAAAAAAA
r2 = 0x80000000
```

Command

```
r1 = port32[r2 + 0x7];
```

New state

```
r1 = 0xAB123456
```

4.1.2.4 LOAD_REG_REG

Syntax

```
Rn = port32[Rx + Ry];
```

Example 19: LOAD_REG_REG Command

Current state

```
(0x80000000) = 0xBBBBBBBB
(0x80000004) = 0xAB123456
r1 = 0xAAAAAAAA
r2 = 0x80000000
r3 = 0x6
```

Command

```
r1 = port32[r2 + r3];
```

New state

```
r1 = 0xAB123456
```

4.1.2.5 LOAD_B_IMM

Syntax

```
Rn = port8[int32];
```

Example 20: LOAD_B_IMM Command

Current state
```
(0x80000000) = 0xAB123456
r1 = 0xAAAAAAAA
```

Command
```
r1 = port8[0x80000000];
```

New state
```
r1 = 0x00000056
```

4.1.2.6 LOAD_B_REG

Syntax
```
Rn = port8[Rx];
```

Example 21: LOAD_B_REG Command

Current state
```
(0x80000000) = 0xAB123456
r1 = 0xAAAAAAAA
r2 = 0x80000000
```

Command
```
r1 = port8[r2];
```

New state
```
r1 = 0x00000056
```

4.1.2.7 LOAD_B_REG_IMM

Syntax
```
Rn = port8[Rx + int32];
```

Example 22: LOAD_B_REG_IMM Command

Current state

```
(0x80000000) = 0xBBBBBBBB
(0x80000004) = 0xAB123456
r1 = 0xAAAAAAAA
r2 = 0x80000000
```

Command

```
r1 = port8[r2 + 0x5];
```

New state

```
r1 = 0x00000034
```

4.1.2.8 LOAD_B_REG_REG

Syntax

```
Rn = port8[Rx + Ry];
```

Example 23: LOAD_B_REG_REG Command

Current state

```
(0x80000000) = 0xBBBBBBBB
(0x80000004) = 0xAB123456
r1 = 0xAAAAAAAA
r2 = 0x80000000
r3 = 0x6
```

Command

```
r1 = port8[r2 + r3];
```

New state

```
r1 = 0x00000012
```

4.1.2.9 LOAD_W_IMM

Syntax

```
Rn = port16[int32];
```

Example 24: LOAD_W_IMM Command

Current state
```
(0x80000000) = 0xAB123456
r1 = 0xAAAAAAAA
```

Command
```
r1 = port16[0x80000000];
```

New state
```
r1 = 0x00003456
```

4.1.2.10 LOAD_W_REG

Syntax
```
Rn = port16[Rx];
```

Example 25: LOAD_W_REG Command

Current state
```
(0x80000000) = 0xAB123456
r1 = 0xAAAAAAAA
r2 = 0x80000000
```

Command
```
r1 = port16[r2];
```

New state
```
r1 = 0x00003456
```

4.1.2.11 LOAD_W_REG_IMM

Syntax
```
Rn = port16[Rx + int32];
```

Example 26: LOAD_W_REG_IMM Command

Current state

```
(0x80000000) = 0xBBBBBBBB
(0x80000004) = 0xAB123456
r1 = 0xAAAAAAAA
r2 = 0x80000000
```

Command

```
r1 = port16[r2 + 0x5];
```

New state

```
r1 = 0x00003456
```

4.1.2.12 LOAD_W_REG_REG

Syntax

```
Rn = port16[Rx + Ry];
```

Example 27: LOAD_W_REG_REG Command

Current state

```
(0x80000000) = 0xBBBBBBBB
(0x80000004) = 0xAB123456
r1 = 0xAAAAAAAA
r2 = 0x80000000
r3 = 0x00000006
```

Command

```
r1 = port16[r2 + r3];
```

New state

```
r1 = 0x0000AB12
```

4.1.2.13 SLOAD_B_IMM

Syntax

```
Rn = sport8[int32];
```

Example 28: SLOAD_B_IMM Command

Current state

```
(0x80000000) = 0xAB123456 // value mapped at port address
r1 = 0xAAAAAAAA
```

Command

```
r1 = sport8[0x80000000];
```

New state

```
r1 = 0x00000056
```

4.1.2.14 SLOAD_B_REG

Syntax

```
Rn = sport8[Rx];
```

Example 29: SLOAD_B_REG Command

Current state

```
(0x80000000) = 0xAB123456 // value mapped at port address
r1 = 0xAAAAAAAA
r2 = 0x80000000
```

Command

```
r1 = sport8[r2];
```

New state

```
r1 = 0x00000056
```

4.1.2.15 SLOAD_B_REG_IMM

Syntax

```
Rn = sport8[Rx + int32];
```

Example 30: SLOAD_B_REG_IMM Command

Current state

```
(0x80000000) = 0xBBBBBBBB // value mapped at port address
(0x80000004) = 0xAB123456 // value mapped at port address
r1 = 0xAAAAAAAA
r2 = 0x80000000
```

Command

```
r1 = sport8[r2 + 0x7];
```

New state

```
r1 = 0xFFFFFFAB
```

4.1.2.16 SLOAD_B_REG_REG

Syntax

```
Rn = sport8[Rx + Ry];
```

Example 31: SLOAD_B_REG_REG Command

Current state

```
(0x80000000) = 0xBBBBBBBB // value mapped at port address
(0x80000004) = 0xAB123456 // value mapped at port address
r1 = 0xAAAAAAAA
r2 = 0x80000000
r3 = 0x6
```

Command

```
r1 = sport8[r2 + r3];
```

New state

```
r1 = 0x00000012
```

4.1.2.17 SLOAD_W_IMM

Syntax

```
Rn = sport16[int32];
```

Example 32: SLOAD_W_IMM Command

Current state

```
(0x80000000) = 0xAB1234 // value mapped at port address
r1 = 0xAAAAAAAA
```

Command

```
r1 = sport16[0x80000000];
```

New state

```
r1 = 0x00001234
```

4.1.2.18 SLOAD_W_REG

Syntax

```
Rn = sport16[Rx];
```

Example 33: SLOAD_W_REG Command

Current state

```
(0x80000000) = 0xAB1234 // value mapped at port address
r1 = 0xAAAAAAAA
r2 = 0x80000000
```

Command

```
r1 = sport16[r2];
```

New state

```
r1 = 0xFFFFABCD
```

4.1.2.19 SLOAD_W_REG_IMM

Syntax

```
Rn = sport16[Rx + int32];
```

Example 34: SLOAD_W_REG_IMM Command

Current state

```
(0x80000000) = 0xBBBBBBBB // value mapped at port address
(0x80000004) = 0xAB123456 // value mapped at port address
r1 = 0xAAAAAAAA
r2 = 0x80000000
```

Command

```
r1 = sport16[r2 + 0x5];
```

New state

```
r1 = 0x00003456
```

4.1.2.20 SLOAD_W_REG_REG

Syntax

```
Rn = sport16[Rx + Ry];
```

Example 35: SLOAD_W_REG_REG Command

Current state

```
(0x80000000) = 0xBBBBBBBB // value mapped at port address
(0x80000004) = 0xCD123456 // value mapped at port address
r1 = 0xAAAAAAAA
r2 = 0x80000000
r3 = 0x00000006
```

Command

```
r1 = sport16[r2 + r3];
```

New state

```
r1 = 0xFFFFCD12
```

4.1.3 Core Memory Write Instructions

Syntax	Latency
port8/16/32[int32] = Rn;	-
port8/16/32[Rx] = Rn;	-
port8/16/32[Rx + int32] = Rn;	-
port8/16/32[Rx + Ry] = Rn;	-
port8/16/32[int32] = int12;	-
port8/16/32[Rx] = int12;	-
port8/16/32[Rx + int32] = int12;	-

Function

These instructions store the value of the register Rn or a 12-bit immediate value in the core data memory and at address-mapped port locations.

4.1.3.1 STORE_IMM

Syntax

```
port32[int32] = Rn;
```

Example 36: STORE_IMM Command

Current state

```
(0x80000000) = 0xBBBBBBBB
r1 = 0xAB456789
```

Command

```
port32[0x80000000] = r1;
```

New state

```
(0x80000000) = 0xAB456789
```

4.1.3.2 STORE_REG

Syntax

```
port32[Rx] = Rn;
```

Example 37: STORE_REG Command

Current state

```
(0x80000000) = 0xBBBBBBBB
r1 = 0xAB456789
r2 = 0x80000000
```

Command

```
port32[r2] = r1;
```

New state

```
(0x80000000) = 0xAB456789
```

4.1.3.3 STORE_REG_IMM

Syntax

```
port32[Rx + int32] = Rn;
```

Example 38: STORE_REG_IMM Command

Current state

```
(0x80000000) = 0xBBBBBBBB
r1 = 0xAB456789
r2 = 0x80000000
```

Command

```
port32[r2 + 0x2] = r1;
```

New state

```
(0x80000000) = 0xAB456789
```

4.1.3.4 STORE_REG_REG

Syntax

```
port32[Rx + Ry] = Rn;
```

Example 39: STORE_REG_REG Command

Current state

```
(0x80000000) = 0xBBBBBBBB
r1 = 0xAB456789
r2 = 0x80000000
r3 = 0x3
```

Command

```
port32[r2 + r3] = r1;
```

New state

```
(0x80000000) = 0xAB456789
```

4.1.3.5 STORE_B_IMM

Syntax

```
port8[int32] = Rn;
```

Example 40: STORE_B_IMM Command

Current state
```
(0x80000000) = 0xBBBBBBBB
r1 = 0xAB456789
```

Command
```
port8[0x80000000] = r1;
```

New state
```
(0x80000000) = 0xBBBBBB89
```

4.1.3.6 STORE_B_REG

Syntax
```
port8[Rx] = Rn;
```

Example 41: STORE_B_REG Command

Current state
```
(0x80000000) = 0xBBBBBBBB
r1 = 0xAB456789
r2 = 0x80000001
```

Command
```
port8[r2] = r1;
```

New state
```
(0x80000000) = 0xBBBB89BB
```

4.1.3.7 STORE_B_REG_IMM

Syntax
```
port8[Rx + int32] = Rn;
```

Example 42: STORE_B_REG_IMM Command

Current state
```
(0x80000000) = 0xBBBBBBBB
r1 = 0xAB456789
r2 = 0x80000000
```

Command
```
port8[r2 + 0x2] = r1;
```

New state
```
(0x80000000) = 0xBB89BBBB
```

4.1.3.8 STORE_W_IMM

Syntax

```
port16[int32] = Rn;
```

Example 43: STORE_W_IMM Command

Current state

```
(0x80000000) = 0xBBBBBBBB
r1 = 0xAB456789
```

Command

```
port16[0x80000000] = r1;
```

New state

```
(0x80000000) = 0xBBBB6789
```

4.1.3.9 STORE_W_REG

Syntax

```
port16[Rx] = Rn;
```

Example 44: STORE_W_REG Command

Current state

```
(0x80000000) = 0xBBBBBBBB
r1 = 0xAB456789
r2 = 0x8000000
```

Command

```
port16[r2] = r1;
```

New state

```
(0x80000000) = 0xBBBB6789
```

4.1.3.10 STORE_SHORT_B_REG

Syntax

```
port8[Rx] = int12;
```

Example 45: STORE_SHORT_B_REG Command

Current state
```
(0x80000000) = 0xBBBBBBBB
r2 = 0x80000001
```

Command
```
port8[r2] = 0x12345678;
```

New state
```
(0x80000000) = 0xBBBB78BB
```

4.1.3.11 STORE_SHORT_B_IMM

Syntax
```
port8[int32] = int12;
```

Example 46: STORE_SHORT_B_IMM Command

Current state
```
(0x80000000) = 0xBBBBBBBB
```

Command
```
port8[0x80000000] = 0x12345678;
```

New state
```
(0x80000000) = 0xBBBBBB78
```

4.1.3.12 STORE_SHORT_B_REG_IMM

Syntax
```
port8[Rx + int32] = int12;
```

Example 47: STORE_SHORT_B_REG_IMM Command

Current state
```
(0x80000000) = 0xBBBBBBBB
r2 = 0x80000000
```

Command
```
port8[r2 + 0x2] = 0x00000AFE;
```

New state
```
(0x80000000) = 0xBBFEBBBB
```

4.1.3.13 STORE_SHORT_W_IMM

Syntax

```
port16[int32] = int12;
```

Example 48: STORE_SHORT_W_IMM Command

Current state

```
(0x80000000) = 0xBBBBBBBB
```

Command

```
port16[0x80000000] = 0x12345678;
```

New state

```
(0x80000000) = 0xBBBB0678
```

4.1.3.14 STORE_SHORT_W_REG

Syntax

```
port16[Rx] = int12;
```

Example 49: STORE_SHORT_W_REG Command

Current state

```
(0x80000000) = 0xBBBBBBBB
r2 = 0x80000001
```

Command

```
Port8[r2] = 0x12345678;
```

New state

```
(0x80000000) = 0xBBBB0678
```

4.1.3.15 STORE_SHORT_W_REG_IMM

Syntax

```
port16[Rx + int32] = int12;
```

Example 50: STORE_SHORT_W_REG_IMM Command

Current state

```
(0x80000000) = 0xBBBBBBBB
r2 = 0x80000000
```

Command

```
port16[r2 + 0x2] = 0x54321AFE;
```

New state

```
(0x80000000) = 0x0AFEBBBB
```

4.1.4 Peripheral Port Read Instructions

Syntax	Latency
`Rn = pport32[int32];`	-
`Rn = pport32[Rx]:`	-
`Rn = pport32[Rx + int32];`	-
`Rn = pport32[Rx + Ry];`	-

Function

These instructions transfer the value from the memory-mapped peripheral port of the **CHILI** to the destination register Rn.

4.1.4.1 PLOAD_IMM

Syntax

```
Rn = pport32[int32];
```

Example 51: PLOAD_IMM Command

Current state

```
(0x80000000) = 0xAB123456
r1 = 0xAAAAAAAA
```

Command

```
r1 = pport32[0x80000000];
```

New state

```
r1 = 0xAB123456
```

4.1.4.2 PLOAD_REG

Syntax

```
Rn = pport32[Rx];
```

Example 52: PLOAD_REG Command

Current state

```
(0x80000000) = 0xAB123456
r1 = 0xAAAAAAAA
r2 = 0x80000000
```

Command

```
r1 = pport32[r2];
```

New state

```
r1 = 0xAB123456
```

4.1.4.3 PLOAD_REG_IMM

Syntax

```
Rn = pport32[Rx + int32];
```

Example 53: PLOAD_REG_IMM Command

Current state

```
(0x80000000) = 0xBBBBBBBB
(0x80000004) = 0xAB123456
r1 = 0xAAAAAAAA
r2 = 0x80000000
```

Command

```
r1 = pport32[r2 + 0x7];
```

New state

```
r1 = 0xAB123456
```

4.1.4.4 PLOAD_REG_REG

Syntax

```
Rn = pport32[Rx + Ry];
```

Example 54: PLOAD_REG_REG Command

Current state

```
(0x80000000) = 0xBBBBBBBB
(0x80000004) = 0xAB123456
r1 = 0xAAAAAAAA
r2 = 0x80000000
r3 = 0x6
```

Command

```
r1 = pport32[r2 + r3];
```

New state

```
r1 = 0xAB123456
```

4.1.5 Peripheral Port Write Instructions

Syntax	Latency
`pport32[int32] = Rn;`	-
`pport32[Rx] = Rn;`	-
`pport32[Rx + int32] = Rn;`	-
`pport32[Rx + Ry] = Rn;`	-
`pport32[int32] = int12;`	-
`pport32[Rx] = int12;`	-
`pport32[Rx + int32] = int12;`	-

Function

These instructions store the value of the register Rn or a 12-bit immediate value in the memory-mapped peripheral port of the **CHILI**.

4.1.5.1 PSTORE_IMM

Syntax

```
pport32[int32] = Rn;
```

Example 55: PSTORE_IMM Command

Current state
```
(0x80000000) = 0xBBBBBBBB
r1 = 0xAB456789
```

Command
```
pport32[0x80000000] = r1;
```

New state
```
(0x80000000) = 0xAB456789
```

4.1.5.2 PSTORE_REG

Syntax
```
pport32[Rx] = Rn;
```

Example 56: PSTORE_REG Command

Current state
```
(0x80000000) = 0xBBBBBBBB
r1 = 0xAB456789
r2 = 0x80000000
```

Command
```
pport32[r2] = r1;
```

New state
```
(0x80000000) = 0xAB456789
```

4.1.5.3 PSTORE_REG_IMM

Syntax
```
pport32[Rx + int32] = Rn;
```

Example 57: PSTORE_REG_IMM Command

Current state
```
(0x80000000) = 0xBBBBBBBB
r1 = 0xAB456789
r2 = 0x80000000
```

Command
```
pport32[r2 + 0x2] = r1;
```

New state
```
(0x80000000) = 0xAB456789
```

4.1.6 Arithmetic Instructions

4.1.6.1 Add

Syntax	Latency
`Rn = Rx + int32;`	1
`Rn = Rx + Ry;`	1

Function

Adds the values in the registers Rx and Ry or the value in Rx and an immediate value and places the result in register Rn.

4.1.6.1.1 ADD_REG

Syntax

```
Rn = Rx + Ry;
```

Example 58: ADD_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00000001
r3 = 0x00000003
```

Command

```
r2 = r1 + r3;
```

New state

```
r2 = 0x00000004
```

4.1.6.1.2 ADD_IMM

Syntax

```
Rn = Rx + int32;
```

Example 59: ADD_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00000001
```

Command

```
r2 = r1 + 0x00000002;
```

New state

```
r2 = 0x00000003
```

4.1.6.2 Subtract

Syntax	Latency
`Rn = Rx - int32;`	1
`Rn = int32 - Rx;`	1
`Rn = Rx - Ry;`	1

Function

Subtracts an immediate value or the value in register `Ry` from the value in register `Rx` and places the result in the destination register.

4.1.6.2.1 SUB_IMM

Syntax

```
Rn = Rx - int32;
```

Example 60: SUB_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00000005
```

Command

```
r2 = r1 - 0x00000002;
```

New state

```
r2 = 0x00000003
```

4.1.6.2.2 IMM_SUB

Syntax

```
Rn = int32 - Rx;
```

Example 61: IMM_SUB Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00003456
```

Command

```
r2 = 0x00000004 - r1;
```

New state

```
r2 = 0xFFFFCBAE
```

4.1.6.2.3 SUB_REG

Syntax

```
Rn = Rx - Ry;
```

Example 62: SUB_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00000000
r3 = 0x00000001
```

Command

```
r2 = r1 - r3;
```

New state

```
r2 = 0xFFFFFFFF
```

4.1.6.3 Sign Extension

Syntax	Latency
`Rn = sex(Rx, int32);`	1
`Rn = sex(Rx, Ry);`	1

Function

Sign-extends register `Rx`. The sign bit (immediate value `int32` or value in `Ry`) is copied to the upper bits in `Rx`.

4.1.6.3.1 SEX_IMM

Syntax

```
Rn = sex(Rx, int32);
```

Example 63: SEX_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00003456
```

Command

```
r2 = sex(r1, 0x00000004);
```

New state

```
r2 = 0xFFFFFFF6
```

4.1.6.3.2 SEX_REG

Syntax

```
Rn = sex(Rx, Ry);
```

Example 64: SEX_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x04323400
r3 = 0x0000000D
```

Command

```
r2 = sex(r1, r3);
```

New state

```
r2 = 0xFFFFF400
```

4.1.6.4 Absolute Value

Syntax	Latency
`Rn = abs(int32);`	1
`Rn = abs(Rx);`	1

Function

Determines the absolute value of the operand and places the result in register Rn.

4.1.6.4.1 ABS_IMM

Syntax

```
Rn = abs(int32);
```

Example 65: ABS_IMM Command

Current state

```
r1 = 0xBBBBBBBB
```

Command

```
r1 = abs(0x00001234);
```

New state

```
r1 = 0x00001234
```

4.1.6.4.2 ABS_REG

Syntax

```
Rn = abs(Rx);
```

Example 66: ABS_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xFFFFFFEC
```

Command

```
r1 = abs(r2);
```

New state

```
r1 = 0x00000014
```

4.1.6.5 Negation

Syntax	Latency
`Rn = -Rx;`	1

Function

Negates the value of the operand by two's complement and places the result is placed in register Rn.

4.1.6.5.1 NEG_REG

Syntax

```
Rn = -Rx;
```

Example 67: NEG_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xFFFFFFEC
```

Command

```
r1 = -r2;
```

New state

```
r1 = 0x00000014
```

4.1.6.6 Multiplication

Syntax	Latency
`Rn = Rx * int32;`	3
`Rn = Rx * Ry;`	3

Function

Multiplies an immediate value or the value of register Ry with the value in register Rx and places the result in register Rn.

4.1.6.6.1 MUL_IMM

Syntax

```
Rn = Rx * int32;
```

Example 68: MUL_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00003456
```

Command

```
r2 = r1 * 0x11223344;
```

New state

```
r2 = 0x02C5A620
```

4.1.6.6.2 MUL_REG

Syntax

```
Rn = Rx * Ry;
```

Example 69: MUL_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0xFF323400
r3 = 0x00453467
```

Command

```
r2 = r1 * r3;
```

New state

```
r2 = 0x04947C00
```

4.1.6.7 Clip

Syntax	Latency
`Rn = clip(int16, int16);`	1
`Rn = clip(Rx, Ry);`	1

Function

Clips the value of register Rn into the boundaries given either by two immediate values or the values in registers Rx and Ry and places the result in register Rn.

4.1.6.7.1 CLIP_IMM

Syntax

```
Rn = clip(int16, int16);
```

Example 70: CLIP_IMM Command[3]

Current state

```
r2 = 0x3E0045C6
```

Command

```
r2 = clip(0x1234, 0x0455);
```

// clips r2 in the boundaries of the clip region

New state

```
r2 = 0x00001234
```

4.1.6.7.2 CLIP_REG

Syntax

```
Rn = clip(Rx, Ry);
```

Example 71: CLIP_REG Command[4]

Current state

```
r2 = 0x0C0045C6
r1 = 0x0E000003
r3 = 0x0F000008
```

Command

```
r2 = clip(r1, r3);
```

// clips r2 in the boundaries of the clip region

New state

```
r2 = 0x0E000003
```

4.1.6.8 Absolute Difference

Syntax	Latency
`Rn = absdiff(Rx, int32):`	1
`Rn = absdiff(Rx, Ry);`	1

Function

Subtracts the immediate value `int32` or the value of register `Ry` from register `Rx` and sets the result as absolute value in register `Rn`.

[3] Border position clip(floor, ceil)
[4] Border position clip(floor, ceil)

4.1.6.8.1 ABSDIFF_IMM

Syntax

```
Rn = absdiff(Rx, int32);
```

Example 72: ABSDIFF_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00003465
```

Command

```
r2 = absdiff(r1, 0x00000004);
```

New state

```
r2 = 0x00003452
```

4.1.6.8.2 ABSDIFF_REG

Syntax

```
Rn = absdiff(Rx, Ry);
```

Example 73: ABSDIFF_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x04323400
r3 = 0x0000000D
```

Command

```
r2 = absdiff(r3, r1);
```

New state

```
r2 = 0x043233F3
```

4.1.6.9 Alignment and Rounding

Syntax	Latency
`Rn = alro(Rx, int32);`	1
`Rn = alro(Rx, Ry);`	1

Function

Performs a 32-bit alignment and rounding operation as given by `Rn <-(Rx+2^(int32-1))>>int32`.

4.1.6.9.1 ALRO_IMM

Syntax

```
Rn = alro(Rx, int32);
```

Example 74: ALRO_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0xAB123456
```

Command

```
r2 = alro(r1, 0x00000008);
```

New state

```
r2 = 0xFFAB1234
```

4.1.6.9.2 ALRO_REG

Syntax

```
Rn = alro(Rx, Ry);
```

Example 75: ALRO_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x11111800
r3 = 0x0000000C
```

Command

```
r2 = alro(r1, r3);
```

New state

```
r2 = 0x00011112
```

4.1.6.10 Leading Bits

Syntax	Latency
`Rn = leadbits(Rx, int32);`	1

Function

Performs the following functions on the leading bits of the input operand in a result range between 0 and 32:

- leading_zeros: returns the number of leading zeros in Rx
- leading_ones: returns the number of leading ones in Rx
- signbits: returns the number of identical leading bits in Rx
- msbpos: returns 32 minus the number of leading zeros

The following example shows how each function operates.

Example 76: Leading Bits

```
R0= 0x00000037;   // 0000 0000 0000 0000 0000 0000 0011 0111B
R1= -R0;          // 1111 1111 1111 1111 1111 1111 1100 1001B
R2= 0x0;          // 0B

R3=leadbits(R0,leading_zeros);      // R3 equals 26D
R3=leadbits(R2,leading_zeros);      // R3 equals 32D

R3=leadbits(R0,leading_ones); // R3 equals 0
R3=leadbits(R1,leading_ones); // R3 equals 26D

R3=leadbits(R0,signbits);           // R3 equals 26D
R3=leadbits(R1,signbits);           // R3 equals 26D

R3=leadbits(R0,msbpos);       // R3 equals 6D
R3=leadbits(R1,msbpos);       // R3 equals 32D
R3=leadbits(R2,msbpos);       // R3 equals 0
```

4.1.6.10.1 LEADBITS

Syntax

```
Rn = leadbits(Rx, int32);
```

Example 77: LEADBITS Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00020AA0
```

Command

```
r2 = leadbits(r1,0x0003);
```

New state

```
r2 = 0x00000012
```

4.1.7 Shift Instructions

4.1.7.1 Arithmetic Shift

Syntax	Latency
`Rn = Rx >> int32;`	1
`Rn = Rx >> Ry;`	1
`Rn = Rx << int32;`	1
`Rn = Rx << Ry;`	1

Function

Arithmetically shifts the operand in register Rx by the 32-bit value in register Ry or by the 32-bit immediate value. The shifted result is placed in register Rn.

Note If the immediate value or the value in Ry is larger than 32 only the lower 5 bits are used for the shift operation.

4.1.7.1.1 SHR_IMM

Syntax

```
Rn = Rx >> int32;
```

Example 78: SHR_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0x348560AB
```

Command

```
r1 = r2 >> 0x00000004;
```

New state

```
r1 = 0x0348560A
```

4.1.7.1.2 SHR_REG

Syntax

```
Rn = Rx >> Ry;
```

Example 79: SHR_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0x80456576
r3 = 0x00000005
```

Command

```
r1 = r2 >> r3;
```

New state

```
r1 = 0xFC022B2B
```

4.1.7.1.3 SHL_IMM

Syntax

```
Rn = Rx << int32;
```

Example 80: SHL_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0x348560AB
```

Command

```
r1 = r2 << 0x00000004;
```

New state

```
r1 = 0x48560AB0
```

4.1.7.1.4 SHL_REG

Syntax

```
Rn = Rx << Ry;
```

Example 81: SHL_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0x80456576
r3 = 0x00000005
```

Command

```
r1 = r2 << r3;
```

New state

```
r1 = 0x08ACAEC0
```

4.1.7.2 Logical Shift

Syntax	Latency
`Rn = Rx >>> int32;`	1
`Rn = Rx >>> Ry;`	1

Function

Logically shifts the operand in register Rx by the 32-bit value in register Ry or by the 32-bit immediate value. The shifted result is placed in register Rn.

4.1.7.2.1 SHRL_IMM

Syntax

```
Rn = Rx >>> int32;
```

Example 82: SHRL_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0x348560AB
```

Command

```
r1 = r2 >>> 0x00000004;
```

New state

```
r1 = 0x0348560A
```

4.1.7.2.2 SHRL_REG

Syntax

```
Rn = Rx >>> Ry;
```

Example 83: SHRL_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0x80456576
r3 = 0x00000005
```

Command

```
r1 = r2 >>> r3;
```

New state

```
r1 = 0x04022B2B
```

4.1.8 Logical Instructions

4.1.8.1 AND

Syntax	Latency
`Rn = Rx & int32;`	1
`Rn = Rx & Ry;`	1

Function

Performs a bitwise logical AND on the value in Rx and the immediate value or the values in Rx and Ry and places the result in register Rn.

4.1.8.1.1 AND_IMM

Syntax

```
Rn = Rx & int32;
```

Example 84: AND_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0x348560AB
```

Command

```
r1 = r2 & 0x0000FFFF;
```

New state

```
r1 = 0x0000560A
```

4.1.8.1.2 AND_REG

Syntax

```
Rn = Rx & Ry;
```

Example 85: AND_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0x80456576
r3 = 0xF0F0F0F0
```

Command

```
r1 = r2 & r3;
```

New state

```
r1 = 0x80406070
```

4.1.8.2 OR

Syntax	Latency
`Rn = Rx | int32;`	1
`Rn = Rx | Ry;`	1

Function

Performs a bitwise logical OR on the value in Rx and the immediate value or the values in Rx and Ry and places the result in register Rn.

4.1.8.2.1 OR_IMM

Syntax

```
Rn = Rx | int32;
```

Example 86: OR_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0x348560AB
```

Command

```
r1 = r2 | 0xD2000000;
```

New state

```
r1 = 0xF68560AB
```

4.1.8.2.2 OR_REG

Syntax

```
Rn = Rx | Ry;
```

Example 87: OR_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0x80456576
r3 = 0xF0F0F0F0
```

Command

```
r1 = r2 | r3;
```

New state

```
r1 = 0xF0F5F5F6
```

4.1.8.3 XOR

Syntax	Latency
`Rn = Rx ^ int32;`	1
`Rn = Rx ^ Ry;`	1

Function

Performs a bitwise logical Exclusive OR on the value in Rx and the immediate value or the values in Rx and Ry and places the result in register Rn.

4.1.8.3.1 XOR_IMM

Syntax

```
Rn = Rx ^ int32;
```

Example 88: XOR_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0x348560AB
```

Command

```
r1 = r2 ^ 0xFE456789;
```

New state

```
r1 = 0xCAC00722
```

4.1.8.3.2 XOR_REG

Syntax

```
Rn = Rx & Ry;
```

Example 89: XOR_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0x80456576
r3 = 0xF0F0F0F0
```

Command

```
r1 = r2 & r3;
```

New state

```
r1 = 0x80459A76
```

4.1.9 Control Flow Instructions

4.1.9.1 JUMP

Syntax	Latency
`jump(int32);`	5 branch delay slots
`jump(Rn);`	5 branch delay slots

Function

Continues program execution at a specified 32-bit program memory destination address.

jump(int32);

Jumps to an absolute program memory address specified by a label. The assembler and the linker calculate the destination address from the label.

jump(Rn);

Jumps to a program memory address specified by register Rn.

4.1.9.1.1 JUMP_IMM

Syntax

```
jump(int32);
```

Example 90: JUMP_IMM Command

Current state
```
PC = 0x00000001
```

Command
```
jump(0x12345678);
```

New state
```
After 5 branch delay slots
PC = 0x12345678
```

4.1.9.1.2 JUMP_REG

Syntax
```
jump(Rn);
```

Example 91: JUMP_REG Command

Current state
```
PC = 0x00000001
r1 = 0x87654321
```

Command
```
jump(r1);
```

New state
```
After 5 branch delay slots
PC = 0x87654321
```

4.1.9.2 Jump Subroutine

Syntax	Latency
`jsr(Rn, int32);`	5 branch delay slots

Function

Jumps to the subroutine location in program memory that is given by the instruction's effective address (label). The return address is stored in register `Rn`. To return from the subroutine use the `jump` instruction.

4.1.9.2.1 JSR

Syntax
```
jsr(Rn, int32);
```

Example 92: JSR Command

Current state
```
PC = 0x00000001
r1 = 0xBBBBBBBB
```

Command
```
jsr(r1, 0x12345678);
```

New state
```
After 5 branch delay slots
PC = 0x12345678
r1 = (0x00000001 + t + 1)
```

4.1.9.3 HALT

Syntax
```
halt;
```

Function
Halts the processor. This idle state is only abandoned when receiving an interrupt.

4.1.9.3.1 HALT

Syntax
```
halt;
```

Example 93: HALT Command

Current state
```
running
```

Command
```
halt;
```

New state
```
process only the next command
```

4.1.10 Test Instructions

In a test instruction the execution of the entire instruction depends on the specified condition (`cond`).

Syntax
```
if (cond) command;
```

The following table lists code to be used in conditions. A conditional execution

consists of a single command line which is processed on two adjacent slots.

Table 10: Conditions for Test Instructions

Condition	Description
`Rx == int32;`	equal
`Rx == Ry;`	equal
`Rx != int32;`	not equal
`Rx != Ry;`	not equal
`Rx < int32;`	less than
`Rx < Ry;`	less than
`Rx <= int32;`	less than or equal
`Rx <= Ry;`	less than or equal
`Rx > int32;`	larger than
`Rx > Ry;`	larger than
`Rx >= int32;`	larger than or equal
`Rx >= Ry;`	larger than or equal
`Rx (us)< int32;`	less than, unsigned
`Rx (us)< Ry;`	less than, unsigned
`Rx (us)<= int32;`	less than or equal, unsigned
`Rx (us)<= Ry;`	less than or equal, unsigned
`Rx (us)> int32;`	larger than, unsigned
`Rx (us)> Ry;`	larger than, unsigned
`Rx (us)>= int32;`	larger than or equal, unsigned
`Rx (us)>= Ry;`	larger than or equal, unsigned
`set(Rx, int32);`	bit set
`set(Rx, Ry);`	bit set
`!set(Rx, int32));`	bit not set
`!set(Rx, Ry);`	bit not set

4.1.10.1 TST_EQ_IMM

Syntax

```
Rx == int32;
```

Example 94: TST_EQ_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
```

Command

```
if(r1 == 0x12345678)
      r2 = 0x00001234;
```

New state

```
r2 = 0xBBBBBBBB
```

4.1.10.2 TST_EQ_REG

Syntax

```
Rx == Ry;
```

Example 95: TST_EQ_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
r3 = 0xBBBBBBBB
```

Command

```
if(r1 == r3)
      r2 = 0x00001234;
```

New state

```
r2 = 0x00001234
```

4.1.10.3 TST_NEQ_IMM

Syntax

```
Rx != int32;
```

Example 96: TST_NEQ_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
```

Command

```
if(r1 != 0x12345678)
      r2 = 0x00001234;
```

New state

```
r2 = 0x00001234
```

4.1.10.4 TST_NEQ_REG

Syntax

```
Rx != Ry;
```

Example 97: TST_NEQ_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
r3 = 0xBBBBBBBB
```

Command

```
if(r1 != r3)
      r2 = 0x00001234;
```

New state

```
r2 = 0xAAAAAAAA
```

4.1.10.5 TST_LT_IMM

Syntax

```
Rx < int32;
```

Example 98: TST_LT_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
```

Command

```
if(r1 < 0x12345678)
      r2 = 0x00001234;
```

New state

```
r2 = 0x00001234
```

4.1.10.6 TST_LT_REG

Syntax

```
Rx < Ry;
```

Example 99: TST_LT_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
r3 = 0x87654321
```

Command

```
if(r1 < r3)
      r2 = 0x00001234;
```

New state

```
r2 = 0xAAAAAAAA
```

4.1.10.7 TST_LTE_IMM

Syntax

```
Rx <= int32;
```

Example 100: TST_LTE_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
```

Command

```
if(r1 <= 0x12345678)
      r2 = 0x00001234;
```

New state

```
r2 = 0x00001234
```

4.1.10.8 TST_LTE_REG

Syntax

```
Rx <= Ry;
```

Example 101: TST_LTE_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
r3 = 0xBBBBBBBB
```

Command

```
if(r1 <= r3)
      r2 = 0x00001234;
```

New state

```
r2 = 0x00001234
```

4.1.10.9 TST_GT_IMM

Syntax

```
Rx > int32;
```

Example 102: TST_GT_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
```

Command

```
if(r1 > 0x12345678)
      r2 = 0x00001234;
```

New state

```
r2 = 0xAAAAAAAA
```

4.1.10.10 TST_GT_REG

Syntax

```
Rx > Ry;
```

Example 103: TST_GT_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
r3 = 0x87654321
```

Command

```
if(r1 > r3)
      r2 = 0x00001234;
```

New state

```
r2 = 0x00001234
```

4.1.10.11 TST_GTE_IMM

Syntax

```
Rx >= int32;
```

Example 104: TST_GTE_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
```

Command

```
if(r1 >= 0x12345678)
      r2 = 0x00001234;
```

New state

```
r2 = 0xAAAAAAAA
```

4.1.10.12 TST_GTE_REG

Syntax

```
Rx >= Ry;
```

Example 105: TST_GTE_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
r3 = 0x87654321
```

Command

```
if(r1 >= r3)
      r2 = 0x00001234;
```

New state

```
r2 = 0x00001234
```

4.1.10.13 TST_ULT_IMM

Syntax

```
Rx (us)< int32;
```

Example 106: TST_ULT_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
```

Command

```
if(r1 (us)< 0x12345678)
      r2 = 0x00001234;
```

New state

```
r2 = 0xAAAAAAAA
```

4.1.10.14 TST_ULT_REG

Syntax

```
Rx (us)< Ry;
```

Example 107: TST_ULT_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
r3 = 0x87654321
```

Command

```
if(r1 (us)< r3)
      r2 = 0x00001234;
```

New state

```
r2 = 0xAAAAAAAA
```

4.1.10.15 TST_ULTE_IMM

Syntax

```
Rx (us)<= int32;
```

Example 108: TST_ULTE_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
```

Command

```
if(r1 (us)<= 0xBBBBBBBB)
      r2 = 0x00001234;
```

New state

```
r2 = 0x00001234
```

4.1.10.16 TST_ULTE_REG

Syntax

```
Rx (us)<= Ry;
```

Example 109: TST_ULTE_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
r3 = 0x87654321
```

Command

```
if(r1 (us)<= r3)
      r2 = 0x00001234;
```

New state

```
r2 = 0xAAAAAAAA
```

4.1.10.17 TST_UGT_IMM

Syntax

```
Rx (us)> int32;
```

Example 110: TST_UGT_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
```

Command

```
if(r1 (us)> 0x12345678)
      r2 = 0x00001234;
```

New state

```
r2 = 00001234
```

4.1.10.18 TST_UGT_REG

Syntax

```
Rx (us)> Ry;
```

Example 111: TST_UGT_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
r3 = 0x87654321
```

Command

```
if(r1 (us)> r3)
      r2 = 0x00001234;
```

New state

```
r2 = 0x00001234
```

4.1.10.19 TST_UGTE_IMM

Syntax

```
Rx (us)>= int32;
```

Example 112: TST_UGTE_IMM Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
```

Command

```
if(r1 (us)>= 0x12345678)
      r2 = 0x00001234;
```

New state

```
r2 = 00001234
```

4.1.10.20 TST_UGTE_REG

Syntax

```
Rx (us)>= Ry;
```

Example 113: TST_UGTE_REG Command

Current state

```
r1 = 0xBBBBBBBB
r2 = 0xAAAAAAAA
r3 = 0xBBBBBBBB
```

Command

```
if(r1 (us)>= r3)
      r2 = 0x00001234;
```

New state

```
r2 = 0x00001234
```

4.1.10.21 TST_BIT_SET_IMM

Syntax

```
set(Rx, int32);
```

Example 114: TST_BIT_SET_IMM Command

Current state

```
r1 = 0x12345678
r2 = 0xAAAAAAAA
```

Command

```
if(set(r1, 0x00000003))
      r2 = 0x00001234;
```

New state

```
r2 = 0x00001234
```

4.1.10.22 TST_BIT_SET_REG

Syntax

```
set(Rx, Ry);
```

Example 115: TST_BIT_SET_REG Command

Current state

```
r1 = 0x12345678
r2 = 0xAAAAAAAA
r3 = 0x00000007
```

Command

```
if(set(r1, r3))
      r2 = 0x00001234;
```

New state

```
r2 = 0xAAAAAAAA
```

4.1.10.23 TST_BIT_NSET_IMM

Syntax

```
!set(Rx, int32);
```

Example 116: TST_BIT_NSET_IMM Command

Current state

```
r1 = 0x12345678
r2 = 0xAAAAAAAA
```

Command

```
if(!set(r1, 0x00000003))
      r2 = 0x00001234;
```

New state

```
r2 = 0xAAAAAAAA
```

4.1.10.24 TST_BIT_NSET_REG

Syntax

```
!set(Rx, Ry);
```

Example 117: TST_BIT_NSET_REG Command

Current state

```
r1 = 0x12345678
r2 = 0xAAAAAAAA
r3 = 0x00000007
```

Command

```
if(!set(r1, r3))
      r2 = 0x00001234;
```

New state

```
r2 = 0x00001234
```

4.2 SIMD Extensions

SIMD extensions add 16-bit arithmetic to the **CHILI** that allows to use one type of instruction with multiple data items in each of the processor's 32-bits wide slots.

4.2.1 Arithmetic Instructions

4.2.1.1 Add16

Syntax	Latency
`Rn = add16(Rx, int32);`	1
`Rn = add16(Rx, Ry);`	1

Function

Adds the values in the registers Rx and Ry or the value in Rx and an immediate value and places the result in register Rn.

4.2.1.1.1 ADD16_REG

Syntax

```
Rn = add16(Rx, Ry);
```

Example 118: ADD16_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00010001
r3 = 0x000F000F
```

Command

```
r2 = add16(r1, r3);
```

New state

```
r2 = 0x00100010
```

4.2.1.1.2 ADD16_IMM

Syntax

```
Rn = add16(Rx, int32);
```

Example 119: ADD16_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00010001
```

Command

```
r2 = add16(r1, 0x00200020);
```

New state

```
r2 = 0x00210021
```

4.2.1.2 Sub16

Syntax	Latency
`Rn = sub16(Rx, int32);`	1
`Rn = sub16(Rx, Ry);`	1
`Rn = sub16(int32, Rx);`	1

Function

Subtracts an immediate value or the value in register Ry from the value in register Rx and places the result in the destination register.

4.2.1.2.1 SUB16_IMM

Syntax

```
Rn = sub16(Rx, int32);
```

Example 120: SUB16_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00000000
```

Command

```
r2 = sub16(r1, 0x00010001);
```

New state

```
r2 = 0xFFFFFFFF
```

4.2.1.2.2 SUB16_REG

Syntax

```
Rn = sub16(Rx, Ry);
```

Example 121: SUB16_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00050005
r3 = 0x00010001
```

Command

```
r2 = sub16(r1, r3);
```

New state

```
r2 = 0x00040004
```

4.2.1.2.3 IMM_SUB16

Syntax

```
n = sub16(int32, Rx);
```

Example 122: IMM_SUB16 Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00020002
```

Command

```
r2 = sub16(0x00000000, r1);
```

New state

```
r2 = 0xFFFEFFFE
```

4.2.1.3 Mul16

Syntax	Latency
`Rn = mul16(Rx, int32);`	3
`Rn = mul16(Rx, Ry);`	3

Function

Multiplies an immediate value or the value of register Ry with the value in register Rx and places the result in register Rn.

4.2.1.3.1 MUL16_IMM

Syntax

```
Rn = mul16(Rx, int32);
```

Example 123: MUL16_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00003456
```

Command

```
r2 = mul16(r1, 0x11223344);
```

New state

```
r2 = 0x000008D8
```

4.2.1.3.2 MUL16_REG

Syntax

```
Rn = mul16(Rx, Ry);
```

Example 124: MUL16_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0xFF323400
r3 = 0x00453467
```

Command

```
r2 = mul16(r1, r3);
```

New state

```
r2 = 0xC87AEC00
```

4.2.1.4 Mac16

Syntax	Latency
`Rn = mac16(Rx, int32);`	1
`Rn = mac16(Rx, Ry);`	1

Function

Performs a multiply-accumulate operation for each 16-bit field as follows:
Rn(31..16) <- Rn(31..16) + Rx(31..16)*Ry(31..16),
Rn(15..0) <- Rn(15..0) + Rx(15..0)*Ry(15..0)

4.2.1.4.1 MAC16_IMM

Syntax

```
Rn = mac16(Rx, int32);
```

Example 125: MAC16_IMM Command

Current state

```
r2 = 0x00050005
r1 = 0x00020002
```

Command

```
r2 = mac16(r1, 0x00020002);
```

New state

```
r2 = 0x00090009
```

4.2.1.4.2 MAC16_REG

Syntax

```
Rn = mac16(Rx, Ry);
```

Example 126: MAC16_REG Command

Current state

```
r2 = 0x00030003
r1 = 0x00FF00FF
r3 = 0x00020002
```

Command

```
r2 = mac16(r1, r3);
```

New state

```
r2 = 0x02010201
```

4.2.1.5 Clip16

Syntax	Latency
`Rn = clip16(int16, int16);`	1
`Rn = clip16(Rx, Ry);`	1

Function

Clips the value of register Rn into the boundaries given either by two immediate values or the values in registers Rx and Ry and places the result in register Rn.

4.2.1.5.1 CLIP16_IMM

Syntax

```
Rn = clip16(int16, int16);
```

Example 127: CLIP16_IMM Command[5]

Current state

```
r2 = 0x02123456
```

Command

```
r2 = clip16(0x0200, 0x0259);
```

// clips r2 in the boundaries of the clip region

New state

```
r2 = 0x02120259
```

4.2.1.5.2 CLIP16_REG

Syntax

```
Rn = clip16(Rx, Ry);
```

[5] Border position clip(floor, ceil)

Example 128: CLIP16_REG Command[6]

Current state

```
r2 = 0xFFCC0022
r1 = 0xFFFE0023
r3 = 0x02451159
```

Command

```
r2 = clip16(r1, r3);
```

// clips r2 in the boundaries of the clip region

New state

```
r2 = 0xFFCC0023
```

4.2.1.6 Alro16

Syntax	Latency
`Rn = alro16(Rx, int32);`	1
`Rn = alro16(Rx, Ry);`	1

Function

Performs a 32-bit alignment and rounding operation as given by `Rn <-(Rx+2^(int32-1))>>int32`.

4.2.1.6.1 ALRO16_IMM

Syntax

```
Rn = alro16(Rx, int32);
```

Example 129: ALRO16_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00080018
```

Command

```
r2 = alro16(r1, 0x00040004);
```

New state

```
r2 = 0x00010020
```

4.2.1.6.2 ALRO16_REG

Syntax

```
Rn = alro16(Rx, Ry);
```

[6] Border position clip(floor, ceil)

Example 130: ALRO16_REG Command

Current state
```
r2 = 0xBBBBBBBB
r1 = 0xF00800F8
r3 = 0x00040006
```

Command
```
r2 = alro16(r1,r3);
```

New state
```
r2 = 0xFF010004
```

4.2.1.7 Abs16

Syntax	Latency
`Rn = abs16(int32);`	1
`Rn = abs16(Rx);`	1

Function
Determines the absolute value of the operand and places the result in register Rn.

4.2.1.7.1 ABS16_IMM

Syntax
```
Rn = abs16(int32);
```

Example 131: ABS16_IMM Command

Current state
```
r1 = 0xBBBBBBBB
```

Command
```
r1 = abs16(0x12341234);
```

New state
```
r1 = 0x12341234
```

4.2.1.7.2 ABS16_REG

Syntax
```
Rn = abs16(Rx);
```

Example 132: ABS16_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0xFFECFFEC
```

Command

```
r2 = abs16(r1);
```

New state

```
r2 = 0x00140014
```

4.2.1.8 Absdiff16

Syntax	Latency
`Rn = absdiff16(Rx, int32);`	1
`Rn = absdiff16(Rx, Ry);`	1

Function

Subtracts the immediate value `int32` or the value of register `Ry` from register `Rx` and sets the result as absolute value in register `Rn`.

4.2.1.8.1 ABSDIFF16_IMM

Syntax

```
Rn = absdiff16(Rx, int32);
```

Example 133: ABSDIFF16_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0xFFFEFFFE
```

Command

```
r2 = absdiff16(r1, 0x00030004);
```

New state

```
r2 = 0x00050006
```

4.2.1.8.2 ABSDIFF16_REG

Syntax

```
Rn = absdiff16(Rx, Ry);
```

Example 134: ABSDIFF16_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x00020002
r3 = 0x00040006
```

Command

```
r2 = absdiff16(r1, r3);
```

New state

```
r2 = 0x00020004
```

4.2.1.9 Sad16

Syntax	Latency
`Rn = sad16(Rx, int32);`	1
`Rn = sad16(Rx, Ry);`	1

Function

Calculates the sum of absolute differences for each 16-bit field as follows:
Rn(31..16) <- Rn(31..16) + abs(Rx(31..16)-Ry(31..16)),
Rn(15..0) <- Rn(15..0) + abs(Rx(15..0) -Ry(15..0))

4.2.1.9.1 SAD16_IMM

Syntax

```
Rn = sad16(Rx, int32);
```

Example 135: SAD16_IMM Command

Current state

```
r2 = 0x00010001
r1 = 0xFFFEFFFE
```

Command

```
r2 = sad16(r1, 0x00030003);
```

New state

```
r2 = 0x00060006
```

4.2.1.9.2 SAD16_REG

Syntax

```
Rn = sad16(Rx, Ry);
```

Example 136: SAD16_REG Command

Current state

```
r2 = 0xFFFEFFFE
r1 = 0x00030003
r3 = 0x00010001
```

Command

```
r2 = sad16(r1, r3);
```

New state

```
r2 = 0x00000000
```

4.2.2 Shift Instructions

4.2.2.1 Shr16

Syntax	Latency
`Rn = shr16(Rx, int32);`	1
`Rn = shr16(Rx, Ry);`	1

Function

Arithmetically right-shifts the operand in register Rx by the 32-bit value in register Ry or by the 32-bit immediate value. The shifted result is placed in register Rn.

4.2.2.1.1 SHR16_IMM

Syntax

```
Rn = shr16(Rx, int32);
```

Example 137: SHR16_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x34853456
```

Command

```
r2 = shr16(r1, 0x00040004);
```

New state

```
r2 = 0x03480345
```

4.2.2.1.2 SHR16_REG

Syntax

```
Rn = shr16(Rx, Ry);
```

Example 138: SHR16_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x80454045
r3 = 0x00040004
```

Command

```
r2 = shr16(r1, r3);
```

New state

```
r2 = 0xF8040404
```

4.2.2.2 Shl16

Syntax	Latency
`Rn = shl16(Rx, int32);`	1
`Rn = shl16(Rx, Ry);`	1

Function

Arithmetically left-shifts the operand in register Rx by the 32-bit value in register Ry or by the 32-bit immediate value. The shifted result is placed in register Rn.

4.2.2.2.1 SHL16_IMM

Syntax

```
Rn = shl16(Rx, int32);
```

Example 139: SHL16_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x34853456
```

Command

```
r2 = shl16(r1, 0x00040004);
```

New state

```
r2 = 0x48504560
```

4.2.2.2.2 SHL16_REG

Syntax

```
Rn = shl16(Rx, Ry);
```

Example 140: SHL16_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x80456576
r3 = 0x00050004
```

Command

```
r2 = shl16(r1, r3);
```

New state

```
r2 = 0x08A05760
```

4.2.2.3 Shrl16

Syntax	Latency
`Rn = shrl16(Rx, int32);`	1
`Rn = shrl16(Rx, Ry);`	1

Function

Logically right-shifts the operand in register `Rx` by the 32-bit value in register `Ry` or by the 32-bit immediate value. The shifted result is placed in register `Rn`.

4.2.2.3.1 SHRL16_IMM

Syntax

```
Rn = shrl16(Rx, int32);
```

Example 141: SHRL16_IMM Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x348560AB
```

Command

```
r2 = shrl16(r1, 0x00040004);
```

New state

```
r2 = 0x0348060A
```

4.2.2.3.2 SHRL16_REG

Syntax

```
Rn = shrl16(Rx, Ry);
```

Example 142: SHRL16_REG Command

Current state

```
r2 = 0xBBBBBBBB
r1 = 0x80456576
r3 = 0x00050004
```

Command

```
r2 = shrl16(r1, r3);
```

New state

```
r2 = 0x04020657
```

4.2.3 Formatting Instructions

4.2.3.1 Load Perm

Syntax	Latency
`Rn = ldpe(Rx, int16);`	1

Function

Loads the value of Rx into one of four 32-bit dedicated registers, which hold permute information (PermReg0 - PermReg3) and are indexed by the immediate value, and returns the current value to Rn.

4.2.3.1.1 LDPE

Syntax

```
Rn = ldpe(Rx, int16);
```

Example 143: LDPE Command

Current state

```
PermReg0 = 0x00000000
PermReg1 = 0x00120012
PermReg2 = 0x00000000
PermReg3 = 0x00000000

r2 = 0xBBBBBBBB
r1 = 0x0A0000A0
```

Command

```
r2 = ldpe(r1, 0x0002);
```

New state

```
PermReg0 = 0x00000000
PermReg1 = 0x00120012
PermReg2 = 0x0A0000A0
PermReg3 = 0x00000000

r2 = 0x00000000
```

4.2.3.2 Permute

Syntax	Latency
`Rn = perm(Rx, Ry, int32);`	1

Function

This instruction takes 8-bit-sized parts from Rx or Ry and copies it in dependence of `int32` into Rn.

Description

The immediate value has following meaning:
`int32` = AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH, where AAAA specifies the source of Rn+1(31..24), BBBB specifies Rn+1(23..16), and so on. HHHH specifies Rn(7..0). Thus `int32` specifies the new arrangement. Each 4-bit item (EEEE to HHHH) is given as follows:
0000 (0)... Rn(N)=Rx(7..0)
0001 (1)... Rn(N)=Rx(15..8)
0010 (2)... Rn(N)=Rx(23..16)
0011 (3)... Rn(N)=Rx(31..24)
0100 (4)... Rn(N)=Ry(7..0)
0101 (5)... Rn(N)=Ry(15..8)
0110 (6)... Rn(N)=Ry(23..16)
0111 (7)... Rn(N)=Ry(31..24)
1000 (8)... Rn(N)=0 // unsigned extension
1001 (9)... Rn(N)=sign(Rn(N-1) // signed extension of the previous 8-bit field
1010 (A)... Rn(N)=Rn(N) // keep unchanged
Special case to retrieve the permute information from PermReg registers instead of using the immediate value. The pattern at position HHHH applies to Rz(N) and

DDDD applies to Rn(N+1).
HHHH = 1011 (B) fetch permute information from PermReg0(15..0)
HHHH = 1100 (C) fetch permute information from PermReg1(15..0)
HHHH = 1101 (D) fetch permute information from PermReg2(15..0)
HHHH = 1110 (E) fetch permute information from PermReg3(15..0)
Bits 15 down to 5 of the immediate value will be ignored if HHHH matches one of the four cases described previously.
DDDD = 1011 (B) fetch permute information from PermReg0(31..16)
DDDD = 1100 (C) fetch permute information from PermReg1(31..16)
DDDD = 1101 (D) fetch permute information from PermReg2(31..16)
DDDD = 1110 (E) fetch permute information from PermReg3(31..16)
Bits 31 down to 20 of the immediate value will be ignored if DDDD matches one of the four cases described previously.
The PermReg registers shall only allow patterns in the range from 0x0 to 0xA. Thus a recursion in the PermReg registers is prevented.

4.2.3.2.1 PERM

Syntax

```
Rn = perm(Rx, Ry, int32);
```

Example 144: PERM Command

Current state

```
PermReg0 = 0x00000000
PermReg1 = 0x00670A87
PermReg2 = 0x00000000
PermReg3 = 0x00000000

r1 = 0xAAAABBBB
r2 = 0xCCCCDDDD
r60 = 0x55551111
r61 = 0x44449999
```

Command

```
r60 = perm(r1, r2, 0x5A72687C);
```

New state

```
PermReg0 = 0x00000000
PermReg1 = 0x00120012
PermReg2 = 0x0A0000A0
PermReg3 = 0x00000000

r60 = 0xBB5500CC
r61 = 0xDD44CCAA
```

CHAPTER 5

Preprocessor

In This Chapter

5.1 Preprocessing Directives

Preprocessor instructions can be embedded in the source code. The source code will be altered *before* assembly. Preprocessor instructions start with a hash (#). Preprocessing enables the following instructions before assembling:

- Replace tokens in the current file with specified replacement tokens
- Embed files within the current file
- Conditionally assemble sections of the current file

5.1.1 Details about Preprocessing

The following table describes the actions the preprocessor performs on the source code.

Table 11: Preprocessor Directives

Preprocessing Directive	Description
Token	A token is a series of characters delimited by white space.
White space	The following white spaces are allowed on a preprocessor directive: space, horizontal tab, vertical tab, form feed and comments.
# token	Preprocessor directives begin with the # token followed by a preprocessor keyword. The # token must appear as the first character that is not white space on a line. The # token is not part of the directive name and can be separated from the name by white spaces.
End of preprocessor directive	A preprocessor directive ends at the new-line character unless the last character of the line is the “\” (backslash) character. In the case of a backslash being the last character in the preprocessor line, the preprocessor interprets the “\” and the new-line character as a continuation marker.

5.2 Preprocessing Operations

Every source file is preprocessed by the assembler before regular assembling. The following operations are carried out:

- New-line characters are introduced as needed to replace system-dependent end-of-line characters, and other system-dependent character-set translations are performed as needed. Trigraph sequences are replaced by equivalent single characters.
- Each “\” (backslash) followed by a new line character is deleted, and the next source line is appended to the line that contained the backslash.
- The source file is decomposed into preprocessing tokens and sequences of white space. A single white space replaces each comment. A source file cannot end with a partial token or comment.
- Preprocessing directives are run, and macros are expanded.
- Escape sequences in character constants and string literals are replaced by their equivalent values.
- Adjacent string literals are concatenated.

The rest of the assembly process operates on the preprocessor output.

Appendix A: Instruction Set Quick Reference

Transfer Instructions

`Rn = int32;`	`Rn = Rx;`

Core Memory Read Instructions

`Rn = port8/16/32[int32];`	`Rn = sport8/16[int32];`
`Rn = port8/16/32[Rx];`	`Rn = sport8/16[Rx];`
`Rn = port8/16/32[Rx + int32];`	`Rn = sport8/16[Rx+int32];`
`Rn = port8/16/32[Rx + Ry];`	`Rn = sport8/16[Rx+Ry];`

Core Memory Write Instructions

`port8/16/32[int32] = Rn;`	`port8/16/32[int32] = int12;`
`port8/16/32[Rx] = Rn;`	`port8/16/32[Rx] = int12;`
`port8/16/32[Rx + int32] = Rn;`	`port8/16/32[Rx + int32] = int12;`
`port8/16/32[Rx + Ry] = Rn;`	

Peripheral Port Read Instructions

`Rn = pport32[int32];`	`Rn = pport32[Rx + int32];`
`Rn = pport32[Rx];`	`Rn = pport32[Rx + Ry];`

Peripheral Port Write Instructions

`pport32[int32] = Rn;`	`pport32[int32] = int12;`
`pport32[Rx] = Rn;`	`pport32[Rx] = int12;`
`pport32[Rx + int32] = Rn;`	`pport32[Rx + int32] = int12;`
`pport32[Rx + Ry] = Rn;`	

Arithmetic Instructions

`Rn = Rx + int32;`	`Rn = Rx * int32;`
`Rn = Rx + Ry;`	`Rn = Rx * Ry;`
`Rn = Rx – int32;`	`Rn = clip(int16, int16);`
`Rn = int32 - Rx;`	`Rn = clip(Rx, Ry);`
`Rn = Rx – Ry;`	`Rn = absdiff(Rx, int32);`
`Rn = sex(Rx, int32);`	`Rn = absdiff(Rx, Ry);`
`Rn = sex(Rx, Ry);`	`Rn = alro(Rx, int32);`
`Rn = abs(int32);`	`Rn = alro(Rx, Ry);`
`Rn = abs(Rx);`	`Rn = leadbits (Rx, int32)`
`Rn = -Rx;`	

Shift Instructions

`Rn = Rx >> int32;`	`Rn = Rx << Ry;`
`Rn = Rx >> Rx;`	`Rn = Rx >>> int32;`
`Rn = Rx << int32;`	`Rn = Rx >>> Ry;`

Logical Instructions

`Rn = Rx & int32;`	`Rn = Rx | Ry;`
`Rn = Rx & Ry;`	`Rn = Rx ^ Ry;`
`Rn = Rx | int32;`	`Rn = Ry ^ int32;`

Control Flow Instructions

`jump(int32);`	`jsr(Rn, int32);`
`jump(Rn);`	`halt;`

Test Instructions

`Rx == int32;`	`Rx (us) < int32;`
`Rx == Ry;`	`Rx (us) < Ry;`
`Rx != int32;`	`Rx (us) <= int32;`
`Rx != Ry;`	`Rx (us) <= Ry;`
`Rx < int32;`	`Rx (us) > int32;`
`Rx < Ry;`	`Rx (us) > Ry;`
`Rx <= int32;`	`Rx (us) >= int32;`
`Rx <= Ry;`	`Rx (us) >= Ry;`
`Rx > int32;`	`set(Rx, Ry);`
`Rx > Ry;`	`set(Rx, int32);`
`Rx >= int32;`	`!set(Rx, Ry);`
`Rx >= Ry;`	`!set(Rx, int32);`

SIMD Instructions

Rn = add16(Rx, int32);	Rn = absdiff16(Rx, int32);
Rn = add16(Rx, Ry);	Rn = absdiff16(Rx, Ry);
Rn = sub16(Rx, int32);	Rn = sad16(Rx, int32);
Rn = sub16(Rx, Ry);	Rn = sad16(Rx, Ry);
Rn = sub16(int32, Rx);	Rn = shr16(Rx, int32);
Rn = mul16(Rx, int32);	Rn = shr16(Rx, Ry);
Rn = mul16(Rx, Ry);	Rn = shl16(Rx, int32);
Rn = mac16(Rx, int32);	Rn = shl16(Rx, Ry);
Rn = mac16(Rx, Ry);	Rn = shrl16(Rx, int32);
Rn = clip16(int16, int16);	Rn = shrl16(Rx, Ry);
Rn = clip16(Rx, Ry);	Rn = ldpe(Rx, int16);
Rn = alro16(Rx, int32);	Rn = perm(Rx, Ry, int32);
Rn = alro16(Rx, Ry)	
Rn = abs16(int32);	
Rn = abs16(Rx);	

Appendix B: Preprocessor Commands

The following table describes briefly the functions offered by the preprocessor:

Table 12: Preprocessor Commands

Function	Description
`#define`	Defines a preprocessor macro.
`#if`	Conditionally includes or surpasses parts of source code, depending on the result of a constant expression.
`#ifdef`	Conditionally includes source text if a macro name is defined.
`#ifndef`	Conditionally includes source text if a macro name is not defined.
`#else`	Conditionally includes source text if the previous #if, #ifdef, #ifndef or #elif test fails.
`#elif`	Conditionally includes sources text if the previous #if, #ifdef, #ifndef or #elif test fails.
`#endif`	Ends conditional text.